# The Formal Semantics of Program Slicing for Non-Terminating Computations

Martin Ward and Hussein Zedan

**Abstract**—Since the original development of program slicing in 1979 [1] there have been many attempts to define a suitable semantics which will precisely define the meaning of a slice. Particular issues include handling termination and non-termination, slicing non-terminating programs and slicing nondeterministic programs. In this paper we review and critique the main attempts to construct a semantics for slicing and present a new operational semantics which correctly handles slicing for non-terminating and nondeterministic programs. We also present a modified denotational semantics which we prove to be equivalent to the operational semantics. This provides programmers with two different methods to prove the correctness of a slice or a slicing algorithm, and means that the program transformation theory and FermaT transformation system, developed over the last 25 years of research, and which has proved so successful in analysing terminating programs, can now be applied to non-terminating interactive programs.

## 1 Introduction

THE traditional denotational semantics treats a program as a "black box" which is provided with an initial state on which it starts execution, processes for a while and finally either terminates in a final state or continues processing forever without terminating. If the program is deterministic then, on termination, each initial state leads to a unique final state, while a nondeterministic program may have more than one possible final state for each initial state. Since the semantics "abstracts away" the internal sequence of operations of the program, it can say nothing about a non-terminating program, apart from the fact that it does not terminate. Typically, non-termination is defined using a special "final state", $\perp$ which does not define values for any variables.

Combined with weakest preconditions expressed in infinitary logic [2], this approach has proved very powerful in the development of a theory of program transformations which has been applied successfully to program development, refinement, reverse engineering and software migration [2,3,4,5,6,7,8,9,10,11,12].

However, for a program which interacts with its environment, such as an embedded system, the program may produce useful output before it terminates, and may indeed never terminate: in fact, non-termination may well be a requirement. With an interactive system, we are interested in the sequence of interactions with the environment, and not just the final state. We would still like to "abstract away" everything that happens in between these interactions. In other words, we are interested in the values of certain variables of interest at certain points of interest in the program. This is precisely the semantics of *program slicing*: with the proviso that we are interested in slicing non-terminating programs.

Program slicing is a program analysis technique for extracting a relevant sub-program, or slice, from a larger program. The slice must replicate part of the behaviour of the original program: specifically, the values of the variables of interest at the points of interest. A slice is not, in general, required to preserve *all* the behaviour of the original program.

Since the original development of program slicing by Mark Weiser in 1979 [1,13], and its application as a debugging technique [14], the concept has proved also to have applications in testing, parallelisation, integration, software safety, program understanding and software maintenance. Surveys of slicing algorithms, applications, variations and results can be found in a number of papers [15,16,17,18,19].

As we will see in Section 2, *any* slicing relation which is behaviour preserving and allows deletion of irrelevant code must satisfy the following properties:

1) It must be semantic equivalence where the original program terminates, and;
2) Allow any program as a slice when the original program does not terminate.

These two results are sufficient to completely characterise the semantic part of the slicing relation. In [20] we describe this relation as *semi-refinement* since it is weaker than semantic equivalence but stronger than semantic refinement.

This would seem to completely solve the question of which semantic relation to use for slicing, and indeed it does: but only for slicing at the *end* of the program.

For slicing in the middle of a program, there is the problem that the program may continue to execute and not terminate *after* the slice point in which case the denotational semantics of the program is **abort** and any slice is allowed. For an interactive, non-terminating program, a denotational semantics does not capture the interactions with the environment. In particular, for programs which are intended to run forever (such as embedded systems and operating systems), the semi-refinement relation of [20] cannot be used, as it stands, to define slicing. In Section 4 we present a new

- *Martin Ward is a Reader in Software Engineering at De Montfort University, Leicester, UK*
- *Hussein Zedan is Professor and Dean of Research and Graduate Studies at Applied Science University in Bahrain*

semantic relation for slicing which captures the interactions of a program with the environment, and therefore allows a program to interact before entering a non-terminating loop, or in the middle of a non-terminating loop. We will show that this relation formalises Weiser slicing not just for end-slicing but also when slicing anywhere in the program. Note that even with this new relation, a statement which does not terminate *and* does not include a slice point can still be semi-refined to any statement.

The new semantics is *operational* in the sense that it defines the meaning of a program in terms of the finite or infinite sequence of states that the program passes through as it operates. An operational semantics can be difficult to work with in practical applications of reverse engineering because the semantics of the original program is one sequence of states, while the reverse engineered program may have a shorter, or even completely different sequence of states. With a denotational semantics, which defines the meaning of a program in terms of the initial and final states, the semantics of the reverse engineered program is *identical to* the semantics of the original program. Proving this identity is in general simpler than proving the more complicated equivalence relation which holds for the operational semantics.

We therefore also define a denotational semantics, which is an extension of the standard denotational semantics formed by constructing the denotational semantics of an annotated program. The annotations are carefully designed to capture the values of the variables of interest at each slice point and ensure that the program terminates after a certain number of passes through each slice point. This ensures that the values of the variables of interest at each slice point can reach the final state. Note that the annotated program may still be non-terminating: loops which contain no slice point are not forced to terminate after a certain number of iterations.

This annotation method allows us to re-use the large body of work in developing and proving the correctness of program transformations which has been accumulated over many years of research [2,3,4,5,6,7,8,9,10,11,12]. The FermaT program transformation system [9] can also be used for the analysis of interactive systems: as we will show in Section 7.1.

In Section 5 we prove that the two semantics (operational and denotational) are equivalent in the sense that either semantics can be derived from the other. Therefore, they define the same slicing relation, and this relation fully characterises “Weiser slicing”. This equivalence means that slices can be derived using either semantics: depending on which is most convenient for the problem at hand. It also gives us confidence that the informal notion of a “slice” has been correctly formalised: there are some subtle points in the definition of the denotational semantic annotations which were only uncovered during the development of the proof.

### 1.1 Outline of the Paper

In Section 2 we discuss Weiser’s definition of slicing and show that his informal description is sufficient to fully characterise the slicing relation.

In Section 3 we discuss the relationship between slicing algorithms and the semantics for slicing and survey the many attempts over the years to pin down the informal concept of a program slice in a formal semantic definition.

In Section 4 we present the formal definitions for the operational and denotational semantics and the corresponding slicing relations. Readers who prefer to start with an informal discussion of the semantics can skip to Section 6 and come back to this section. We extend the definition of a slicing criterion to include multiple labels with a separate set of variables of interest at each label. This extension is necessary because a slice for several labels cannot necessarily be constructed from the set of slices, one for each label: see Section 6.4. We also present an extension of the usual denotational semantics, formed by annotating the program, which can be used for analysing non-terminating interactive programs and for slicing non-terminating programs.

In Section 5 we prove the equivalence of slicing defined by the operational semantics and slicing defined by the denotational semantics on an annotated version of the program.

In Section 6 we illustrate the semantics with a number of example programs: in particular, the examples which proved so problematic for previous attempts to define a semantics for slicing. We also present a larger example of an interactive non-terminating program, which is typical of code translated from assembler, and show how the FermaT Migration Engine (FME) can be used to analyse the interactive behaviour of this program by adding annotations to convert it to a terminating program, transforming the program and then removing the annotations.

Section 7 briefly discusses some practical applications of semantic slicing.

Finally, Section 8 concludes.

## 2 Background

In this section we discuss Weiser’s definition of slicing and deduce certain properties of the slicing relation which turn out to be sufficient to fully characterise the relation.

### 2.1 Weiser Slice

Weiser defined a slice as a “reduced executable program which preserves part of the behaviour of the original program” [13]. The slicing relation is therefore a combination of a syntactic relation and a semantic relation. The syntactic relation expresses the fact that the slice is formed from the original program by deleting statements. In [20] we formalise the concept of deleting statements in the “reduction relation”. Program $\mathbf{S}_2$ is a reduction of $\mathbf{S}_1$, denoted $\mathbf{S}_2 \sqsubseteq \mathbf{S}_1$ whenever $\mathbf{S}_2$ can be constructed from $\mathbf{S}_1$ by replacing statements by **skip** statements. A **skip** statement has no effect and is simply used as a “placeholder” to indicate where a statement in $\mathbf{S}_1$ was deleted, and this makes it trivially easy to match up components of $\mathbf{S}_2$ with the corresponding components of $\mathbf{S}_1$. The semantic relation expresses the fact that the slice preserves part of the behaviour of the original program. In the literature, [19,20,21], slices which preserve both relations are referred to as *syntactic slices* while slices which preserve only the semantic relation are *semantic slices*. One application of semantic slicing is in the analysis of interactive systems where we are interested in the semantic slices on the points where the program interacts with its environment, another is in analysing the normal behaviour of a program after "slicing away" error handling code (see Section 7).

Weiser did not give a formal definition of the semantic relation, but from his discussion of slicing certain properties can be deduced. In his proof of the non-existence of an algorithm to find minimal slices, Weiser presented the program fragment in Figure 1 and wrote: "Imagine slicing on the value of X at line 5. An algorithm to find a statement-minimal slice would include line 3 if and only if the function before line 3 did halt. Thus such an algorithm could determine if an arbitrary program could halt, which is impossible" [13]. It is clear from this example that Weiser intended for slicing to be applied to programs which may fail to terminate on certain initial states, and that when constructing a slice, it is valid to delete code which appears after a non-terminating loop. Otherwise, deleting line 3 in his example would never be valid: but his proof that there does not exist an algorithm for finding a statement-minimal slice hinges on the fact that line 3 can be deleted exactly when the preceding code does not halt. Weiser also clearly intended that any code which does not involve the variables of interest can be deleted, regardless of whether or not that code terminates.

```
1  read(X)
2  if (X)
         then
         ...
         perform any function not involving X here
         ...
3        X := 1
4  else  X := 2 endif
5  write(X)
```

Figure 1. Weiser's Example Program

A simple example of a statement which terminates and does not involve $X$ is the statement **skip** which terminates immediately without affecting any variable. A simple example of a statement which does not terminate and also does not involve $X$ is the loop **while true do skip od**. So, according to Weiser, it is incorrect to delete the statement labelled $L_2$ in Figure 2 but it is allowable to delete the statement labelled $L_4$ in Figure 3.

```
read($X$)
if $X$
  then $L_1$: skip;
       $L_2$: $X := 1$
  else $X := 2$ fi;
write($X$)
```

Figure 2. Weiser Slicing Example $W_1$

```
read($X$)
if $X$
  then $L_3$: while true do skip od;
       $L_4$: $X := 1$
  else $X := 2$ fi;
write($X$)
```

Figure 3. Weiser Slicing Example $W_2$

In the rest of this section we will formalise the properties of slicing implied by Weiser's discussion and deduce certain constraints that they place on any proposed semantic relation for it to be considered a formalisation of "Weiser slicing". We consider any slice which satisfies these properties to be a "valid slice" and any definition of slicing which allows *only* valid slices, and *all* valid slices, to be a valid definition.

Initially, our focus is on slicing at the end of the program, later we will extend our scope to slicing in the middle of potentially non-terminating programs. We start with some definitions:

***Definition 2.1.*** A statement **S** is *$x$-preserving*, if the value of variable $x$ is unchanged over any execution of **S**. A statement is *locally* $x$-preserving if every assignment to the variable $x$ (if any) is $x$-preserving.

Any locally $x$-preserving statement is also $x$-preserving. Also, any statement which does not assign to $x$ is trivially locally $x$-preserving, and therefore also $x$-preserving.

We formalise Weiser's concept of preserving the values of the variables of interest via the concept of a *state*:

***Definition 2.2.*** A *state* is a function from a set of variables (the *domain* or *state space* of the state) to a set of values. So a state defines a value for each variable in the current domain.

The state containing the variables $x_1, x_2, \dots, x_n$ having values $e_1, e_2, \dots, e_n$ respectively may be written:

$$\{x_1 \mapsto e_1, x_2 \mapsto e_2, \dots, x_n \mapsto e_n\}$$

***Definition 2.3.*** Given a semantic equivalence relation $\approx$ and a set $X$ of variables, the *restriction* of $\approx$ to $X$, denoted $\approx_X$ is the equivalence relation defined by taking all variables apart from those in $X$ out of all the final states and comparing the result.

Note that if $\mathbf{S}_1 \approx \mathbf{S}_2$ then for any $X$, we have $\mathbf{S}_1 \approx_X \mathbf{S}_2$. Also, if $\mathbf{S}_1 \approx_X \mathbf{S}_2$ and $X$ contains all the variables in $\mathbf{S}_1$ and $\mathbf{S}_2$, then $\mathbf{S}_1 \approx \mathbf{S}_2$.

***Definition 2.4.*** A programming language and its semantics is *potentially divergent* if, for any given set $X$ of variables of interest: there is a program **abort**$(X)$ which is locally $x$-preserving for all $x \in X$ and which does not terminate for any $x \in X$. The formal definition is that for any program **S**:

$$\mathbf{S};\ \textbf{abort}(X)\ \approx_X\ \textbf{abort}(X)$$

The definition simply states that a potentially divergent language is any language in which it is possible to write an infinite loop. Informally: since we are slicing on the end of the program, if control flow never reaches the end of the program then there is no behaviour which needs to be preserved and therefore any statements in the program can be deleted. In C the statement:

```
while (1) {
}
```

is a suitable implementation of **abort**$(X)$ for any set $X$, and in Java:

```
while (true) {
}
```

is suitable. In assembler language we can use a branch instruction to create an infinite loop:

```
FOO      B   FOO
```

where the instruction branches unconditionally to itself.

In the lazy semantics of Danicic et al [22,23] we can define **abort**$(X)$ as

$$\textbf{while true do } x_1 := x_1;\ x_2 := x_2;\ \ldots;\ x_n := x_n \textbf{ od}$$

This contains assignments for all the variables $x_1, \ldots, x_n$ in $X$, but is still locally $x$-preserving for all of these variables, since each variable is assigned its current value. See Section 3.2.3 for a discussion of this semantics.

Regular expressions, finite state machines and Hofstadter's BlooP language [24] are languages in which all loops are guaranteed to terminate, so these are not potentially divergent. (They are also not Turing-complete).

Let $R_X$ be a slicing relation, in the sense that $\mathbf{P}\ R_X\ \mathbf{S}$ if and only if **S** is a valid slice of **P** when we are slicing on the set $X$ of variables at the end of the program.

As mentioned earlier Weiser's definition of slicing can be split into two parts:

1) A *syntactic part*: the slice **S** must be formed from the program **P** by deleting statements, or equivalently by replacing statements by **skip** statements. This relation is denoted as $\mathbf{S} \sqsubseteq \mathbf{P}$.
2) A *semantic part*: the slice **S** must preserve the values of the variables of interest at the points of interest.

We are interested in the semantic part of the slicing definition. Let $\preccurlyeq_X$ be a semantic relation on statements. This is any partial order relation on the semantics of the programs. We say that $\preccurlyeq_X$ *partially defines* $R_X$ if and only if:

$$\mathbf{P}\ R_X\ \mathbf{S} \iff \mathbf{S} \sqsubseteq \mathbf{P} \wedge \mathbf{P} \preccurlyeq_X \mathbf{S}$$

If **P** terminates for some initial state then $\mathbf{P}\ R_X\ \mathbf{S}$ implies $\mathbf{P} \approx_X \mathbf{S}$ for that initial state, since the slice has to preserve the values of all variables in $X$.

***Definition 2.5.*** A *truncating* slicing relation is one which allows deletion of a statement at the end of a program whenever that statement does not modify any variable of interest. Arguably, this is the simplest possible example of deleting irrelevant code. A formal definition of the property is the following: if **P** is of the form $\mathbf{S}_1;\ \mathbf{S}_2$ and $\mathbf{S}_2$ is $x$-preserving, for all $x \in X$, then $\mathbf{P}\ R_X\ \mathbf{S}_1$

In order to preserve program behaviour, the semantic slicing relation must be semantic equivalence (on the variables of interest) when the original program terminates. This raises the question of what is allowed as a valid slice when the original program does not terminate.

The next theorem shows that, in any potentially divergent programming language, if the slicing relation allows deletion of irrelevant code and we are slicing at the end of the program then the semantic relation which partially defines the slicing relation will allow *any* program as a valid slice of a non-terminating program.

***Theorem 2.6.*** Any semantic partial order relation $\preccurlyeq_X$ which partially defines a truncating slicing relation $R_X$ on a potentially divergent language and semantics, is such that for all statements **S**:

$$\textbf{abort}(X) \preccurlyeq_X \mathbf{S}$$

**Proof:** Let $R_X$ be any truncating slicing relation on a potentially divergent language. Let $\approx_X$ be the semantic equivalence relation defined by the semantics on the final values of the variables in $X$. Let **S** be any statement.

Consider the program $\mathbf{S};\ \textbf{abort}(X)$. By truncation, **S** is a valid slice of $\mathbf{S};\ \textbf{abort}(X)$, so:

$$\mathbf{S};\ \textbf{abort}(X)\ R_X\ \mathbf{S} \tag{1}$$

and therefore:

$$\mathbf{S};\ \textbf{abort}(X) \preccurlyeq_X \mathbf{S} \tag{2}$$

By potential divergence, we have:

$$\mathbf{S};\ \textbf{abort}(X)\ \approx_X\ \textbf{abort}(X) \tag{3}$$

By substituting 3 into 2 we have:

$$\textbf{abort}(X) \preccurlyeq_X \mathbf{S}$$

So, **S** is a valid slice of **abort**$(X)$. ■

At first sight, this result may seem counter-intuitive: but if control flow never reaches the slice point (which here is the end of the program) then there is no behaviour which the slice is required to preserve. If the slicing relation is not required to preserve anything, then it is not restricted (semantically) in any way.

Putting these results together, we see that *any* slicing relation which is behaviour preserving and allows deletion of irrelevant code must:

1) Be semantic equivalence where the original program terminates, and;
2) Allow any program as a slice when the original program does not terminate (the proof of Theorem 2.6 applies to any non-terminating statement, not just **abort**$(X)$).

These two results are sufficient to completely characterise the semantic part of the slicing relation. In [20] we describe this relation as *semi-refinement* since it is weaker than semantic equivalence but stronger than semantic refinement.

# 3 Related Work

In this section we survey a number of different approaches to the definition of a formal semantics for slicing and evaluate whether they characterise Weiser slicing.

## 3.1 Slicing Algorithms

Weiser's algorithm for constructing slices [1] makes use of program dependencies. Horwitz and Reps PDG (Program Dependence Graph) algorithm [25] is essentially Weiser's algorithm extended to handle interprocedural slicing. A refinement of Weiser's algorithm is presented in [26] which takes into account the context in which each procedure call occurs, and also attempts to determine which procedure parameters are needed in the slice.

These algorithms for constructing slices preceded the development of a theory for program slicing: although from the beginning Weiser made a clear distinction between the *definition* of a program slice and the particular slice constructed by a particular *algorithm*. Originally, it was never suggested that "a slice" should be defined as simply "what the slicing algorithm produces". This distinction can be clearly seen in Weiser's paper [13] in which he proves that there is no algorithm for finding *minimal* slices (see Section 2.1 above).

More recently, algorithms for computing slices of reactive programs [27] and finite state machines [28] have been developed and these required handling programs which may not terminate. This has led to new definitions and algorithms for computing various forms of control dependence in reactive systems which make extensive use of exception handling [29,30].

Halder and Cortei [31] present an improved slicing algorithm, based on the work of Mastroeni and Zanardini [32], which uses semantics-based data dependencies to disregard some false dependencies in order to give more precise slices. They claim that a slice on a set of variables $V$ can be formed from the union of the slices on each variable in $V$. However, De Lucia et al [33] present a simple counterexample to show that in general, the union of two slices is not necessarily a slice. According to Horwitz et al. [34] two *program dependence graph based* slices of the same program can be seen as two non-interfering versions of the program and therefore can be safely integrated. However, it is not clear whether this result still holds for Halder and Cortei's improved dependence-based slices. In Section 6.4 we present an example (Example 2) which shows that the union of two slices for the same variable at two different slice points is not necessarily a slice for the variable on bot slice points simultaneously. Therefore for a fully comprehensive definition of slicing it is necessary to define a slice in terms of a *set* of slice points with a *set* of variables at each slice point (where each slice point may include a different set of variables).

This work on extending or refining the definitions of control and data dependency, while relevant to the construction of slicing *algorithms* is not relevant to the *definition* of a slice, so will not be discussed further.

As we will see in Section 6.5.1, any definition of slicing based on dependencies will, in certain cases, include statements that have no effect on the variables of interest: so any definition of slicing based on program dependency will therefore be incomplete in the sense that it will exclude valid slices.

## 3.2 Semantics for Program Slicing

There have been many attempts to define the semantic relationship involved in program slicing. Some researchers simply ignored the possibility that a program may not terminate: for example, Venkatesh [35] wrote "As program slices are considered meaningful for terminating computations only, we will omit that technical detail (non-termination) from the semantic function definitions in this paper." This approach is clearly insufficient when we wish to consider slicing on interactive systems, such as operating systems and embedded systems, which are designed to be non-terminating. It is also insufficient for program analysis where we are attempting to analyse an unknown program and may not know in advance under what conditions the program terminates.

Binkley et al [23] claim that "Weiser deliberately left unspecified the behaviour of slices in states where the original program fails to terminate". Weiser may not have spelled out explicitly the behaviour of a slice in those circumstances, but, as discussed above, he did make it clear [13] that it is valid for a slicing algorithm to delete code which appears *after* a non-terminating loop. As we saw in Section 2.1, this stipulation is sufficient to fully determine the behaviour of end slices in states where the original program fails to terminate. Weiser's informal description of a slice therefore implicitly contains enough information to fully determine the semantic relation of slicing. So it turns out that Weiser did not leave anything unspecified.

Note that the slicing *relation* is not necessarily a one-to-one relation: there may be many different programs which are valid slices of a given program. (In particular, any program should be considered a valid slice of itself under any slicing criterion). This is another reason why it is not suitable to define a slice as the output of any particular slicing algorithm. Even though there are only a finite number of potential syntactic slices, no algorithm can produce a list of all valid (and only valid) slices: since by picking the smallest program in the list, we would have an algorithm for a statement-minimal slice: which Weiser proved to be impossible. Since there is no algorithm which can generate all valid slices (and only valid slices), it is not possible to define a valid slice as the output of any particular algorithm. Also, when we are analysing interactive systems, semantic slices (which do not need to be constructed from the original program by deleting statements) may be more useful. There are infinitely many different semantic slices for any given program, so no algorithm can compute them all.

In the rest of this section we will examine and critique the various attempts to define a semantics for program slicing.

### 3.2.1 *Reps and Yang's Definition*

Reps and Yang [36] *define* a slice in terms of data and control dependencies. In effect they make the semantics of a program be precisely the set of program dependencies, as computed via a particular dependency-tracking algorithm. For Weiser's example in Figure 1, their definition implies that line 3 *must* be in the slice, regardless of whether or not the function before line 3 terminates: because their dependency tracker lists line 3 as a dependency. So this definition does not completely characterise Weiser slicing. The main results of their paper are to prove that the PDG (Program Dependency Graph) slice preserves the behaviour of the original program on the variables of interest, and that the slice terminates whenever the original program terminates. They note that a slice may terminate when the original program diverges: so PDG slices do not necessarily preserve non-termination.

### *3.2.2 Binkley and Gallagher's Definition*

Binkley and Gallagher's survey of program slicing [16] defines a slice as follows:

***Definition 3.1.*** For statement $s$ and variable $v$, the slice **S** of program **P** with respect to the slicing criterion $\langle s;\ v\rangle$ is any executable program with the following properties:

1) **S** can be obtained by deleting zero or more statements from **P**.
2) If **P** halts on input $I$, then the value of $v$ at statement $s$ each time $s$ is executed in **P** is the same in **P** and **S**. If **P** fails to terminate normally $s$ may execute more times in **S** than in **P**, but **P** and **S** compute the same values each time $s$ is executed by **P**.

This definition preserves program behaviour and termination: at least for cases where the slicing point is at the end of the program. It does not require slices to preserve non-termination. In the case of end-slicing (slicing on the value of a variable at the end of the program), the slice may terminate and execute the statement at the end of the program exactly once, when the original program did not terminate. More generally, this definition allows a slice to delete code after a non-terminating loop, and delete the loop itself if the statement $s$ is not part of the loop.

Note that the definition refers to a single variable at a single slice point: as already discussed, we cannot compute (or define) slices of more complex slicing criteria as the union of simpler slices.

Considering only the semantic part of the definition, if we are slicing at the end of the program then property (2) allows *any* statement **S** as a valid slice of **P** in the case where **P** does not terminate.

Also, for a programming language which includes nondeterminism, this definition does not allow *any* potential slice to be valid for a terminating nondeterministic program! Consider the program **P**:

```
if true → $x := 1$
□ true → $x := 2$ fi;
$y := x$
```

where we are slicing on the value of $x$ at the assignment to $y$. According to Binkley and Gallagher's definition, there are *no* valid slices of **P**! Even **P** is not a valid slice of itself: since the value of $x$ at $y := x$ may be different each time $y := x$ is executed. This might appear to be unimportant in practice (since most executable programs are also deterministic), but in the context of program analysis and reverse engineering it is quite common to "abstract away" some implementation details and end up with a nondeterministic abstraction of the original program. If one wishes to carry out further analysis on this program via slicing, then it is essential that the definition of slicing, and the algorithms implementing the definition, are able to cope with nondeterminism. Nondeterminism is discussed by Dijkstra [37,38] and by many other researchers.

Fortunately, the definition can be modified to work with nondeterministic programs. Instead of talking about "the value of $v$ at statement $s$ each time $s$ is executed in **P**" we need to talk about the set of all possible sequences of values that can occur in different nondeterministic executions. We also wish to extend the definition of a slice point from a single variable at a single point in the program to a set of points of interest with a separate set of variables of interest at each point of interest.

### *3.2.3 Lazy Semantics*

Cartwright and Felleisen [39] define a "lazy" semantics for program slicing in which some variables are mapped to the special value $\perp$, representing non-termination, while other variables have ordinary values. The value of a variable is defined as $\perp$ if it is assigned inside an infinite loop, however, a program might assign the variable a new value "after" the infinite loop which allows the variable to "recover" from being $\perp$.

A slice on a set of variables is any program, formed from the original by deleting statements, which is equivalent to the original program on the variables of interest under the lazy semantics.

Under Cartwright and Felleisen's semantics, the program:

```
while true do
  $x := x + 1$;
  $y := y + 1$ od;
$x := 1$;
$z := 1$
```

gives $x$ the final value 1, while the value of $y$ is undefined. For the **while** loop alone, the value of $x$ is undefined: so the program can "recover" from the non-termination caused by the loop if a variable is subsequently assigned a defined value. When slicing on the final value of $z$, the infinite loop and assignment to $x$ can be deleted since they have no effect on the final value of $z$. But the assignment to $z$ itself *cannot* be deleted: even though in practice this statement can never be executed: this contradicts Weiser's concept of slicing as discussed above in Section 2.1. So the lazy semantics does not fully characterise Weiser slicing.

Among some researchers (see the next two subsections) there appears to be a reluctance to allow a slice to delete code which is included by the data dependency algorithm: for example, code which appears after a non-terminating loop. This may be due to a confusion between the *definition* of a slice and the set of slices produced by a particular algorithm, such as the standard dependency algorithm. Because the dependency algorithm includes the assignment to $z$ in the slice, some researchers believe that the semantics ought to also require that the assignment be included. (Weiser did not originate this confusion: as discussed in Section 2.1, he explicitly allowed deleting code after a non-terminating loop.)

Including more statements than strictly necessary in a slice is a minor defect: after all, no algorithm can generate minimal slices for every program, so every algorithm will include "unnecessary" statements under *some* circumstances. (Although, this does not absolve a *definition* of slicing from including extra statements!) However, Cartwright and Felleisen's semantics has a more serious problem: which is illustrated by the programs in Figure 4. Program $P_2$ is constructed from $P_1$ by deleting statements, and according to the lazy semantics, the two programs are semantically equivalent. So Cartwright and Felleisen's definition of slicing allows $P_2$ as a valid slice of $P_1$: even though $P_2$ does not terminate when $y > 0$ initially, but $P_1$ terminates for all initial states.

(a) Program $P_1$

```
while $y > 0$ do
  $y := y + 1$;
  $y := y - 2$ od;
$x := 1$
```

(b) Program $P_2$

```
while $y > 0$ do
  $y := y + 1$ od;
$x := 1$
```

Figure 4. Lazy semantics slicing example

For program $W_2$ in Figure 3, Cartwright and Felleisen's semantics does not allow statement $L_4$ to be deleted, so this semantics does not fully characterise Weiser slicing.

They refer to their work as "the semantics of program dependence" [39]: and it is the case that the traditional program dependence algorithm will include statement $L_4$ in Figure 3 as a dependency of the value of $X$ in the write statement. However, their semantics does *not* fully characterise program dependency: consider the example in Figure 5. With this example, their semantics *does* allow deletion of statement $L_5$, even though the traditional program dependency algorithm includes *both* $L_5$ and $L_6$ as dependencies. So their semantics is only an approximation of the semantics of program dependency.

```
read($X$);
if false then $L_5$:  $X := 1$
         else $L_6$:  $X := 2$ fi;
write($X$)
```

Figure 5. Program Dependency Example

### 3.2.4 *Transfinite Semantics*

Giacobazzi and Mastroeni [40] define an extended compositional semantics which is able to observe transfinite computation: that is, computations that may occur after a given number of infinite loops. They claim that "this generalization is necessary to deal with program manipulation techniques modifying the termination status of programs, such as program slicing." Transfinite semantics is defined in terms of state sequences whose length can be any finite or infinite ordinal. For example, in the program:

```
while true do
  $w := w + 1$;
  while $y \neq z$ do
    $y := y - 1$ od od;
$x := 1$
```

we have an infinite concatenation of finite or infinite traces.

Nestra [41] claims that dataflow slices are "not correct w.r.t. standard semantics" because infinite loops may be sliced away. He shows that in a transfinite semantics there is no need to consider traces resulting from the execution of every statement in an infinite loop: only the states at the bottom of the loop body are needed. Figure 6 illustrates the difference. In Giacobazzi and Mastroeni's semantics, the final value of $x$ is undefined in $p_3$ but 2 in $P_4$. In Nestra's semantics it is 2 in both programs.

(a) Program $P_3$

```
while true do
  $x := 1$;
  $x := 2$ od
```

(b) Program $P_4$

```
while true do

  $x := 2$ od
```

Figure 6. Transfinite semantics examples

However, both of these transfinite semantics allow the non-terminating program $P_2$ in Figure 4 as a valid slice of the terminating program $P_1$.

Neither of these transfinite semantics allow statement $L_4$ in Figure 3 to be deleted, so they do not fully characterise Weiser slicing. Also, both of these transfinite semantics allow statement $L_5$ in Figure 5 to be deleted, so they do not fully characterise program dependency.

### 3.2.5 *Danicic's Non-Standard Semantics*

Danicic et al [42] showed that none of the lazy or transfinite semantics in the previous subsections satisfies the *replacement property*. This property states that any component of a program can be replaced by a semantically equivalent component and the resulting program will be semantically equivalent to the original. The problem is illustrated in Figure 7. In $P_6$ the statement $y := y$ has been replaced by the semantically equivalent statement **skip**. But $P_5$ and $P_6$ are not equivalent under either Cartwright and Felleisen's lazy semantics, Giacobazzi and Mastroeni's transfinite semantics, or Nestra's transfinite semantics. For $P_5$ the value of $y$ is undefined under all three semantics, but for $P_6$ the value of $y$ is 1.

Danicic et al [42] solved this particular problem with a modified lazy semantics which does satisfy the replacement property. This approach does not, however, cater for slicing at arbitrary points in the middle of a program. This semantics also suffers from the problem that a non-terminating

(a) Program $P_5$

```
x := 1;
y := 1;
while true do
   x := −x od;
if x > 0
   then y := y
    else x := −x fi
```

(b) Program $P_6$

```
x := 1;
y := 1;
while true do
   x := −x od;
if x > 0
   then skip
    else x := −x fi
```

Figure 7. The Replacement Property

program is allowed as a valid slice of a terminating program: for example, $P_2$ in Figure 4 is a valid slice of $P_1$ in the semantics of Danicic et al [42]: this is because both programs give $x$ the value 1 in the final state. The fact that $y$ has the value $\perp$ in $P_2$ for some initial states in which $y$ has a terminating value in $P_1$ does not have any impact on the slice.

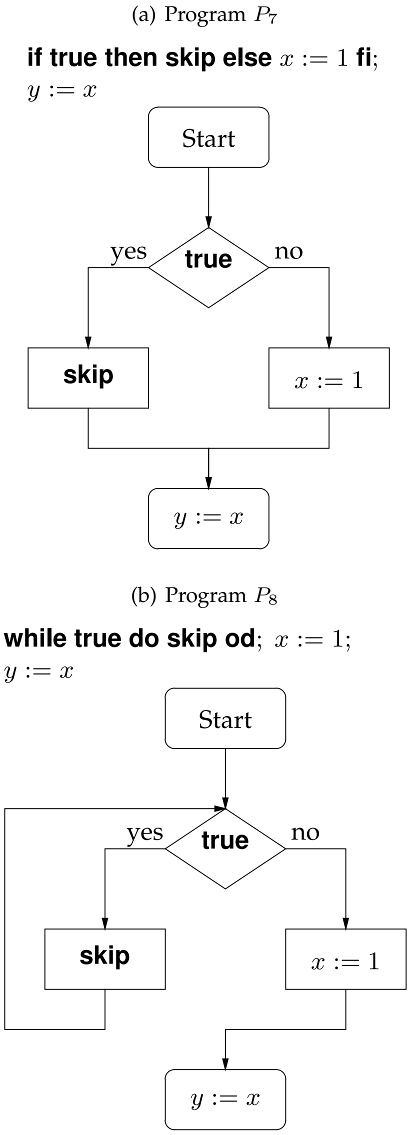
Danicic's semantics does not allow deletion of statement $L_4$ in Figure 3, so it also does not fully characterise Weiser slicing. It allows statement $L_5$ in Figure 5 to be deleted, so it also does not fully characterise program dependency.

A general problem with all forms of lazy and transfinite semantics is that they do not correspond with how a program is actually executed on a computer. In these semantic models, a program somehow continues to execute after an infinite number of iterations of a loop (or even after an infinite number of executions of infinite loops!). As discussed above, this means that the slicing definition will require that a slice preserves parts of the program which are not actually reachable in any execution. The problem is even more obvious when we examine the control flow graphs of the programs. Nodes in the flowgraph which are only reached via the "no" exit from a decision node whose test is the predicate **true** *can* be deleted if the decision node resulted from an **if** statement, but *cannot* be deleted if the decision node resulted from a **while** loop! For example, in Figure 8, where we are slicing on the final value of $y$, the statement $x := 1$ can be deleted from $P_7$ but not from $P_8$, even though neither statement can ever be executed. In both cases the statement appears in the "no" branch of a test of "**true**".

### 3.2.6 *Slicing Based on Semantic Equivalence*

Kamkar [43] defined slicing according to strict semantic equivalence. In this context, a *strict* semantic equivalence is one which distinguishes between terminating and non-terminating code (in contrast to lazy semantics, for example).

This form of slicing therefore preserves the behaviour of the program and also preserves both termination and non-termination of the original program. This will lead to larger slices than necessary in some cases. For example, a loop which has no effect on the variables of interest will have to be included in the slice if there is the possibility that the loop may not terminate: since this non-termination has to be preserved. As a "knock-on" effect, code which affects variables referenced in the loop *also* has to be included in the slice, even if this code always terminates and has no effect on the variables of interest.



(a) Program $P_7$

**if true then skip else** $x := 1$ **fi**;
$y := x$

(b) Program $P_8$

**while true do skip od**; $x := 1$;
$y := x$

Figure 8. Unreachable Code

This semantics does, at least, allow code *after* a non-terminating loop to be deleted, unlike the various forms of transfinite and lazy semantics, and does not allow a non-terminating program as a valid slice of a terminating program. For the program in Figure 3, based on Weiser's example, Kamkar's semantics allows statement $L_4$ to be deleted, but does not allow statement $L_3$ to be deleted. So this semantics does not fully characterise Weiser slicing.

Harman et al [23,44,45,46,47] present what at first sight appears to be a very general framework which defines a "program slice" as a combination of two relations:

1) A syntactic partial order relation (typically specified as a *computable* relation); and
2) A semantic equivalence relation.

They define an *amorphous slice* as a slice which only preserves the semantic equivalence relation.

The definition looks generic enough to include many useful slicing relations: but in practice, insisting that the semantic relation be an *equivalence* is too severe a restriction. In [48] we prove that given very reasonable property of the programming language and a very reasonable property of slicing, the only equivalence relation which defines a slicing relation is *universal equivalence*. This is the relation which defines every program as "equivalent" to every other program: so slices defined in terms of this equivalence relation do not preserve the behaviour of the original program on the variables of interest.

To prove this result, we use the statement **abort**$(X)$ discussed above (Section 2.1). In the lazy semantics of Danicic et al [22,23] we can define **abort**$(X)$ as

$$\textbf{while true do } x_1 := x_1;\ x_2 := x_2;\ \ldots;\ x_n := x_n \textbf{ od}$$

which contains assignments for all the variables $x_1, \ldots, x_n$ in $X$, but is still locally $x$-preserving for all of these variables, since each variable is assigned its current value.

*Definition 3.2.* A semantic equivalence relation $\approx$ *partially defines* a slicing relation $R$ if whenever **S** is a slice of **P**, according to $R$, on variable set $X$, then **P** and **S** are equivalent according to $\approx_X$, ie:

$$\mathbf{P}\ R_X\ \mathbf{S} \quad \Rightarrow \quad \mathbf{P} \approx_X \mathbf{S}$$

Note that programs may be equivalent which are not slices of each other, but that every program is equivalent to all its slices (on the restriction of the equivalence relation to $X$).

*Definition 3.3.* The *universal equivalence relation* is the semantic equivalence relation $\overset{\mathrm{U}}{\approx}$ for which any two programs are equivalent, i.e. for all statements $\mathbf{S}_1$ and $\mathbf{S}_2$:

$$\mathbf{S}_1 \overset{\mathrm{U}}{\approx} \mathbf{S}_2$$

*Theorem 3.4.* Any equivalence relation $\approx$ which partially defines a truncating slicing relation $R$ on a potentially divergent language and semantics, is the universal equivalence relation.

**Proof:** Let $R$ be any truncating slicing relation on a potentially divergent language. Let $\approx$ be any semantic equivalence relation which partially defines $R$. Let **S** be any statement and $X$ be any set of variables. Let $R_X$ be the subset of $R$ when we are slicing on $X$ at the end of the program.

Consider the program **S**; **abort**$(X)$.

By the truncation property, **S** is a valid slice of the sequence **S**; **abort**$(X)$, i.e:

$$\mathbf{S};\ \textbf{abort}(X)\ R_X\ \mathbf{S}$$

Since $\approx$ partially defines $R$, we have:

$$\textbf{abort}(X);\ \mathbf{S} \approx_X \mathbf{S}$$

By potential divergence we have:

$$\mathbf{S};\ \textbf{abort}(X) \approx_X \textbf{abort}(X)$$

Therefore, by the transitivity of $\approx_X$:

$$\mathbf{S} \approx_X \textbf{abort}(X)$$

But this holds for any statement **S**. So for any statements $\mathbf{S}_1$ and $\mathbf{S}_2$:

$$\mathbf{S}_1 \approx_X \textbf{abort}(X) \quad \text{and} \quad \mathbf{S}_2 \approx_X \textbf{abort}(X)$$

So by the symmetry and transitivity of $\approx$:

$$\mathbf{S}_1 \approx_X \mathbf{S}_2$$

But this is true for every $X$: in particular, for the set which contains all variables in $\mathbf{S}_1$ and $\mathbf{S}_2$. So:

$$\mathbf{S}_1 \approx \mathbf{S}_2$$

But this is true for all statements $\mathbf{S}_1$ and $\mathbf{S}_2$, so $\approx$ is therefore the universal equivalence relation. ■

This theorem shows that if we are working in a language in which it is possible to write an infinite loop, and we want to allow slices to delete irrelevant code at the end of a program, then the only equivalence relation we can use to define slices is universal equivalence. Therefore, if we also want our slices to preserve the behaviour of the program on the variables of interest, then there is *no* equivalence relation which defines the slicing relation!

In terms of Figures 2 and 3, the theorem shows that any semantic equivalence relation which allows statement $L_4$ in Figure 3 to be deleted will *also* allow statement $L_2$ in Figure 2 to be deleted: and so will *not* preserve the value of $X$ at the statement write$(X)$. So there is no semantic equivalence relation which fully characterises Weiser slicing.

It is clear from this discussion that the semantic relation for slicing *cannot* be an equivalence relation, and we need to look at more general relations.

After reading the proof, this result may appear to be an obvious one which we have belaboured more than necessary. However, since the concept of an "amorphous slice" (which is a slice defined in terms of a semantic equivalence relation) has appeared in many journal and conference papers over a period of more than ten years [23,44,45,46, 47], it seemed worthwhile to put the issue to rest once and for all.

### 3.2.7 *Slicing Based on Refinement*

As already mentioned, Weiser's informal definition of a program slice (see Section 2.1) refers to a syntactic relation and a semantic relation. In [8] Ward defined slicing using *reduction* as the syntactic relation and *refinement* as the semantic relation: where the refinement is defined on a projection of the semantics of the program to the variables of interest.

Using refinement as the semantic relation will allow irrelevant code to be deleted, regardless of whether or not the deleted code terminates. Termination of the original program is preserved, but non-termination does not have to be preserved. This semantics allows statements $L_3$ and $L_4$ in example $W_2$ (Figure 3) to be deleted and characterises Weiser slicing for deterministic programs.

A potential problem with this definition of slicing is that it is not behaviour preserving for nondeterministic programs. For example the nondeterministic program:

$x := 1;$<br>
**if true** $\to x := 1$<br>
□ **true** $\to x := 2$ **fi**

always terminates and assigns $x$ the value 1 or 2 nondeterministically. A reduction of this program, which is also a refinement, is:

$x := 1$

which always assigns the value 1. A programmer cannot analyse a slice of the program and then deduce facts about the value assigned to a variable in the original program (assuming that the original program terminates). In the example above, the slice always sets $x$ to the value 1, but the original program may set $x$ to the value 2.

This problem was recognised and fixed in a later formalisation of slicing by using the semantic relation of *semi-refinement* [20], see the next section.

### *3.2.8 Slicing Based on Semi-Refinement*

To fix the problems caused by defining slicing in terms of refinement, we invented a new semantic relation: *semi-refinement* [10,20,49]. Program $\mathbf{S}_2$ is a semi-refinement of $\mathbf{S}_1$, written $\mathbf{S}_1 \preccurlyeq \mathbf{S}_2$, provided $\mathbf{S}_1$ and $\mathbf{S}_2$ are semantically equivalent whenever $\mathbf{S}_1$ terminates. To be precise:

1) If $\mathbf{S}_1$ terminates on some initial state, then $\mathbf{S}_2$ also terminates, and the set of possible final states for $\mathbf{S}_2$ (on this initial state) is identical to the set of possible final states for $\mathbf{S}_1$;
2) If $\mathbf{S}_1$ does not terminate, then $\mathbf{S}_2$ can do anything at all.

In terms of a denotational semantics which maps each initial state to the set of possible final states, if $f_1$ is the semantics for $\mathbf{S}_1$ and $f_2$ is the semantics for $\mathbf{S}_2$, then $\mathbf{S}_1 \preccurlyeq \mathbf{S}_2$ iff:

$$\forall s.\, (\perp \in f_1(s) \ \vee \ f_2(s) = f_1(s))$$

where $\perp$ is the special state denoting non-termination.

Semi-refinement can equivalently be defined in terms of weakest preconditions (see [20]):

For any program $\mathbf{S}$ and condition $\mathbf{R}$ on the final state, the *weakest precondition* $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ is the weakest condition on the initial state such that if $\mathbf{S}$ is started in a state satisfying this condition, then it is guaranteed to terminate in a state satisfying $\mathbf{R}$. Using weakest preconditions:

$$\mathbf{S}_1 \preccurlyeq \mathbf{S}_2 \quad \text{iff} \quad \mathbf{S}_1 \approx \{\mathrm{WP}(\mathbf{S}_1, \textbf{true})\};\ \mathbf{S}_2$$

Here, the statement $\{\mathrm{WP}(\mathbf{S}_1, \textbf{true})\}$ is an *assertion*. This is an executable statement, not an annotation. In general the assertion $\{\mathbf{B}\}$ is equivalent to **skip** when the condition $\mathbf{B}$ is true, and **abort** when $\mathbf{B}$ is false. So the assertion statement $\{\mathbf{B}\}$ is in effect a conditional **abort**:

**if B then skip else abort fi**

If $\mathbf{S}_1$ does not terminate, then $\mathrm{WP}(\mathbf{S}_1, \textbf{true})$ is false and the assertion $\{\mathrm{WP}(\mathbf{S}_1, \textbf{true})\}$ is **abort**. In this case, both sides of the equivalence are **abort**, and the equivalence is satisfied, whatever the semantics of $\mathbf{S}_2$. If $\mathbf{S}_1$ does terminate, then $\mathrm{WP}(\mathbf{S}_1, \textbf{true})$ is **true** and the assertion is a **skip** statement. In this case, we must have $\mathbf{S}_1$ semantically equivalent to $\mathbf{S}_2$.

In order to preserve program behaviour, the semantic slicing relation must be semantic equivalence (on the variables of interest) when the original program terminates. As discussed above, when we are slicing on the end of the program, the slicing relation must allow any program as a valid slice of a non-terminating program. The semi-refinement relation therefore steers a path between the Scylla of semantic equivalence and the Charybdis of unrestricted refinement.

In this paper we extend the concept of semi-refinement in order to handle slicing in the middle of potentially non-terminating nondeterministic programs. This extension allows the method to be used to analyse the interaction behaviour of non-terminating embedded systems.

### *3.2.9 Finite Trajectory Semantics*

The question of defining the meaning of a program slice for non-terminating programs led to the development of a *finite trajectory semantics* by Barraclough et al [50].

The finite trajectory semantics is defined in terms of a small, deterministic programming language which includes labels on assignments, skip statements and tests. The program statements are (using our notation):

- $L$ : **skip**
- $L : x := e$
- $\mathbf{S}_1;\ \mathbf{S}_2$
- **if** $L$ : **B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**
- **while** $L$ : **B do** $\mathbf{S}_1$ **od**

where $L$ is a label, $x$ is a variable, $e$ is an expression, $\mathbf{B}$ is a boolean expression and $\mathbf{S}_1$ and $\mathbf{S}_2$ are statements. Note that both statements and predicates can be labelled. A *quotient* of a program is obtained by replacing sub-programs in the original program by **skip** statements: so a quotient is what we have previously defined as a *reduction*. (See Section 3.2.7 and [20]).

The *overriding* operator on states is defined as follows. For any states $s_1$ and $s_2$, the state $(s_1 \circledast s_2)$ is defined on a variable $x$ as:

$$(s_1 \circledast s_2)(x) \ =_{\text{DF}} \begin{cases} s_2(x) & \text{if } x \text{ is in the domain of } s_2 \\ s_1(x) & \text{otherwise} \end{cases}$$

Given an integer $n$ and statement $\mathbf{S}$, the trajectory semantics $\vec{\mathcal{T}}_n[\![\mathbf{S}]\!]$ maps each initial state to a *finite* sequence of labelled states, defined as follows:

$$\begin{aligned} \vec{\mathcal{T}}_n[\![L : \textbf{skip}]\!](s) &=_{\text{DF}} \langle\langle L, s\rangle\rangle \\ \vec{\mathcal{T}}_n[\![L : x := e]\!](s) &=_{\text{DF}} \langle\langle L, s \circledast \{x \mapsto \mathcal{E}[\![e]\!](s)\}\rangle\rangle \\ \vec{\mathcal{T}}_n[\![\mathbf{S}_1;\ \mathbf{S}_2]\!](s) &=_{\text{DF}} \vec{\mathcal{T}}_n[\![\mathbf{S}_1]\!](s) +\!\!\!+ \vec{\mathcal{T}}_n[\![\mathbf{S}_2]\!](s') \end{aligned}$$

where $s'$ is the state in the last element of $\vec{\mathcal{T}}_n[\![\mathbf{S}_1]\!](s)$ and $+\!\!\!+$ denotes concatenation of sequences

$$\vec{\mathcal{T}}_n[\![\textbf{if } L : \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}]\!](s)$$
$$=_{\text{DF}} \begin{cases} \langle\langle L, s\rangle\rangle +\!\!\!+ \vec{\mathcal{T}}_n[\![\mathbf{S}_1]\!](s) & \text{if } \mathcal{E}[\![\mathbf{B}]\!](s) \\ \langle\langle L, s\rangle\rangle +\!\!\!+ \vec{\mathcal{T}}_n[\![\mathbf{S}_2]\!](s) & \text{otherwise} \end{cases}$$

$$\vec{\mathcal{T}}_n[\![\textbf{while } L : \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od}]\!](s)$$
$$=_{\text{DF}} \vec{\mathcal{T}}_n[\![W_n(L, \mathbf{B}, \mathbf{S})]\!](s)$$

where $W_n(L, \mathbf{B}, \mathbf{S})$ is the *nth unfolding* of the **while** loop, defined as follows:

$$\begin{aligned} W_0(L, \mathbf{B}, \mathbf{S}) &=_{\text{DF}} \textbf{if } L : \mathbf{B} \textbf{ then skip else skip fi} \\ W_{n+1}(L, \mathbf{B}, \mathbf{S}) &=_{\text{DF}} \textbf{if } L : \mathbf{B} \textbf{ then } \mathbf{S};\ W_n(L, \mathbf{B}, \mathbf{S}) \\ &\qquad\qquad \textbf{else skip fi} \end{aligned}$$

There are two important points to note about the finite trajectory semantics of the **while** loop, compared to our WSL (Wide Spectrum Language) **while** loop (see [2,9]):

1) There is no equivalent to the WSL **abort** statement, and consequently, no equivalent to the WSL assertion statement;
2) The **while** loop is defined by unfolding, starting with **if** $L$ : $\mathbf{B}$ **then skip else skip fi** as the zeroth unfolding. In our WSL semantics the **while** loop is also defined by unfolding, but starting with **abort**. Starting with **abort** ensures that each higher truncation is a semi-refinement of all the earlier truncations: there is no such simple semantic relationship between the different unfoldings of a loop in the trajectory semantics.

In the trajectory semantics all trajectories are terminating and finite: there are no non-terminating primitive statements and all loops are forced to terminate after $n$ iterations (with control passing to the statement after the loop). A non-terminating program is indicated by the trajectory lengths increasing without bounds as $n$ increases.

Slicing is defined in terms of an equivalence relation on finite trajectories: *finite trajectory backward slice equivalence*, denoted $\vec{\mathcal{S}}_{(V,L)}$ where $V$ is a set of variables of interest and $L$ is a label representing the program point of interest. Informally, programs $P$ and $Q$ are finite trajectory backward slice equivalent, with respect to slicing criterion $(V, L)$, if for each initial state $s$ there exists a sufficiently large integer $n_s$ such that for all $n > n_s$ the traces $\vec{\mathcal{T}}_n[\![P]\!]$ and $\vec{\mathcal{T}}_n[\![Q]\!]$ contain the same number of states labelled $L$ and for each of these states, the variables in $V$ have the same values.

The formal definition makes use of the *projection* of a sequence of labelled states, $\mathsf{Proj}_{(V,L)}$ where:

***Definition 3.5.*** The projection operator $\mathsf{Proj}_{(V,L)}$ is defined:

$$\begin{aligned} \mathsf{Proj}_{(V,L)}(\lambda_1 +\!\!\!+ \lambda_2) &=_{\text{DF}} \mathsf{Proj}_{(V,L)}(\lambda_1) +\!\!\!+ \mathsf{Proj}_{(V,L)}(\lambda_2) \\ \mathsf{Proj}_{(V,L)}(\langle L, s\rangle) &=_{\text{DF}} \langle\langle L, s \downarrow V\rangle\rangle \\ \mathsf{Proj}_{(V,L)}(\langle L', s\rangle) &=_{\text{DF}} \langle\rangle \quad \text{if } L' \neq L \end{aligned}$$

This operator extracts the states which have the required label and restricts the domain of each extracted state to the variables of interest.

Since $\vec{\mathcal{S}}_{(V,L)}$ is a semantic equivalence relation it might be expected to suffer from some of the problems that have plagued all other attempts to define slicing in terms of semantic equivalence. Indeed, Theorem 3.4 shows that *every* semantic equivalence relation will suffer from some problem or other.

Consider the program
**while** $L_1$ : **true do** $L_2$ : **skip od**; $L : x := e$

where we are slicing on the value of $x$ at label $L$. Every finite trajectory consists of a finite sequence of states labelled with $L_1$ or $L_2$ followed by a state labelled $L$. So every projection on $(\{x\}, L)$ includes the state labelled $L$: and therefore the corresponding assignment statement must be included in every slice (even though this statement can never actually be reached in any execution of the program). Similarly, in Figure 3 the trajectory semantics disallows deleting statement $L_4$, even though it appears after an infinite loop. So the trajectory semantics does not fully characterise Weiser slicing.

The authors of [50] criticise the lazy and transfinite semantics because: "They are conceptual models that requires [sic] the reader, for example, to imagine programs continuing to execute after executing infinite loops." As we have just seen, their semantics is open to exactly the same criticism! By forcing every loop to terminate after $n$ iterations, with control flow continuing after the loop, any code appearing after one or more infinite loops will have an effect on the semantics.

However, there is an even more serious problem which is illustrated in Figure 9. Program $P_8$ is obtained from $P_7$ by deleting the statement labelled $L_3$. The finite trajectories of $P_7$ and $P_8$ when projected with respect to slicing criterion $(\{x\}, L)$ are identical for all $n$: both projections contain a single labelled state in which $x$ has the value 1.

(a) Program $P_7$

```
while L1 : y > 0 do
   L2 : y := y + 1;
   L3 : y := y − 2 od;
L : x := 1
```

(b) Program $P_8$

```
while L1 : y > 0 do
   L2 : y := y + 1 od;
L : x := 1
```

Figure 9. Trajectory semantics slicing example

So according to the finite trajectory semantics, $P_8$ is a valid slice of $P_7$. But, as we saw with Figure 4, $P_8$ does not terminate when $y > 0$ initially, while $P_7$ terminates for all initial states. Once again, we have a non-terminating program defined as a valid slice of a terminating program.

# 4 An Improved Semantics for Slicing

In this section we give the formal definitions of the language, the operational and denotational semantics, and our slicing relation which aims to capture Weiser's concept of slicing for both terminating and non-terminating programs, and for deterministic and nondeterministic programs.

## 4.1 Motivation

To handle slicing in the middle of a potentially non-terminating program we need a semantics that is *operational* in the sense that the semantics records the sequence of states as the program executes, not just the final state. To handle nondeterminism, the semantic function maps each initial state to the set of possible finite or infinite sequences of states which can result from execution on the given initial state. Such a sequence of states is called a *history*. To apply program slicing to a history we ensure that each state is labelled, so that we can determine the states which are relevant to the points of interest. To allow simultaneous slicing at multiple points in the program we define a slicing criterion as a function from labels to sets of variables. For each label (point of interest) in the domain of the slicing criterion, the corresponding set of variables is the set of variables of interest at that point. Our *projection function* takes a history and a slicing criterion and removes unwanted states from the history and removes unwanted variables from the remaining states. An important feature of the projection function is that if an infinite sequence of states is removed from the history, then the special state $\perp$ is appended to the result to indicate that the resulting finite sequence of states still represents a non-terminating computation. Note that $\perp$ is unlabelled and can only appear as the final element in a history. The state $\perp$ can appear in the semantics of the assertion statement and in the result of a projection. We define **abort** as the assertion $\{$**false**$\}$ and **skip** as the assertion $\{$**true**$\}$.

Our projection function is therefore significantly different to the one presented in Barraclough et al [50] and discussed in Section 3.2.9.

The semantic relation we use to define slicing is a natural extension of semi-refinement applied to histories. An infinite or finite terminating history can only be semi-refined by itself, while a finite non-terminating history (i.e. one which ends in $\perp$) can be semi-refined by any *extension* of the history: this agrees with the original history up to the final element (the $\perp$) and has $\perp$ replaced by any other history. For example, the history $\langle\langle L_1, s_1\rangle, \langle L_2, s_2\rangle, \perp\rangle$ can be semi-refined to $\langle\langle L_1, s_1\rangle, \langle L_2, s_2\rangle, \langle L_3, s_3\rangle\rangle$. This allows any non-terminating loop which does not include any slice points (and which therefore is projected to $\langle\perp\rangle$) to be sliced to any other statement. (Similarly, **abort** can be sliced to any other statement.) To see why this is necessary, consider the program:

**while true do** $L$: **skip od**; **S**

where **S** is any statement. The semantics of this program on initial state $s$ is the infinite sequence $\langle\langle L, s\rangle, \langle L, s\rangle, \dots\rangle$. If $L$ is not in the slicing criterion, then this history is projected to $\langle\perp\rangle$. Since the loop does not affect any variables used by **S**, it can be sliced away, and the result is the semantics of **S**: which can be anything, since **S** is an arbitrary statement. This ensures that semi-refinement on the operational semantics, together with the syntactic reduction relation, provides a complete and correct formalisation of Weiser slicing which also applies to nondeterministic programs and potentially non-terminating interactive programs.

This operational semantics is intuitively clear, but difficult to work with in practical applications. Therefore we have also developed a method for annotating a program in such a way as to "capture" the values of the variables of interest at the points of interest and ensure that these values can potentially reach the end of the program and therefore appear in the final state. The denotational semantics of this annotated program can then be used to define slicing on nondeterministic and potentially non-terminating programs. To ensure that the information can reach the end of the program, we enforce termination on loops which contain slice points after the slice point has been passed a certain number of times. Note that after this enforced termination the whole program terminates immediately: control does *not* pass to any statements following the loop. This is necessary to ensure that statements *after* a non-terminating loop have no effect on the final state and can be deleted: for example the statement labelled $L_4$ in Figure 3. Even though termination is enforced, there is still a significant difference between an annotated non-terminating loop and an annotated terminating loop: the former cannot pass control to the next statement as the latter does.

In Section 5 we prove that the slicing relation defined by the operational semantics is identical to the slicing relation defined by the denotational semantics applied to an annotated program. Some subtle issues with the precise modification and annotation needed for the denotational semantic approach to be operational semantic approach were only uncovered in the process of proving the equivalence of the two approaches. So this task is decidedly non-trivial and the equivalence proof is a very welcome reassurance.

This proof means that we can apply all the results of the last 25 years of research and development in program transformation theory (which is based on the denotational semantics) to program slicing and analysis of potentially non-terminating and nondeterministic interactive programs.

Note that this proof differs significantly from the standard proof of equivalence of operational and denotational semantics which proves that, for terminating programs, the final state or set of states generated by the operational semantics is equal to the final state or set of states in the denotational semantics: in other words, operational semantics can simulate denotational semantics for terminating programs by ignoring the intermediate states. Our proof shows that a modified denotational semantics can simulate operational semantics, including all intermediate states, or just certain required intermediate states, for both terminating and non-terminating programs.

## 4.2 The Language

The language we are using is based on the Wide Spectrum Language WSL [9] and consists of two primitive statements and four compound statements.

### 4.2.1 Primitive Statements

The primitive statements are restricted to assertions and simple assignments. For any formula **Q**, variable $x$ and expression $e$ the following are primitive statements:

1) **Assertion:** $\{$**Q**$\}$
2) **Assignment:** $x := e$

Note that an assertion is an executable statement, rather than simply an annotation. If the condition **Q** is true, then the assertion does nothing, otherwise it aborts (does not terminate). Any primitive statement can be annotated with a label, for example: $L_1\colon x := 2;\ L_2\colon \{x > 0\}$. Any unlabelled program is assumed to be annotated with the special label $L_0$, if necessary.

As usual, we define the primitive statements **skip** and **abort** as assertions:

$$\textbf{skip} =_{\text{DF}} \{\textbf{true}\} \quad \text{and} \quad \textbf{abort} =_{\text{DF}} \{\textbf{false}\}$$

### *4.2.2 Compound Statements*

The compound statements are as follows: for any statements $\mathbf{S}_1$ and $\mathbf{S}_2$, and any formula $\mathbf{B}$, the following are also statements:

1) **Sequence:** $\mathbf{S}_1;\ \mathbf{S}_2$
2) **Deterministic Choice: if** $\mathbf{B}$ **then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**
3) **Nondeterministic Choice:** $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$
4) **While Loop: while B do S od**

Using the above constructs, we can define all the constructs in Dijkstra's "guarded command language" [37,38]. Similarly, Dijkstra's guarded commands can be used to define this subset of WSL. For example, the assertion $\{\mathbf{B}\}$ can be defined as a conditional **abort** in the guarded command language:

$$\textbf{if } \mathbf{B} \ \rightarrow\ \textbf{skip} \ \Box\ \neg\mathbf{B} \ \rightarrow\ \textbf{abort fi}$$

Conversely, Dijkstra's conditional statement:

$$\textbf{if } \mathbf{B}_1 \ \rightarrow\ \mathbf{S}_1 \ \Box\ \mathbf{B}_2 \ \rightarrow\ \mathbf{S}_2 \textbf{ fi}$$

can be defined as a nested **if** statement:

```
if $\mathbf{B}_1 \wedge \neg\mathbf{B}_2$
  then $\mathbf{S}_1$
  else if $\neg\mathbf{B}_1 \wedge \mathbf{B}_2$
         then $\mathbf{S}_2$
         else if $\mathbf{B}_1 \wedge \mathbf{B}_2$ then $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$
                                   else $\{$false$\}$ fi fi fi
```

So our language, while simpler to Dijkstra's guarded command language, is notationally equivalent. The proofs in Section 5 could easily be extended to Dijkstra's language.

## 4.3 States

A *state* is a collection of variables (the *domain* or *state space*) each of which is assigned a value from a given set $\mathcal{H}$ of values. A state is therefore modelled as a function from the domain to the set of values. For example, the state $\{x \mapsto 0, y \mapsto 1\}$ has domain $\{x, y\}$ and assigns $x$ the value 0 and $y$ the value 1. The special state, denoted $\perp$, does not assign values to any variables but indicates non-termination or an error condition. States other than $\perp$ are *proper states*. The set of all proper states on domain $V$ and value set $\mathcal{H}$ is therefore the set of functions from $V$ to $\mathcal{H}$, which is denoted $\mathcal{H}^V$. The set of all states on $V$ is: $D_{\mathcal{H}}(V) =_{\text{DF}} \{\perp\} \cup \mathcal{H}^V$.

A *state predicate* is a set of proper states. For example, if $\mathcal{H}$ is the set $\{0, 1\}$ then the state predicate $\{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$ contains all the states, and only those states, where $x = y$. The set of all state predicates is $E_{\mathcal{H}}(V) =_{\text{DF}} \wp(\mathcal{H}^V)$. Note that there is no state predicate which contains the state $\perp$. This state does not satisfy *any* predicate: not even the predicate **true** (which is satisfied by every state other than $\perp$).

The *domain* of a state, or other function, is denoted $\mathsf{Dom}(s)$. The state which is a subset of $s$ with the domain restricted to the variables in $X$ is denoted $s \downarrow X$. For example, if $s$ is the state $\{x \mapsto 0, y \mapsto 1\}$ then $s \downarrow \{x\}$ is the state $\{x \mapsto 0\}$.

The *overriding* operator on states $s_1 \circledast s_2$ returns a state whose domain is $\mathsf{Dom}(s_1) \cup \mathsf{Dom}(s_2)$. For each variable $x$ in the domain:

$$(s_1 \circledast s_2)(x) \ =_{\text{DF}} \begin{cases} s_2(x) & \text{if } x \text{ is in the domain of } s_2 \\ s_1(x) & \text{otherwise} \end{cases}$$

For example the state $s \circledast \{x \mapsto e\}$ is $s$ with $x$ added to the domain (if necessary) and the value of $x$ set to $e$. So we have:

$$s \circledast \{x \mapsto e\} = (s \setminus \{x \mapsto s(x)\}) \cup \{x \mapsto e\}$$

More generally, we have the following identity:

$$s_1 \circledast s_2 = (s_1 \downarrow (\mathsf{Dom}(s_1) \setminus \mathsf{Dom}(s_2))) \cup s_2$$

## 4.4 Interpretations

The formulae and expressions in WSL are taken from a first order logic language $\mathcal{L}$. If we fix on a particular set of values, and choose an interpretation of the symbols of the base logic $\mathcal{L}$ in terms of the set of values, then we can interpret formulae in $\mathcal{L}$ as state predicates and statements of WSL as state transformations. To be precise:

***Definition 4.1.*** A *structure* $M$ for $\mathcal{L}$ is a set $\mathcal{H}$ of values together with functions that map the constant symbols, function symbols and relation symbols of $\mathcal{L}$ to elements, functions and relations on $\mathcal{H}$. Given a structure $M$ for $\mathcal{L}$ and a set $V$ of variables, we can define an *interpretation* of each formula $\mathbf{B}$ as a state predicate $\mathrm{int}_M(\mathbf{B}, V)$, consisting of the states which satisfy the formula. We can also consider the interpretation $\mathrm{int}_M(e, V)$ of each expression $e$ as a function which maps a state to a value. For example:

$$\mathrm{int}_M(\textbf{true}, V) = \mathcal{H}^V$$
$$\mathrm{int}_M(\textbf{false}, V) = \varnothing$$

For each $s \in \mathcal{H}^V$:

$$\mathrm{int}_M(x + y, V)(s) = s(x) + s(y)$$

For example, if $\mathcal{H} = \{0, 1\}$ and $V = \{x, y\}$, then the state predicate $\mathrm{int}_M(x = y, V)$ is the set of states in which the value given to $x$ equals the value given to $y$, ie:

$$\mathrm{int}_M(x = y, V) = \{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$$

***Definition 4.2.*** Given a set of *sentences* (formulae with no free variables), $\Delta$, a structure $M$ is a *model* for $\Delta$ if and only if $\mathrm{int}_M(\mathbf{Q}, V) = \mathcal{H}^V$ for every $\mathbf{Q} \in \Delta$.

### 4.5 Operational Semantics

A *labelled state* is a pair $\langle L, s\rangle$ consisting of a label $L$ and a proper state $s$. For any labelled state: $\langle L, s\rangle[1] = L$ and $\langle L, s\rangle[2] = s$. The labels are taken from a set of labels, which is denoted $\boldsymbol{L}$.

A *labelled program* is a WSL program in which some primitive statements have labels. For example:
**if** $y = 3$ **then** $L_1 : x := 3;\ L_2 : y := 4$ **else** $L_3$ : **skip fi**

The labels are used to refer to components of a program, not as targets of **goto** statements, so there is usually no need for labels to be unique. A set of primitive statements in a program can be indicated either by giving them all the same label, or by referring to a set of labels. The special label $L_0$ is assumed to be the label of any unlabelled primitive statement.

A *state history*, or just *history* for short, is either a finite sequence of labelled proper states, optionally ending in $\perp$ (without a label), or an infinite sequence of labelled proper states. So a state history will take one of the three forms:

$$\begin{aligned}&\langle\langle L_1, s_1\rangle, \langle L_2, s_2\rangle, \dots, \langle L_n, s_n\rangle\rangle \quad \text{or}\\ &\langle\langle L_1, s_1\rangle, \langle L_2, s_2\rangle, \dots, \langle L_n, s_n\rangle, \perp\rangle \quad \text{or}\\ &\langle\langle L_1, s_1\rangle, \langle L_2, s_2\rangle, \dots\rangle\end{aligned}$$

Note that $\perp$ can only appear as the *last* element in the history, and does not have a label.

A *terminating history* is any finite history which ends in a labelled proper state. Otherwise, the history is *non-terminating* (in which case it is either infinite or ends in $\perp$).

The *length* of a history $\ell(h)$ is the number of elements in the sequence. If $h$ is infinite then $\ell(h) = \omega$.

The $n$th element of history $h$ is denoted $h[n]$. If this is a labelled state, then $h[n][1]$ is the label and $h[n][2]$ is the state. The notation $h[n \mathinner{..} m]$ denotes the subsequence of $h$ from elements $n$ to $m$ inclusive. $h[n \mathinner{..} m] = \langle\rangle$ if $m < n$. The concatenation of two histories is denoted $h_1 \mathbin{+\!\!+} h_2$ and defined as follows:

$$\begin{aligned} h_1 \mathbin{+\!\!+} h_2 &=_{\text{DF}} h_1 \quad \text{if } h_1 \text{ is non-terminating}\\ &=_{\text{DF}} \langle h_1[1], \dots, h_1[\ell(h_1)], h_2[1], \dots\rangle \quad \text{otherwise}\end{aligned}$$

Note that if $h_1$ ends in $\perp$ then it is non-terminating (even though it is finite), so we still have $h_1 \mathbin{+\!\!+} h_2 = h_1$ in this case.

Note also that if $h_1$ is terminating and $h_2$ is infinite, then $h_1 \mathbin{+\!\!+} h_2$ is infinite. Therefore, $h_1 \mathbin{+\!\!+} h_2$ is only terminating when *both* $h_1$ and $h_2$ are terminating.

The *final state* of history $h$, $\mathsf{Final}(h)$ is defined as:

$$\mathsf{Final}(h) =_{\text{DF}} \begin{cases} h[\ell(h)][2] & \text{if } h \text{ is terminating}\\ \perp & \text{otherwise}\end{cases}$$

From the definitions, we see that, if $h_2$ is non-empty, then:

$$\mathsf{Final}(h_1 \mathbin{+\!\!+} h_2) = \begin{cases} \perp & \text{if } h_1 \text{ is non-terminating}\\ \mathsf{Final}(h_2) & \text{otherwise}\end{cases}$$

For every finite integer $n \geqslant 0$, the $n$th *truncation* of a history $h$, denoted $h \uparrow n$, is defined as follows:

$$h \uparrow n =_{\text{DF}} \begin{cases} h[1 \mathinner{..} n] \mathbin{+\!\!+} \langle\perp\rangle & \text{if } n < \ell(h)\\ h & \text{otherwise}\end{cases}$$

A truncation of a history is analogous to an observation of a program as it executes for a certain number of steps (or a certain number of interactions with the environment). A $\perp$ at the end of the observation (or truncation) indicates that the program was still executing at the end of the observation period.

If histories $h_1$ and $h_2$ are such that there exists $n$ such that $h_1 = h_2 \uparrow n$, then we say that $h_1$ is a truncation of $h_2$, denoted $h_1 \preccurlyeq h_2$. If $h_1$ is a truncation of $h_2$ and not equal to $h_2$, then $h_1$ is a *proper truncation* of $h_2$, denoted $h_1 \prec h_2$. Every finite history is a truncation of itself, and $\langle\perp\rangle$ is a truncation (the zeroth truncation) of every non-empty history. In fact, $\langle\perp\rangle$ is a proper truncation of every history other than $\langle\rangle$ and $\langle\perp\rangle$.

An *extension* of a history $h$ is any history $h'$ such that $h \preccurlyeq h'$. If $h \prec h'$ then $h'$ is a *proper extension* of $h$. A terminating or infinite history only has itself as an extension, and has no proper extensions. Only a finite non-terminating history (i.e. one ending in $\perp$) can have a proper extension. Recall that $\perp$ can only appear at the end of a history, so an extension of a history is formed by removing the final $\perp$ and appending another history.

Any structure can be used to define an operational semantics for WSL. The semantic function maps each WSL statement to a function which maps each state to a set of histories. To simplify the definitions, we define these functions in such a way that any history set which contains a finite non-terminating history (i.e. a history ending in $\perp$) should also include every extension of this history. This also allows us to define $\sqcap$ as a simple union, while satisfying ($\mathbf{S} \sqcap$ **abort**) $\approx$ **abort** and allows us to define the semantics of **while** as a simple intersection. Refinement is also a simple subset relation with this definition.

Let $D^*_{\mathcal{H}}(V)$ be the set of all histories on $V$ and $\mathcal{H}$. (This includes both finite and infinite histories) We can extend the concatenation operator $\mathbin{+\!\!+}$ to sets of histories in the obvious way. If $h$ is a history and $H$ a history set then:

$$h \mathbin{+\!\!+} H =_{\text{DF}} \{\, h \mathbin{+\!\!+} h_1 \mid h_1 \in H \,\}$$

For $\perp$ we define: $\mathrm{Int}^{\mathrm{op}}_M(\mathbf{S}, V)(\perp) = D^*_{\mathcal{H}}(V)$ for every statement $\mathbf{S}$.

For each proper initial state $s$ we define:

$$\begin{aligned}&\mathrm{Int}^{\mathrm{op}}_M(L\colon \{\mathbf{B}\}, V)(s)\\ &\qquad =_{\text{DF}} \begin{cases}\{\langle\langle L, s\rangle\rangle\} & \text{if } s \in \mathrm{int}_M(\mathbf{B}, V)\\ D^*_{\mathcal{H}}(V) & \text{otherwise}\end{cases}\\ &\mathrm{Int}^{\mathrm{op}}_M(L\colon x := e, V)(s)\\ &\qquad =_{\text{DF}} \{\langle\langle L, s \circledast \{x \mapsto \mathrm{int}_M(e, V)(s)\}\rangle\rangle\}\\ &\mathrm{Int}^{\mathrm{op}}_M((\mathbf{S}_1 \sqcap \mathbf{S}_2), V)(s)\\ &\qquad =_{\text{DF}} \mathrm{Int}^{\mathrm{op}}_M(\mathbf{S}_1, V)(s) \cup \mathrm{Int}^{\mathrm{op}}_M(\mathbf{S}_2, V)(s)\\ &\mathrm{Int}^{\mathrm{op}}_M(\mathbf{S}_1;\ \mathbf{S}_2, V)(s)\end{aligned}$$

$$=_{\text{DF}} \bigcup\{h \mathbin{+\!\!+} \text{Int}^{\text{op}}_M(\mathbf{S}_2, V)(\mathsf{Final}(h)) \mid h \in \text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s)\}$$

$$\text{Int}^{\text{op}}_M(\textbf{if B then S}_1 \textbf{ else S}_2 \textbf{ fi}, V)(s)$$

$$=_{\text{DF}} \begin{cases} \text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s) & \text{if } s \in \text{int}_M(\mathbf{B}, V) \\ \text{Int}^{\text{op}}_M(\mathbf{S}_2, V)(s) & \text{otherwise} \end{cases}$$

$$\text{Int}^{\text{op}}_M(\textbf{while B do S od}, V)(s)$$

$$=_{\text{DF}} \bigcap_{n<\omega} \text{Int}^{\text{op}}_M(\textbf{while B do S od}^n, V)(s)$$

where the *nth approximation* of the **while** loop, denoted **while B do S od**$^n$, is defined inductively as follows:

$$\begin{aligned} \textbf{while B do S od}^0 &=_{\text{DF}} \textbf{abort} \\ \textbf{while B do S od}^{n+1} &=_{\text{DF}} \textbf{if B then S; while B do S od}^n \textbf{ fi} \end{aligned}$$

The semantics for the sequence $\mathbf{S}_1; \mathbf{S}_2$ is simply the set of histories of the form $h_1 \mathbin{+\!\!+} h_2$ where $h_1$ is in the semantics for $\mathbf{S}_1$ and $h_2$ is in the semantics for $\mathbf{S}_2$ when the initial state for $\mathbf{S}_2$ is the final state in $h_1$. Recall that if $h_1$ is non-terminating then $h_1 \mathbin{+\!\!+} h_2 = h_1$.

Since **skip** $= \{$**true**$\}$ and **abort** $= \{$**false**$\}$ we have:

$$\begin{aligned} \text{Int}^{\text{op}}_M(L\colon \textbf{skip}, V)(s) &=_{\text{DF}} \{\langle\langle L, s\rangle\rangle\} \\ \text{Int}^{\text{op}}_M(L\colon \textbf{abort}, V)(s) &=_{\text{DF}} D^*_{\mathcal{H}}(V) \end{aligned}$$

For any history set $H$, define

$$\mathsf{Finals}(H) =_{\text{DF}} \{\, \mathsf{Final}(h) \mid h \in H \,\}$$

to be the set of final states. Clearly:

$$\mathsf{Finals}(D^*_{\mathcal{H}}(V)) = D_{\mathcal{H}}(V)$$

Although we want to include every extension of a non-terminating finite history in the history set, it is sometimes useful to distinguish between histories which are included because of this rule, and other histories. These "other histories" are those which are not a proper extension of some shorter history in the set. We call these the *minimal* elements:

***Definition 4.3.*** The *minimal* elements of a history set are those which are not proper extensions of some other element:

$$\mathsf{Mins}(H) =_{\text{DF}} \{\, h \in H \mid \neg\exists h' \in H.\, h' \prec h \,\}$$

A history set is *normal* if it is the union of the extensions of all of its minimal elements:

***Definition 4.4.*** A *normal* history set is any set $H$ of histories such that:

$$H = \bigcup\{\, \mathsf{Extensions}(h) \mid h \in \mathsf{Mins}(H) \,\}$$

where:

$$\mathsf{Extensions}(h) =_{\text{DF}} \begin{cases} \{h\} & \text{if } h \text{ is terminating or infinite} \\ h' \mathbin{+\!\!+} D^*_{\mathcal{H}}(V) & \text{otherwise} \end{cases}$$

and $h = h' \mathbin{+\!\!+} \langle\perp\rangle$

So a normal history set may be derived from any of its subsets containing all the minimal elements: simply by adding all the possible extensions (if any) of the minimal elements to the set.

The next theorem shows that history sets which result from the operational semantics of a potentially nondeterministic program are all normal:

***Theorem 4.5.*** For any statement $\mathbf{S}$ and initial state $s$, the history set $H = \text{Int}^{\text{op}}_M(\mathbf{S}, V)(s)$ is the union of the extensions of all minimal elements.

**Proof:** The proof is by induction on the structure of $\mathbf{S}$. For a primitive statement the semantics is either a single terminating history or the set $D^*_{\mathcal{H}}(V)$, both of which are normal.

For the compound statements, the history set is either equal to, or a union or intersection of history sets for smaller statements (or statements with a lower depth of iteration nesting). The intersection or union of a set of normal sets is also normal. ■

### 4.5.1 Semi-Refinement on History Sets

Semi-refinement on normal history sets is defined in terms of the minimal elements of the sets. Intuitively, a semi-refinement is constructed by taking the set of minimal elements and replacing some of the elements by an arbitrary, non-empty set of extensions, to produce a new set of elements from which a new normal history set is constructed. See Section 6.1 for some examples which show why this formal definition is needed.

***Definition 4.6.*** The definition of semi-refinement on normal history sets is:

$$\begin{aligned} &H_1 \preccurlyeq H_2 \quad \text{iff} \\ &\quad \forall h_1 \in \mathsf{Mins}(H_1).\, \exists h_2 \in \mathsf{Mins}(H_2).\, (h_1 \preccurlyeq h_2) \ \wedge \\ &\qquad \forall h_2 \in \mathsf{Mins}(H_2).\, \exists h_1 \in \mathsf{Mins}(H_1).\, (h_1 \preccurlyeq h_2) \end{aligned}$$

An equivalent but slightly simpler definition is:

***Definition 4.7.***

$$\begin{aligned} &H_1 \preccurlyeq' H_2 \quad \text{iff} \\ &\qquad H_2 \subseteq H_1 \ \wedge \\ &\quad \forall h_1 \in \mathsf{Mins}(H_1).\, \exists h_2 \in \mathsf{Mins}(H_2).\, (h_1 \preccurlyeq h_2) \end{aligned}$$

***Theorem 4.8.*** The two definitions given above are equivalent.

**Proof:** Assume that $H_1 \preccurlyeq H_2$ according to Definition 4.6 and let $h_2$ be any element of $H_2$. Since $H_2$ is the union of the extensions of its minimal elements, there must exist $h_2' \in \mathsf{Mins}(H_2)$ such that $h_2 \in \mathsf{Extensions}(h_2')$, i.e. $h_2' \preccurlyeq h_2$. Now, by Definition 4.6, there exists $h_1 \in \mathsf{Mins}(H_1)$ such that $h_1 \preccurlyeq h_2'$, therefore by transitivity, $h_1 \preccurlyeq h_2$. Since $H_1$ is the union of the extensions of its minimal elements, we must have $h_2 \in H_1$ as required. This is true for every $h_2 \in H_2$, so $H_2 \subseteq H_1$.

So Definition 4.6 implies Definition 4.7

Conversely, let $H_1 \preccurlyeq' H_2$ according to Definition 4.7 and let $h_2$ be any element of $\mathsf{Mins}(H_2)$. We need to find an element $h_1 \in \mathsf{Mins}(H_1)$ such that $h_1 \preccurlyeq h_2$. Since $H_2 \subseteq H_1$ we have $h_2 \in H_1$. By definition, $H_1$ is the union of the set of extensions of its minimal elements, so there must be $h_1 \in \mathsf{Mins}(H_1)$ such that $h_2 \in \mathsf{Extensions}(h_1)$, in which case $h_1 \preccurlyeq h_2$ as required.

So Definition 4.7 implies Definition 4.6. ■

Semi-refinement satisfies these properties:

1) Any terminating or infinite history in the original set must be included in the semi-refinement;
2) Any finite non-terminating history (i.e. one which ends in $\perp$) in the original set must either be included or replaced by one or more extensions of it in the semi-refinement;
3) All the histories in the semi-refinement must be derived from one of the previous two rules.

In [20] we defined a semi-refinement relation on the denotational semantics of WSL as follows: $\mathbf{S}_1 \preccurlyeq \mathbf{S}_2$ if and only if:

1) Whenever $\mathbf{S}_1$ terminates on some initial state, $\mathbf{S}_2$ is equivalent to $\mathbf{S}_1$ on that initial state;
2) Otherwise, when $\mathbf{S}_1$ does not terminate on some state, $\mathbf{S}_2$ can do anything on that state.

For the operational semantics, the semi-refinement has to preserve the behaviour of the original program up to the point where it aborts, after which the semi-refinement can do anything.

We illustrate the concept with some simple examples:

The non-terminating program $L_1 \colon x := 1;$ **abort** has operational semantics

$$\{\langle\langle L_1, \{x \mapsto 1\}\rangle, \perp\rangle\}$$

A semi-refinement of this set is:

$$\{\langle\langle L_1, \{x \mapsto 1\}\rangle, \langle L_2, \{x \mapsto 2\}\rangle, \perp\rangle\}$$

which is the semantics of: $L_1 \colon x := 1;\ L_2 \colon x := 2;$ **abort**, which is also non-terminating. Another semi-refinement is:

$$\{\langle\langle L_1, \{x \mapsto 1\}\rangle, \langle L_2, \{x \mapsto 2\}\rangle\rangle\}$$

which is the semantics of the terminating program $L_1 \colon x := 1;\ L_2 \colon x := 2$.

Another example is:

**if** $y = 0$ **then** $x := 42$ **else abort fi**

Which can be semi-refined to: $x := 42$.

These semi-refinements are also valid semi-refinements in the denotational semantic relation.

*Theorem 4.9.* If $H_1 \preccurlyeq H_2$ then for any history $h$, we have $h +\!\!+ H_1 \preccurlyeq h +\!\!+ H_2$.

**Proof:** If $h$ is non-terminating then $h +\!\!+ H_1 = \{h\}$ and the result is trivial. So suppose that $h$ is terminating. By Definition 4.7, $H_2 \subseteq H_1$, so: $h +\!\!+ H_2 \subseteq h +\!\!+ H_1$. Let $h_1$ be any minimal element of $h +\!\!+ H_1$. We need to find a minimal element of $h +\!\!+ H_2$ which is an extension of $h_1$. Now, $h_1$ must be of the form $h +\!\!+ h_1'$ where $h_1' \in H_1$. Also, $h_1'$ must be a *minimal* element of $H_1$ since otherwise there exists $h' \in H_1$ such that $h' \prec h_1'$ and then $h +\!\!+ h'$ contradicts the minimality of $h_1$ in $h +\!\!+ H_1$. Now, since $H_1 \preccurlyeq H_2$ there must exist a $h_2' \in H_2$ such that $h_1' \preccurlyeq h_2'$. Let $h_2 = h +\!\!+ h_2'$, then $h_2 \in h +\!\!+ H_2$ satisfies $h_1 \preccurlyeq h_2$. This is true for every minimal element $h_1$ in $h +\!\!+ H_1$, so by Definition 4.7: $h +\!\!+ H_1 \preccurlyeq h +\!\!+ H_2$ as required. ■

Semi-refinement is defined for operational semantics on nondeterministic programs as follows:

***Definition 4.10.***
Operational nondeterministic semi-refinement:
If $\Delta$ is a given set of sentences and $\mathbf{S}_1$ and $\mathbf{S}_2$ are such that for every model $M$ of $\Delta$ and every state $s \in D_{\mathcal{H}}(V)$:

$$\mathrm{Int}_M^{\mathrm{op}}(\mathbf{S}_1, V)(s) \preccurlyeq \mathrm{Int}_M^{\mathrm{op}}(\mathbf{S}_2, V)(s)$$

then we say that $\mathbf{S}_2$ is a semi-refinement of $\mathbf{S}_1$, or $\mathbf{S}_1$ is semi-refined by $\mathbf{S}_2$, and write:

$$\Delta \models \mathbf{S}_1 \preccurlyeq \mathbf{S}_2$$

The semantics of a **while** loop is defined in terms of the semantics of all of the loop's *approximations*. An approximation of a loop is the result of executing up to $n$ iterations of the loop. If the program is still running after $n$ iterations, then the approximation will **abort**. The idea is to define the meaning of the full loop by collecting together the information from all the approximations.

*Theorem 4.11.* Let $\mathbf{S}$ be any statement and $\mathbf{B}$ be any formula. Let $\mathsf{DO}^n =$ **while B do S od**$^n$ and $\boldsymbol{f}_n = \mathrm{Int}_M^{\mathrm{op}}(\mathsf{DO}^n, V)$. Then, for all $s$ and $n < \omega$: $\boldsymbol{f}_n(s) \preccurlyeq \boldsymbol{f}_{n+1}(s)$.

**Proof:** The base case is trivial since

$$\boldsymbol{f}_0(s) = \mathrm{Int}_M^{\mathrm{op}}(\textbf{abort}, V) = D_{\mathcal{H}}^*(V)$$

for all $s$. The only minimal element in $D_{\mathcal{H}}^*(V)$ is $\langle\perp\rangle$, so for every history set $H \subseteq D_{\mathcal{H}}^*(V)$ we have $D_{\mathcal{H}}^*(V) \preccurlyeq H$.

Suppose that the result holds for $n$ and for all $s$. We claim that $\boldsymbol{f}_{n+1}(s) \preccurlyeq \boldsymbol{f}_{n+2}(s)$.

If $s \notin \mathrm{int}_M(\mathbf{B}, V)$ then $\boldsymbol{f}_0(s) = D_{\mathcal{H}}^*(V)$ and $\boldsymbol{f}_n(s) = \{\langle\langle L_0, s\rangle\rangle\}$ for all $n > 0$ and the result holds.

Conversely, if $s \in \mathrm{int}_M(\mathbf{B}, V)$ then:

$$\begin{aligned} &\boldsymbol{f}_{n+1}(s) \\ &\quad= \mathrm{Int}_M^{\mathrm{op}}(\textbf{if B then S};\ \mathsf{DO}^n \textbf{ else skip fi}, V)(s) \\ &\quad= \mathrm{Int}_M^{\mathrm{op}}(\mathbf{S};\ \mathsf{DO}^n, V)(s) \\ &\quad= \bigcup\{\, h +\!\!+ \boldsymbol{f}_n(\mathsf{Final}(h)) \mid h \in \mathrm{Int}_M^{\mathrm{op}}(\mathbf{S}, V)(s)))\,\} \end{aligned}$$

By the induction hypothesis and Theorem 4.9:

$$\begin{aligned} &\boldsymbol{f}_{n+1}(s) \\ &\quad\preccurlyeq \bigcup\{\, h +\!\!+ \boldsymbol{f}_{n+1}(\mathsf{Final}(h)) \mid h \in \mathrm{Int}_M^{\mathrm{op}}(\mathbf{S}, V)(s)))\,\} \\ &\quad\preccurlyeq \mathrm{Int}_M^{\mathrm{op}}(\textbf{if B then S};\ \mathsf{DO}^{n+1} \textbf{ else skip fi}, V)(s) \end{aligned}$$

since $s \in \mathrm{int}_M(\mathbf{B}, V)$

$$\preccurlyeq \boldsymbol{f}_{n+2}(s)$$

Which proves the result. ■

### 4.5.2 Projection

A program slice is usually defined in terms of a program point and a variable, or set of variables. If the original program terminates on a particular initial state then the slice also terminates and produces the same value(s) for the variable(s) at the given point [13,36].

We generalise the program point to a *set* of labels, with a (potentially different) set of variables of interest at each label of interest. The formal definition of a slicing criterion is therefore a function which maps a label to a set of variables.

***Definition 4.12.*** A *Slicing Criterion* $C$ is a partial function $C$ which maps from labels to sets of variables. For each label $L$ in the domain of $C$, all statements with label $L$ are "points of interest" and $C(L)$ is a set of variables: the "variables of interest" at this particular "point of interest". Note that each different point of interest may have a different set of variables of interest.

Recall that if $s$ is a state and $X$ a set of variables, then $s \downarrow X$, is the state: $\{x_1 \mapsto s(x_1), \ldots, x_n \mapsto s(x_n)\}$ where $\{x_1, \ldots, x_n\}$ is the set $X \cap \mathsf{Dom}(s)$. Each variable in $s \downarrow X$ has the same value as in $s$.

To define projection on histories, we first define projection on a singleton history (a terminating history containing a single labelled state).

***Definition 4.13.*** If $L$ is a label, $s$ a state and $C$ a slicing criterion, then the *projection* of $\langle\langle L, s\rangle\rangle$ on C is defined:

$$\langle\langle L, s\rangle\rangle \downarrow C \;=_{\text{DF}}\; \begin{cases} \langle\rangle & \text{if } L \notin \mathsf{Dom}(C) \\ \langle\langle L, s \downarrow C(L)\rangle\rangle & \text{otherwise} \end{cases}$$

We also define: $\langle\bot\rangle \downarrow C \;=_{\text{DF}}\; \langle\bot\rangle$ for every $C$. This will ensure that the projection of a finite non-terminating history is also non-terminating.

Given a history $h$ and slicing criterion $C$, define the function $\mathsf{Proj}(h, C)$ as follows:

$$\mathsf{Proj}(h, C) \;=_{\text{DF}}\; \langle h[1]\rangle \downarrow C +\!\!+ \langle h[2]\rangle \downarrow C +\!\!+ \ldots$$

Note that if $h$ is infinite, then the infinite concatenation could still return a finite result. In this case, we must have some $N$ such that $\langle h[n]\rangle \downarrow C = \langle\rangle$ for all $n \geqslant N$.

If the original history was infinite, then the program it represents was non-terminating. If $\mathsf{Proj}(h, C)$ is finite, then it represents a terminating computation. We want to ensure that the projection is non-terminating whenever the original history was non-terminating. To do this we append the state $\bot$ to represent the infinite sequence of "elided" labelled states.

Therefore, we define the projection of a history on a slicing criterion as follows:

$$h \downarrow C \;=_{\text{DF}}\; \begin{cases} \mathsf{Proj}(h, C) +\!\!+ \langle\bot\rangle & \text{if } h \text{ is infinite and} \\ & \mathsf{Proj}(h, C) \text{ is finite} \\ \mathsf{Proj}(h, C) & \text{otherwise} \end{cases}$$

This ensures that the projection of a non-terminating history is always non-terminating. Any projection of a terminating history will necessarily be terminating.

The projection relation is extended to operational state transformations in the obvious way. For any operational state transformation $f$ and slicing criterion $C$, define:

$$(f \downarrow C)(s) \;=_{\text{DF}}\; f(s) \downarrow C$$

We use the concept of projection to give a formal definition for operational slicing:

***Definition 4.14.*** An *Operational Syntactic Slice* of $\mathbf{S}$ on a slicing criterion $C$ is any program $\mathbf{S}'$ such that:

$$\mathbf{S}' \sqsubseteq \mathbf{S} \quad \text{and} \quad \Delta \mathrel{|}\!\models \mathbf{S} \downarrow C \;\preccurlyeq\; \mathbf{S}' \downarrow C$$

***Definition 4.15.*** An *Operational Semantic Slice* of $\mathbf{S}$ on a slicing criterion $C$ is any program $\mathbf{S}'$ such that:

$$\Delta \mathrel{|}\!\models \mathbf{S} \downarrow C \;\preccurlyeq\; \mathbf{S}' \downarrow C$$

To analyse the behaviour of an interactive program, we can do the following:

1) Label all statements which can interact with the environment;
2) For each label, $L$ determine the set of variables whose values are examined or updated by the interaction;
3) Define a slicing criterion $C$ such that, for each label $L$, $C(L)$ is the set of variables identified in step 2;
4) Slice the program on slicing criterion $C$.

In the next section we introduce a denotational semantics for the language and define a slicing relation on in terms of the semantics. In Section 5 we prove that these two formalisations of slicing are equivalent. This gives us confidence that the formalisation of the informal concept of a "program slice" is correct.

## 4.6 Denotational Semantics

The denotational semantics defines the semantics of a program as a function which maps each initial state to the *set* of possible final states. If the set of final states includes $\bot$, then the program may choose not to terminate. In this case, we don't care what else it may choose to do: so if the set of final states includes $\bot$ then it is defined to include every other state in $D_{\mathcal{H}}(V)$ as well. This definition means that we can define refinement as a simple subset relation on the denotational semantics. If $f_1$ is the semantic function for statement $\mathbf{S}_1$ and $f_2$ is the semantic function for $\mathbf{S}_2$, then $f_2$ is a refinement of $f_1$ if and only if $\forall s.\, f_2(s) \subseteq f_1(s)$. Therefore, we call this property *subset refinement*.

The rationale for including every state when the set of final states includes $\bot$ comes from the following chain of reasoning:

1) A *specification* defines a set of initial states (the states for which the program is defined) and for each of these initial state it defines the set of allowed final states. These are all proper states, so the specification for a non-terminating program defines an empty set of states for which the program is defined;

2) For a program to satisfy the specification, the program's set of final states must be a subset of the allowed final states, for each of the initial states for which the program is required to terminate.
3) A *refinement* of a program is defined as any program which satisfies all the specifications satisfied by the original program;
4) If a program's set of final states includes $\perp$ for a particular initial state, then it cannot satisfy any specifications for that initial state. So *any* program is a refinement of any program which *may* abort (for that initial state);
5) So, any program which may abort on an initial state is equivalent to any other program which may abort on that state, in the sense that each program is a refinement of the other;
6) If we ensure that every other state is included in the set of final states whenever $\perp$ is included, then refinement becomes a simple subset relation and two programs are semantically equivalent precisely when they have identical semantics. Another advantage is that the semantics of the **while** is very simple: the set of final states for the whole loop is simply the intersection of the sets of final states for all the finite approximations.

These considerations lead to the following definition of the semantic function for nondeterministic programs.

For initial state $\perp$ we define: $\mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}, V)(\perp) = D_{\mathcal{H}}(V)$ for every statement **S**. Otherwise:

$$\mathrm{Int}^{\mathrm{den}}_M(L\colon \{\mathbf{B}\}, V)(s) =_{\mathrm{DF}} \begin{cases} \{s\} & \text{if } s \in \mathrm{int}_M(\mathbf{B}, V) \\ D_{\mathcal{H}}(V) & \text{otherwise} \end{cases}$$

$$\mathrm{Int}^{\mathrm{den}}_M(L\colon x := e, V)(s) =_{\mathrm{DF}} \{s \circledast \{x \mapsto \mathrm{int}_M(e, V)(s)\}\}$$

$$\mathrm{Int}^{\mathrm{den}}_M((\mathbf{S}_1 \sqcap \mathbf{S}_2), V)(s) =_{\mathrm{DF}} \mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}_1, V)(s) \cup \mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}_2, V)(s)$$

$$\mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}_1;\ \mathbf{S}_2, V)(s) =_{\mathrm{DF}} \bigcup \Big\{ \mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}_2, V)(t) \Bigm| t \in \mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}_1, V)(s) \Big\}$$

$$\mathrm{Int}^{\mathrm{den}}_M(\textbf{if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}, V)(s) =_{\mathrm{DF}} \begin{cases} \mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}_1, V)(s) & \text{if } s \in \mathrm{int}_M(\mathbf{B}, V) \\ \mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}_2, V)(s) & \text{otherwise} \end{cases}$$

$$\mathrm{int}^{\mathrm{den}}_M(\textbf{while B do S od}, V)(s) =_{\mathrm{DF}} \bigcap_{n<\omega} \mathrm{Int}^{\mathrm{den}}_M(\textbf{while B do S od}^n, V)(s)$$

The semantics for the **while** loop is simply the intersection of the semantics for each finite approximation. The result is the *least defined* statement which is a refinement of *all* the approximations.

An important property of nondeterministic choice is that combining any statement with **abort** will give **abort**, i.e. for any statement **S**, any reasonable semantic equivalence relation should have:

$$(\mathbf{S} \sqcap \textbf{abort}) \approx \textbf{abort}$$

This makes sense because the statement $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$ can only satisfy a specification if both $\mathbf{S}_1$ and $\mathbf{S}_2$ satisfy the specification. **abort** can only satisfy the totally undefined specification (which is **abort** itself), so (**S** $\sqcap$ **abort**) also only satisfies the specification **abort**. So (**S** $\sqcap$ **abort**) must be equivalent to **abort**.

The semi-refinement relation is defined on sets of states as follows. For any sets of states $S_1, S_2$:

$$S_1 \preccurlyeq S_2 \quad \text{iff} \quad \perp \in S_1 \vee S_1 = S_2$$

Semi-refinement is defined for denotational semantics on nondeterministic programs as follows:

***Definition 4.16.***
Denotational nondeterministic semi-refinement:
If $\Delta$ is a give set of sentences and $\mathbf{S}_1$ and $\mathbf{S}_2$ are such that for every model $M$ of $\Delta$ and every state $s \in D_{\mathcal{H}}(V)$:

$$\mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}_1, V)(s) \preccurlyeq \mathrm{Int}^{\mathrm{den}}_M(\mathbf{S}_2, V)(s)$$

then we say that $\mathbf{S}_2$ is a semi-refinement of $\mathbf{S}_1$, or $\mathbf{S}_1$ is semi-refined by $\mathbf{S}_2$, and write:

$$\Delta \models \mathbf{S}_1 \preccurlyeq \mathbf{S}_2$$

### 4.7 Extended Denotational Semantics

The paper "Slicing as a Program Transformation" [20] converts middle slicing to end slicing by adding code to append the current values of the variables of interest to a new variable slice at the appropriate points and then end slicing on slice. This will "capture" the sequence of values taken on by the variables of interest at the points of interest: but the captured values will only reach the end of the program if the program subsequently terminates.

Similarly, it is possible to "capture" all the interactions of the program with its environment: by recording the sequence of values taken on by of the variables of interest in a new variable which will appear in the final state. Again, this approach will not work for programs which (intentionally) do not terminate.

To ensure that the captured data is observable in the final state, we annotate the program in such a way as to enforce immediate termination when a sufficient number of interactions have been encountered. (The requirement which constitutes a "sufficient" number of interactions will be provided in an additional input variable). The "extended denotation semantics" for the original program is simply the standard WSL semantics [20] for the annotated program. The middle slice for a potentially non-terminating program is then defined as an end slice on the variable slice for the annotated program, using the slicing relation defined in [20].

In the previous section we defined an operational semantics and projection operator which give an operational definition of slicing. In the rest of this section we describe how to carry out this slicing using the extended denotational semantics.

Let $C$ be any slicing criterion, and let **S** be any program. We want to "capture" the values of the variables of interest at the labels of interest (as defined by $C$) by appending to a new variable $\mathsf{slice}$. We want to stop execution (enforce immediate termination) as soon as $\mathsf{slice}$ gets sufficiently large. The obvious way to do this is to test the length of $\mathsf{slice}$ and stop as soon as it reaches a certain length. This will work for deterministic programs, but simply testing the length is not sufficiently precise for nondeterministic programs (see Section 6.4). Instead we need to test, for each label $L$, the number of elements in $\mathsf{slice}$ which have that label. We set a maximum value, or "quota" for each label and terminate the program as soon as any label has reached its quota of elements in $\mathsf{slice}$. The variable $\mathsf{limit}$ contains a function from labels to integers which records the maximum number of slice elements allowed for each label. This is called a *limit function*.

Let $q$ be a terminating history and $\boldsymbol{L}$ be the set of all labels and $c : \boldsymbol{L} \to \mathbb{N}$ be a limit function which records the maximum count for each label. The relation $\ll$ is defined:

$$q \ll c \ =_{\text{DF}}\ \forall L \in \boldsymbol{L}.\, \ell(q \downarrow C_L) < c(L)$$

where $C_L$ is the slicing criterion $\{L \mapsto V\}$. This relation tests each label against its quota: as defined by the limit function $c$. The original program is annotated with tests for this condition, arranged to cause the program to terminate immediately as soon as this condition fails. Note that if $c(L) = 0$ for any label, then no history can satisfy the condition and the program terminates immediately. The condition $\exists L \in \boldsymbol{L}.\, c(L) = 0$ is equivalent to the condition $\langle\rangle \not\ll c$.

The value appended to the sequence in $\mathsf{slice}$ is the pair $\langle L, \mathsf{state}(C(L))\rangle$ where $L$ is the current label and $\mathsf{state}(C(L))$ is some suitable WSL representation of the current values of the variables in the set $C(L)$. For example, if $C(L)$ is the set $\{x, y\}$ then we can define $\mathsf{state}(C(L))$ as $\langle\langle\text{"x"}, x\rangle, \langle\text{"y"}, y\rangle\rangle$.

The function $\mathsf{Apply}(C, \mathbf{S})$ takes a slicing criterion $C$ and a labelled program $\mathbf{S}$ and returns a new unlabelled program which is an annotated version of $\mathbf{S}$ in which assignments to $\mathsf{slice}$ have been added and tests of $\mathsf{slice}$ against $\mathsf{limit}$ have been introduced at all program points, and the program terminates as soon as $\mathsf{slice} \not\ll \mathsf{limit}$. The formal definition follows.

If $L\colon \mathbf{S}$ is a labelled primitive statement, then we define:

$$\begin{aligned} &\mathsf{Apply}(C, L\colon \mathbf{S}) \\ &\quad =_{\text{DF}}\ \textbf{if}\ \mathsf{slice} \ll \mathsf{limit}\ \textbf{then}\ \mathbf{S}\ \textbf{fi} \quad \text{if } L \notin \mathsf{Dom}(C) \\ &\quad =_{\text{DF}} \begin{cases} \textbf{if}\ \mathsf{slice} \ll \mathsf{limit} \\ \textbf{then}\ \mathbf{S};\ \mathsf{slice} := \mathsf{slice} \mathbin{+\!\!+} \langle\langle L, \mathsf{state}(C(L))\rangle\rangle\ \textbf{fi} \\ \quad \text{otherwise} \end{cases} \end{aligned}$$

For compound statements we define:

$$\begin{aligned} &\mathsf{Apply}(C, \mathbf{S}_1;\ \mathbf{S}_2) \\ &\qquad =_{\text{DF}}\ \mathsf{Apply}(C, \mathbf{S}_1);\ \mathsf{Apply}(C, \mathbf{S}_2) \\ &\mathsf{Apply}(C, \textbf{if}\ \mathbf{B}\ \textbf{then}\ \mathbf{S}_1\ \textbf{else}\ \mathbf{S}_2\ \textbf{fi}) \\ &\qquad =_{\text{DF}}\ \textbf{if}\ \mathbf{B}\ \textbf{then}\ \mathsf{Apply}(C, \mathbf{S}_1)\ \textbf{else}\ \mathsf{Apply}(C, \mathbf{S}_2)\ \textbf{fi} \\ &\mathsf{Apply}(C, \textbf{while}\ \mathbf{B}\ \textbf{do}\ \mathbf{S}_1\ \textbf{od}) \\ &\qquad =_{\text{DF}}\ \textbf{while}\ \mathsf{slice} \ll \mathsf{limit} \wedge \mathbf{B}\ \textbf{do}\ \mathsf{Apply}(C, \mathbf{S}_1)\ \textbf{od} \end{aligned}$$

This technique of annotating a program to enforce termination under certain conditions has been used for many years to model unbounded loops with **exit** statements and action systems with the special action $Z$: where the statement **call** $Z$ causes immediate termination of the action system (see [9] for example). So the methods for transforming, simplifying and analysing programs annotated in this way are well developed.

We can now define a semantic slice relation on the denotational semantics.

***Definition 4.17.*** Let $V' = V \cup \{\mathsf{slice}, \mathsf{limit}\}$. If for every model $M$ of the countable set of sentences $\Delta$ we have:

$$\mathrm{int}_M^{\mathrm{den}}(\mathbf{S}_1 V') \downarrow \{\mathsf{slice}\} \ \preccurlyeq\ \mathrm{int}_M^{\mathrm{den}}(\mathbf{S}_2, V') \downarrow \{\mathsf{slice}\}$$

then we write:

$$\Delta \models \mathbf{S}_1 \ \overset{\mathsf{slice}}{\preccurlyeq}\ \mathbf{S}_2$$

Recall that for a state $s$ and set of variables $X$, the state $s \downarrow X$ is the subset of $s$ whose domain consists of the variables $\mathsf{Dom}(s) \cap X$.

***Definition 4.18.*** A *Denotational Syntactic Slice* of $\mathbf{S}$ on a slicing criterion $C$ is any program $\mathbf{S}'$ such that:

$$\mathbf{S}' \sqsubseteq \mathbf{S} \quad \text{and} \quad \Delta \models \mathsf{Apply}(C, \mathbf{S}) \ \overset{\mathsf{slice}}{\preccurlyeq}\ \mathsf{Apply}(C, \mathbf{S}'))$$

***Definition 4.19.*** A *Denotational Semantic Slice* of $\mathbf{S}$ on a slicing criterion $C$ is any program $\mathbf{S}'$ such that:

$$\Delta \models \mathsf{Apply}(C, \mathbf{S}) \ \overset{\mathsf{slice}}{\preccurlyeq}\ \mathsf{Apply}(C, \mathbf{S}'))$$

In [20] we give a definition of slicing defined in terms of weakest preconditions and prove that it is equivalent to the denotational semantic definition.

We now have two different definitions of a slice on a slicing criterion $C$. One is defined by an operational semantics and the other is defined by a denotational semantics on a program generated by applying the slicing criterion to the original program.

In the next section we will prove that these two definitions of slicing are equivalent. Therefore: an interactive program, such as an embedded system, can be analysed by slicing on its set of interactions with the environment. This analysis can be carried out using the operational semantics, or equivalently, the denotational semantics.

## 5 Equivalence of Operational and Denotational Semantics for Slicing

In this section we formally prove the equivalence of the operational and denotational semantics for slicing. This allows us to use either method for constructing slices, and for analysing the interactive behaviour of programs, and gives confidence that the informal notion of a "program slice" has been formalised correctly.

To prove equivalence of the two forms of slicing we take the operational semantics, project it using the slicing criterion, and then truncate each history, if necessary, just enough to ensure that it satisfies $h \ll c$. The function $\boldsymbol{f}^c(s)$ returns this set of truncated histories.

The idea is that this set of truncated histories can be computed from the denotational semantics of the annotated program by examining the final value of slice in each final state when executed in an initial state in which slice contains the empty sequence and limit contains the limit function $c$. (Recall that the annotations ensure that the program terminates immediately as soon as slice $\ll$ limit becomes false).

In order to push the proof through for sequential composition, we need to take account of the sequence of states which the surrounding program has executed *before* starting execution of the statement under consideration. This sequence is denoted by $q$ in the proof. Instead of initialising slice to the empty sequence we initialise slice with the value $q$ and initialise limit with the value $c \oplus q$ where, for each label $L$:

$$(c \oplus q)(L) =_{\text{DF}} c(L) + \ell(q \downarrow C_L)$$

The function $\boldsymbol{g}^{c,q}(s)$ is the denotational semantics of the annotated program with these initialisations. Finally, we define $\boldsymbol{h}^{q,c}$ to be $\{q\}$ if $\boldsymbol{g}^{c,q}(s)$ does not terminate and otherwise to be the set of values of slice in all the final states in $\boldsymbol{g}^{c,q}(s)$.

The equation we want to prove is then:

$$q +\!\!+ \boldsymbol{f}^c(s) = \boldsymbol{h}^{c,q}(s) \qquad (4)$$

This equation shows that the projected operational semantics $\boldsymbol{f}^c(s)$ can be extracted from the denotational semantics of the annotated program via $\boldsymbol{h}$. If we define the slicing criterion to include all variables at all labels, then the projection has no effect: so we can extract the entire operational semantics, if required.

Informally, we can think of each execution of the annotated program under the denotational semantics as a "probe" which computes the slice up to a certain limit (given by the limit function $c$ provided in the initial value of variable limit). If the probe execution does not terminate, then the limit function can be tightened until termination is reached. The set of all "probe" executions provides all the information needed to compute the complete slice. Since limit is a new variable, if the annotated version of a proposed slice **S** is a semi-refinement of the annotated version of **P**, then (according to 4) the projection of the operational semantics for **S** will be a semi-refinement of the projection of the operational semantics for **P**. So the two definitions of slicing are equivalent.

First, we prove some preliminary theorems.

***Theorem 5.1.*** For all statements **S** and initial states $s$:

$$\text{Int}^{\text{den}}_M(\mathbf{S}, V)(s) = \mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}, V)(s))$$

**Proof:** The proof is by induction on the structure of **S** and the depth of **while** loop nesting.

**Case (1):** Suppose **S** is the assertion $L\colon \{\mathbf{Q}\}$. If $s \in \text{int}_M(\mathbf{Q}, V)$ then $\text{Int}^{\text{den}}_M(\mathbf{S}, V)(s) = \{s\}$ and $\text{Int}^{\text{op}}_M(\mathbf{S}, V)(s) = \{\langle\langle L, s\rangle\rangle\}$ and the result follows. Conversely, suppose $s \notin \text{int}_M(\mathbf{Q}, V)$, then $\text{Int}^{\text{den}}_M(\mathbf{S}, V)(s) = D_{\mathcal{H}}(V)$ and $\text{Int}^{\text{op}}_M(\mathbf{S}, V)(s) = D^*_{\mathcal{H}}(V)$ and the result follows.

**Case (2):** Suppose **S** is the assignment $L\colon x := e$. Then $\text{Int}^{\text{den}}_M(\mathbf{S}, V)(s) = \{s \circledast \{x \mapsto e\}\}$ and $\text{Int}^{\text{op}}_M(\mathbf{S}, V)(s) = \{\langle\langle L, s \circledast \{x \mapsto e\}\rangle\rangle\}$ and the result follows.

**Case (3):** Suppose $\mathbf{S} = \mathbf{S}_1;\ \mathbf{S}_2$:
If $\perp \in \mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s))$ then $\perp \in \mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_1;\ \mathbf{S}_2, V)(s))$ and:

$$\begin{aligned}&\mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_1;\ \mathbf{S}_2, V)(s))\\ &\qquad = D_{\mathcal{H}}(V) = \text{Int}^{\text{den}}_M(\mathbf{S}_1;\ \mathbf{S}_2, V)(s))\end{aligned}$$

So suppose $\perp \notin \mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s))$, so every $h \in \mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s))$ is terminating:

$$\begin{aligned}&\mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}, V)(s))\\ &= \mathsf{Finals}\Big(\bigcup\{h +\!\!+ \text{Int}^{\text{op}}_M(\mathbf{S}_2, V)(\mathsf{Final}(h))\\ &\qquad\qquad \mid h \in \text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s)\}\Big)\\ &= \bigcup\{\mathsf{Finals}(h +\!\!+ \text{Int}^{\text{op}}_M(\mathbf{S}_2, V)(\mathsf{Final}(h)))\\ &\qquad \mid h \in \text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s)\}\\ &= \bigcup\{\mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_2, V)(\mathsf{Final}(h)))\\ &\qquad \mid h \in \text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s)\}\end{aligned}$$

since $h$ is a terminating history

$$= \bigcup\Big\{\ \text{Int}^{\text{den}}_M(\mathbf{S}_2, V)(\mathsf{Final}(h))\ \Big|\ h \in \text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s)\ \Big\}$$

by the induction hypothesis on $\mathbf{S}_2$

$$\begin{aligned}&= \bigcup\Big\{\ \text{Int}^{\text{den}}_M(\mathbf{S}_2, V)(t)\ \Big|\ t \in \mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s))\ \Big\}\\ &= \bigcup\Big\{\ \text{Int}^{\text{den}}_M(\mathbf{S}_2, V)(t)\ \Big|\ t \in \text{Int}^{\text{den}}_M(\mathbf{S}_1, V)(s))\ \Big\}\end{aligned}$$

by the induction hypothesis on $\mathbf{S}_1$

$$= \text{Int}^{\text{den}}_M(\mathbf{S}_1;\ \mathbf{S}_2, V)(s)$$

**Case (4):** If **S** is **if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi** then consider the cases $s \in \text{int}_M(\mathbf{B}, V)$ and $s \notin \text{int}_M(\mathbf{B}, V)$ and apply the induction hypothesis for $\mathbf{S}_1$ and $\mathbf{S}_2$ respectively.

**Case (5):** If **S** is $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$ then:

$$\begin{aligned}&\mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}, V)(s))\\ &\quad = \mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s) \cup \text{Int}^{\text{op}}_M(\mathbf{S}_2, V)(s))\\ &\quad = \mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_1, V)(s)) \cup \mathsf{Finals}(\text{Int}^{\text{op}}_M(\mathbf{S}_2, V)(s))\\ &\quad = \text{Int}^{\text{den}}_M(\mathbf{S}_1, V)(s) \cup \text{Int}^{\text{den}}_M(\mathbf{S}_2, V)(s)\end{aligned}$$

by the induction hypothesis on $\mathbf{S}_1$ and $\mathbf{S}_2$

$$= \text{Int}^{\text{den}}_M((\mathbf{S}_1 \sqcap \mathbf{S}_2), V)(s)$$

**Case (6):** If **S** is **while B do** $\mathbf{S}_1$ **od** then let $\mathsf{DO}^n =$ **while B do** $\mathbf{S}_1$ $\textbf{od}^n$. Let $\langle H_0, H_1, \dots\rangle$ be an infinite sequence of history sets such that $H_n \preccurlyeq H_{n+1}$ for all $n < \omega$. We claim that:

$$\mathsf{Finals}(\bigcap_{n<\omega} H_n) = \bigcap_{n<\omega} \mathsf{Finals}(H_n)$$

See below (Theorem 5.6) for the proof of this claim.

$$\begin{aligned}&\mathsf{Finals}(\text{Int}^{\text{op}}_M(\textbf{while B do } \mathbf{S}_1 \textbf{ od}, V)(s))\\ &\quad = \mathsf{Finals}(\bigcap_{n<\omega} \text{Int}^{\text{op}}_M(\mathsf{DO}^n, V)(s))\end{aligned}$$

by definition.

$$= \bigcap_{n<\omega} \mathsf{Finals}(\mathrm{Int}_M^{\mathrm{op}}(\mathsf{DO}^n, V)(s))$$

by Theorem 4.11 above and Theorem 5.6 below.

$$= \bigcap_{n<\omega} (\mathrm{Int}_M^{\mathrm{den}}(\mathsf{DO}^n, V)(s))$$

by the induction hypothesis for each $\mathsf{DO}^n$

$$= \mathrm{Int}_M^{\mathrm{den}}(\textbf{while B do S}_1\ \textbf{od}, V)(s))$$

by definition. ■

***Definition 5.2.*** A *terminating* history set is one in which all elements are terminating.

***Lemma 5.3.*** If history set $H_1$ is terminating and $H_1 \preccurlyeq H_2$, then $H_1 = H_2$.

**Proof:** By Definition 4.7 $H_2 \subseteq H_1$, so $H_2$ must also be terminating. Also, any $h_1 \in H_1$ is terminating and therefore minimal, so by Definition 4.7 there exists a minimal element $h_2 \in H_2$ such that $h_1 \preccurlyeq h_2$. Since $h_1$ is terminal, we must have $h_2 = h_1$ and therefore $h_1 \in H_2$. But this is true for all $h_1 \in H_1$. So $H_1 \subseteq H_2$ and therefore, $H_1 = H_2$. ■

***Definition 5.4.*** A *finitely branching* history set is one whose elements form a finitely branching tree. The set of tree nodes is defined as the set of all prefixes (initial segments) of elements in $H$:

$$N = \{\, h[1\,..\,n] \mid h \in H \ \wedge\ n \leqslant \ell(h) \,\}$$

The set of edges is defined as:

$$E = \{\, \langle h[1\,..\,n-1], h\rangle \mid h \in N \ \wedge\ n = \ell(h) > 0 \,\}$$

***Lemma 5.5.*** For any statement $\mathbf{S}$ and initial state $s$ the set $\mathsf{Mins}(\mathrm{Int}_M^{\mathrm{op}}(\mathbf{S}, V)(s))$ is a finitely-branching history set.

**Proof:** The proof is by induction on the structure of $\mathbf{S}$ and the depth of **while** loop nesting. For the assertion, $(\mathrm{Int}_M^{\mathrm{op}}(L\colon \{\mathbf{Q}\}, V)(s))$ is either $\{\langle L, s\rangle\}$, which is trivially finitely-branching, or $D^*_{\mathcal{H}}(V)$. In the latter case, $\mathsf{Mins}(D^*_{\mathcal{H}}(V)) = \{\langle\bot\rangle\}$ which is also finitely branching.

For the **while** loop, $\mathrm{Int}_M^{\mathrm{op}}(\mathbf{S}, V)(s)$ is the intersection of a sequence of history sets. By the induction hypothesis, all of these sets are finitely branching.

The other cases are straightforward. ■

To complete the proof of Theorem 5.1 we need to prove the following:

***Theorem 5.6.*** Let $\langle H_n \mid n < \omega\rangle$ be any infinite sequence of finitely branching history sets such that $H_n \preccurlyeq H_{n+1}$ for all $n$. Then:

$$\mathsf{Finals}(\bigcap_{n<\omega} H_n) = \bigcap_{n<\omega} \mathsf{Finals}(H_n)$$

**Proof:** Suppose that there exists $k$ such that $H_n = H_k$ for all $n \geqslant k$. By the definition of semi-refinement (Definition 4.7), $H_k \subseteq H_n$ for all $n \leqslant k$. So $\bigcap_{n<\omega} H_n = H_k$ and $\bigcap_{n<\omega} \mathsf{Finals}(H_n) = \mathsf{Finals}(H_k)$ and the result follows.

Conversely, suppose for every $k$ there exists $n > k$ such that $H_k \prec H_n$. As in Theorem 4.11, we can construct an infinite sequence of integers $m_n$ such that $H_{m_n} \prec H_{m_{n+1}}$ for all $n$. Clearly:

$$\bigcap_{n<\omega} H_n = \bigcap_{n<\omega} H_{m_n}$$

For every $n$ we have $H_{m_n} \prec H_{m_{n+1}}$, so there must be a finite non-terminating history in every $H_{m_n}$. So $\bot \in \mathsf{Finals}(H_{m_n})$ for every $n$ and so $\bot \in \mathsf{Finals}(H_n)$ and therefore $\mathsf{Finals}(H_n) = D_{\mathcal{H}}(V)$ for every $n$, so $\bigcap_{n<\omega} \mathsf{Finals}(H_n) = D_{\mathcal{H}}(V)$.

We claim that there is an infinite history $h \in \bigcap_{n<\omega} H_n$, which means that

$$\mathsf{Finals}(\bigcap_{n<\omega} H_n) = D_{\mathcal{H}}(V)$$

and the theorem is proved.

To prove the claim we define $\mathsf{pm}(H)$ as the set of all prefixes of minimal elements of any history set $H$:

$$\mathsf{pm}(H) =_{\text{DF}} \{\, h[1\,..\,n] \mid h \in \mathsf{Mins}(H) \ \wedge\ n \leqslant \ell(h) \,\}$$

For every $H_{m_n}$, if $h$ is a minimal element of $H_{m_n}$ then either $h \in H_{m_{n+1}}$ or there is an extension of $h$ in $H_{m_{n+1}}$. Either way, $h$ is in the set of prefixes of minimal elements of $H_{m_{n+1}}$. So by induction on $n$, $h \in \mathsf{pm}(\bigcap_{n<\omega} H_n)$. But this applies to all prefixes of all minimal elements of any $H_n$. So:

$$\bigcup_{n<\omega} \mathsf{pm}(H_n) \subseteq \mathsf{pm}(\bigcap_{n<\omega} H_n)$$

Now consider the tree defined by $\bigcup_{n<\omega} \mathsf{pm}(H_n)$. By Lemma 5.5, this is a finitely-branching tree. Also, for each $m$, $H_{m_n} \prec H_{m_{n+1}}$ implies $\mathsf{Mins}(H_{m_n}) \subset \mathsf{Mins}(H_{m_{n+1}})$, so $\mathsf{pm}(H_{m_n}) \subset \mathsf{pm}(H_{m_{n+1}})$ so the tree is a union of an infinite sequence of trees, each of which is strictly larger than the previous one. So the tree is infinitely large.

By König's Lemma [51], embedded in this infinitely large and finitely branching tree there must be an infinite path $\langle h_1, h_2, \dots, h_n, \dots\rangle$ where each $h_n$ is of length $n$ and is a prefix of an element in $\bigcup_{n<\omega} \mathsf{pm}(H_n)$, so is itself an element of $\bigcup_{n<\omega} \mathsf{pm}(H_n)$. Construct the infinite path $h$ from this sequence in the obvious way: $h = \langle h_1[1], h_2[2], h_3[3], \dots\rangle$. Pick any $H_n$ in the sequence, and let $h[1\,..\,m]$ be the largest prefix of $h$ which is in $H_n$. Now, by definition of $h$ there must be some other element $H_k$ such that $h[1\,..\,m+1]$ is a prefix of an element of $H_k$. Clearly $k > n$, so $H_n \prec H_k$ and so $h[1\,..\,m+1]$ must be a prefix of an extension of an element in $H_n$. The only possible history in $H_n$ which can be extended to $h[1\,..\,m+1]$ is $h[1\,..\,m] \mathbin{+\!\!+} \langle\bot\rangle$, since $h[1\,..\,m]$ is the largest prefix of $h$ which is a prefix of an element of $H_n$. Now, $h[1\,..\,m] \mathbin{+\!\!+} \langle\bot\rangle$ is a minimal element in $H_n$, so every extension of it is in $H_n$. In particular, $h \in H_n$. But this applies to every $n$, so $h \in \bigcap_{n<\omega} H_n$ as required.

This completes the proof. ■

The following counterexample shows that the condition that each $H_n$ is finitely branching is a necessary condition. For integers $j$ and $n$, define the history $h_{jn}$ as follows:

$$h_{j,n} = \begin{cases} \langle\langle L_0, \{x \mapsto j\}\rangle, \langle L_0, \{x \mapsto j-1\}\rangle & \\ \qquad \ldots \langle L_0, \{x \mapsto 0\}\rangle\rangle & \text{if } j < n \\ \langle\langle L_0, \{x \mapsto j\}\rangle, \langle L_0, \{x \mapsto j-1\}\rangle & \\ \qquad \ldots \langle L_0, \{x \mapsto j-n\}\rangle, \perp\rangle & \text{if } j \geqslant n \end{cases}$$

In the first case, there are $j+1$ elements in the history. In the second case there are $n+1$ elements. Now let:

$$H_n = \bigcup_{j<\omega} \mathsf{Extensions}(h_{j,n})$$

Every $H_n$ contains a non-terminating element: the history $h_{n,n}$. So $\perp \in \mathsf{Finals}(H_n)$ for every $n$, and $\perp \in \bigcap_{n<\omega} \mathsf{Finals}(H_n)$. We claim that there is no non-terminating history which appears in every $H_n$. Every history in every $H_n$ starts with an element of the form $\langle L_0, x \mapsto k\rangle$. Suppose that $h$ is a non-terminating history which appears in every $H_n$. The first element of $h$ is of the form $\langle L_0, x \mapsto k\rangle$. But in every $H_{k+1}$, every element which starts $\langle L_0, h \mapsto k\rangle$ is terminating. So $h \notin H_{k+1}$ which is a contradiction. So every element of $\bigcap_{n<\omega} H_n$ terminates, so $\perp \notin \mathsf{Finals}(\bigcap_{n<\omega} H_n)$.

## 5.1 Proof of Equivalence

The main theorem in this section (Theorem 5.15 will show that provided the sets of labels in the two branches of a choice are disjoint (i.e. the same label does not appear in both branches of a nondeterministic choice), then operational slicing is equivalent to denotational slicing on nondeterministic programs.

First we define some notation.

***Definition 5.7.*** For any history $h$ and function $c : \boldsymbol{L} \to \mathbb{N}$, define the *restriction* of $h$ to $c$, denoted $h \Downarrow c$, as follows:

$$h \Downarrow c \ =_{\text{DF}} \begin{cases} h & \text{if } h \ll c \\ \langle\rangle & \text{if } \langle\rangle \not\ll c \\ h' & \text{otherwise} \end{cases}$$

where:

$$h' = h[1\mathinner{..}\max(\{\, n \in 1\mathinner{..}\ell(n) \mid h[1\mathinner{..}n-1] \ll c \,\})]$$

Informally, this does nothing to $h$ if $h \ll c$ and otherwise truncates $h$ to the point where it just fails to satisfy $h \ll c$. This is the point where the annotated program will immediately terminate, without appending anything further to slice.

The restriction operator is extended to sets of histories in the obvious way:

***Definition 5.8.*** For any set $H$ of histories and function $c : \boldsymbol{L} \to \mathbb{N}$, define the set $H \Downarrow c$ as follows:

$$H \Downarrow c \ =_{\text{DF}} \ \{\, h \Downarrow c \mid h \in H \,\}$$

Given a function $c : \boldsymbol{L} \to \mathbb{N}$ and a terminating history $q$ we define a new function $c \oplus q : \boldsymbol{L} \to \mathbb{N}$ which increases the value of $c$ on each label according to the number of elements of $q$ which have that label. Similarly we define $c \ominus q : \boldsymbol{L} \to \mathbb{N}$ which decreases the value of $c$ on each label, provided $c(L) \geqslant \ell(q \downarrow C_L)$ for each $L$. The operators $\oplus$ and $\ominus$ are used to loosen and tighten the limit function according to the given terminating history $q$.

***Definition 5.9.*** For each label $L \in \boldsymbol{L}$:

$$(c \oplus q)(L) \ =_{\text{DF}} \ c(L) + \ell(q \downarrow C_L)$$

$$(c \ominus q)(L) \ =_{\text{DF}} \ c(L) - \ell(q \downarrow C_L)$$

where, for any label $L$, define $C_L$ as the criterion $\{L \mapsto V\}$.

***Lemma 5.10.*** Properties of $\oplus$ and $\ominus$:
For each history $q$ and function $c : \boldsymbol{L} \to \mathbb{N}$:

1) $(c \oplus q) \ominus q = c$
2) If $c \ominus q$ is well-defined then $(c \ominus q) \oplus q = c$
3) $c \ominus (q \Downarrow c)$ is always well-defined.
4) $q \Downarrow (c \oplus q) = q$
5) If $q \not\ll c$ then $(q \Downarrow c) \not\ll c$
6) If $q \not\ll c$ then $c \ominus (q \Downarrow c)$ has a zero value for some label, i.e. $\langle\rangle \not\ll c \ominus (q \Downarrow c)$
7) For any histories $q_1$ and $q_2$ we have: $(c \oplus q_1) \oplus q_2 = c \oplus (q_1 \mathbin{+\!\!+} q_2)$.

**Proof:** The proofs follow directly from the definitions. ∎

We next define the functions $\boldsymbol{f}$, $\boldsymbol{g}$ and $\boldsymbol{h}$, each of which maps a state to a set of states or set of histories.

Let **S** be any (potentially nondeterministic) statement for which the same label does not appear in both branches of any nondeterministic choice, and let $C$ be any slicing criterion. **S** and $C$ will be fixed for the rest of this section.

***Definition 5.11.*** For each initial state $s \in D_{\mathcal{H}}(V)$, function $c : \boldsymbol{L} \to \mathbb{N}$ and terminating history $q$, define:

$$\begin{aligned} \boldsymbol{f}(s) \ &=_{\text{DF}} \ \mathrm{Int}^{\mathrm{op}}_M(\mathbf{S}, V)(s) \downarrow C \\ \boldsymbol{f}^c(s) \ &=_{\text{DF}} \ \boldsymbol{f}(s) \Downarrow c \\ \boldsymbol{g}^{c,q}(s) \ &=_{\text{DF}} \ \mathrm{Int}^{\mathrm{den}}_M(\mathsf{Apply}(C, \mathbf{S}), V') \\ & \qquad (s \circledast \{\mathsf{limit} \mapsto c \oplus q, \mathsf{slice} \mapsto q\}) \end{aligned}$$

The function $\boldsymbol{h}^{c,q}(s)$ is defined from $\boldsymbol{g}$ as follows:

If $c(L) = 0$ for any label $L$, then $\langle\rangle \not\ll c$ and we cannot have $\forall L \in \boldsymbol{L}.\, \ell(q \downarrow C_L) < c(L)$ for any $q$, so we define:

$$\boldsymbol{h}^{c,q}(s) \ =_{\text{DF}} \ \{q\}$$

for these values of $c$ and any $q$ and $s$.

Otherwise, we must have $c(L) > 0$ for every label $L$.

If $\boldsymbol{g}^{c,q}(s)$ terminates, then define:

$$\boldsymbol{h}^{c,q}(s) \ =_{\text{DF}} \ \{\, t(\mathsf{slice}) \mid t \in \boldsymbol{g}^{c,q}(s) \,\}$$

If $\boldsymbol{g}^{c,q}(s)$ does not terminate, i.e. if $\perp \in \boldsymbol{g}^{c,q}(s)$, then we attempt to reduce $c$ until it does terminate. If $\boldsymbol{g}^{c',q}(s)$ aborts for every $c'$ where $\langle\rangle \ll c'$ and $c' \leqslant c$, then define:

$$\boldsymbol{h}^{c,q}(s) \ =_{\text{DF}} \ \{q \mathbin{+\!\!+} \langle\perp\rangle\}$$

For example, $\boldsymbol{h}^{c,q}(\perp) = \{q\} \mathbin{+\!\!+} \langle\perp\rangle$ for every $c$ where $\langle\rangle \ll c$.

Otherwise, $\boldsymbol{g}^{c,q}$ terminates for small values of $c$ but then fails to terminate at some point as $c$ is increased. At the boundary value of $c$, $\boldsymbol{g}^{c,q}$ appends one or more elements to slice and then terminates. If $c$ is increased, then $\boldsymbol{g}^{c,q}$ aborts without appending anything further to slice. Suppose $c(L)$ is increased by 1. Then we must previously have had $\mathsf{slice} \not\ll \mathsf{limit}$ immediately after appending an element labelled $L$ to slice: and this caused termination. Now we have $\mathsf{slice} \ll \mathsf{limit}$ and execution continues and aborts without appending anything further. So each execution path ending in a label $L$ leads to an abort.

Let $c'$ be the largest function smaller than $c$ such that $\langle\rangle \ll c'$ and $\boldsymbol{g}^{c',q}(s)$ terminates. In other words:

$$c' \leqslant c \;\wedge\; \perp \notin \boldsymbol{g}^{c',q}(s) \;\wedge \\ \forall c'' : \boldsymbol{L} \to \mathbb{N}.\, (\perp \notin \boldsymbol{g}^{c'',q}(s) \Rightarrow c'' \leqslant c')$$

For each label $L$, if $c'(L) < c(L)$ then we know that increasing $c'(L)$ by one will produce a function for which $\boldsymbol{g}$ does not terminate. Suppose that $L$ has just been appended to slice: then in the execution of $\boldsymbol{g}^{c',q}(s)$, we must have had $\mathsf{slice} \ll c'$ just before the append. If $c'(L)$ is increased, then $\boldsymbol{g}$ aborts without appending anything further. So we must have had $\mathsf{slice} \not\ll c'$ just after the append, which caused immediate termination, while increasing $c'(L)$ causes a subsequent abort. This is true for every label $L$ such that $c'(L) < c(L)$. So we will indicate this fact by appending $\langle\perp\rangle$ to each path in the final value of slice in $\boldsymbol{g}^{c',q}(s)$ which ends with any label $L$ such that $c'(L) < c(L)$.

First define the function app which computes what to append to a given history:

$$\mathsf{app}(c, c', h) \;=_{\text{DF}}\; \begin{cases} \langle\perp\rangle & \text{if } c'(h[\ell(h)][1]) < c(h[\ell(h)][1]) \\ \langle\rangle & \text{otherwise} \end{cases}$$

Now define:

$$\boldsymbol{h}^{c,q}(s) \;=_{\text{DF}} \\ \Big\{\; t(\mathsf{slice}) \mathbin{+\!\!+} \mathsf{app}(c, c', t(\mathsf{slice})) \;\Big|\; t \in \boldsymbol{g}^{c',q}(s) \;\Big\}$$

*Lemma 5.12.* Every element of $\boldsymbol{h}^{c,q}(s)$ is of the form $q \mathbin{+\!\!+} t$ for some finite history $t$.

**Proof:** This is because $\boldsymbol{h}^{c,q}(s)$ is constructed by appending to the history in the final value of slice in the execution of $\mathsf{Apply}(C, \mathbf{S})$ where slice initially has the value $q$ and $\mathsf{Apply}(C, \mathbf{S})$ can only modify slice by appending to the current value. ■

For any non-empty sequence $x$ define

$$\mathsf{butlast}(x) \;=_{\text{DF}}\; x[1 \,..\, \ell(x) - 1]$$

which is $x$ with the last element removed.

*Lemma 5.13.* If $\boldsymbol{g}^{c,q}(s)$ terminates and $q \mathbin{+\!\!+} t$ is a possible final value for slice and $t \neq \langle\rangle$, then $\mathsf{butlast}(t) \ll c$.

**Proof:** For $q \mathbin{+\!\!+} t$ to be a possible final value for slice, there must be some point in the execution of $\mathbf{S}$ where a primitive statement is executed which appends the last element of $q \mathbin{+\!\!+} t$ onto slice. Just before executing this statement we must have $\mathsf{slice} = q \mathbin{+\!\!+} \mathsf{butlast}(t)$, and for the statement to be executed we must have $\mathsf{slice} \ll \mathsf{limit}$, i.e. $q \mathbin{+\!\!+} \mathsf{butlast}(t) \ll c \oplus q$, so $\mathsf{butlast}(t) \ll c$ as required. A formal proof is by induction on the structure of $\mathbf{S}$. ■

*Theorem 5.14.* For all $s \in D_{\mathcal{H}}(V)$ and for all terminating histories $q$ and all functions $c : \boldsymbol{L} \to \mathbb{N}$, the set $\boldsymbol{h}^{c,q}(s)$ is a set of minimal elements.

In other words $\mathsf{Mins}(\boldsymbol{h}^{c,q}(s)) = \boldsymbol{h}^{c,q}(s)$.

**Proof:** By Lemma 5.12, every non-terminating history in $\boldsymbol{h}^{c,q}(s)$ is of the form $q \mathbin{+\!\!+} t \mathbin{+\!\!+} \langle\perp\rangle$ where $t$ is a terminating history. Suppose that there exists a history $t_1$ such that $q \mathbin{+\!\!+} t \mathbin{+\!\!+} t_1 \in \boldsymbol{h}^{c,q}(s)$ and $t_1 \neq \langle\rangle$ and $t_1 \neq \langle\perp\rangle$. Then by the definition of $\boldsymbol{h}$ there exists $c' < c$ such that $\boldsymbol{g}^{c',q}(s)$ terminates with $q \mathbin{+\!\!+} t$ as a possible final value for slice. In other words:

$$\perp \notin \boldsymbol{g}^{c',q}(s) \quad \text{and} \quad \exists s' \in \boldsymbol{g}^{c',q}(s).\, s'(\mathsf{slice}) = q \mathbin{+\!\!+} t$$

Since $\boldsymbol{g}^{c'',q}(s)$ aborts for any $c''$ such that $c' < c'' \leqslant c$, the value $q \mathbin{+\!\!+} t$ in slice must have got "too big" and caused subsequent guarded statements to terminate instead of aborting. So the test $\mathsf{slice} \ll \mathsf{limit}$ must have failed at this point. So we must have $q \mathbin{+\!\!+} t \not\ll c' \oplus q$, i.e. $t \not\ll c'$.

Now consider $q \mathbin{+\!\!+} t \mathbin{+\!\!+} t_1$. Let $t_1'$ be $t_1$ with any trailing $\perp$ deleted. Then $t_1' \neq \langle\rangle$ and $t_1' \neq \langle\perp\rangle$, so $t_1'$ must start with at least one proper labelled state.

By the definition of $\boldsymbol{h}$, there exists $c_1 < c$ such that $\boldsymbol{g}^{c_1,q}(s)$ terminates with $q \mathbin{+\!\!+} t \mathbin{+\!\!+} t_1'$ as a possible final value for slice. Since $c'$ is the largest function for which $\boldsymbol{g}$ terminates, we must have $c_1 \leqslant c'$. Also, since $t_1'$ is non-empty, by Lemma 5.13 we have $\mathsf{butlast}(q \mathbin{+\!\!+} t \mathbin{+\!\!+} t_1') \ll c_1 \oplus q$, so (since $t_1'$ is non-empty) $q \mathbin{+\!\!+} t \ll c_1 \oplus q$, i.e. $t \ll c_1$. But $c_1 \leqslant c'$, so we have $t \ll c'$ which is a contradiction. ■

The next theorem is the main result in this section. It shows that there is a close connection between the projected operational semantics of a program and the denotational semantics of the corresponding annotated program.

*Theorem 5.15.* For all $s \in D_{\mathcal{H}}(V)$ and for all terminating histories $q$ and all functions $c : \boldsymbol{L} \to \mathbb{N}$:

$$q \mathbin{+\!\!+} \boldsymbol{f}^{c}(s) = \boldsymbol{h}^{c,q}(s)$$

**Proof:** The proof is by induction on the structure of $\mathbf{S}$ and the depth of **while** loop nesting.

If $c(L) = 0$ for any label $L$, then $\boldsymbol{h}^{c,q}(s) = \{q\}$ and $\boldsymbol{f}^{c}(s) = \langle\rangle$ and the result follows. So in the following cases we may assume that $c(L) > 0$ for every label $L$.

**Case (1):** For the first base case, suppose that $\mathbf{S} = L\colon \mathbf{S}'$ where $\mathbf{S}'$ is any primitive statement and $L \notin \mathsf{Dom}(C)$. Let $s$ be any initial state in $D_{\mathcal{H}}(V)$. First, suppose that $\mathbf{S}'$ terminates. The set of final states for a terminating primitive statement is a singleton. Let $t$ be the (single) final state. Then:

$$\boldsymbol{f}^{c}(s) = \{\langle\langle L, t\rangle\rangle\} \downarrow C \Downarrow c = \{\langle\rangle\} \Downarrow c = \{\langle\rangle\}$$

Now:

$$\mathsf{Apply}(C, \mathbf{S}) = \textbf{if } \mathsf{slice} \ll c \textbf{ then } \mathsf{S}' \textbf{ fi}$$

which does not assign to $\mathsf{slice}$. So: $\boldsymbol{g}^{c,q}(s)$ terminates and $\mathsf{slice}$ is unchanged in every final state, i.e.:

$$\boldsymbol{g}^{c,q}(s) = \{t \circledast \{\mathsf{limit} \mapsto c \oplus q, \mathsf{slice} \mapsto q\}\} \quad (\text{since } \langle\rangle \ll c)$$

So: $\boldsymbol{h}^{c,q}(s) = \{t(\mathsf{slice})\} = q$ and the result follows.

If $\mathbf{S}'$ does not terminate on $s$ then: $\boldsymbol{f}^c(s) = D^*_{\mathcal{H}}(V) \downarrow C \Downarrow c = \{\langle\perp\rangle\}$, for all $\langle\rangle \ll c$ and $\boldsymbol{g}^{c,q}(s)$ aborts (since $\langle\rangle \ll c$). In fact, $\boldsymbol{g}^{c',q}(s)$ aborts for every $c'$ where $\langle\rangle \ll c'$ and $c' \leqslant c$. So: $\boldsymbol{h}^{c,q}(s) = \{q \mathbin{+\!\!+} \langle\perp\rangle\}$ and the result follows.

**Case (2):** Now suppose that $\mathbf{S} = L\colon \mathbf{S}'$ where $\mathbf{S}'$ is any primitive statement and $L \in \mathsf{Dom}(C)$. As above, suppose that suppose that $\mathbf{S}'$ terminates in state $t$. Then:

$$\begin{aligned}\boldsymbol{f}^c(s) &= \{\langle\langle L, t\rangle\rangle\} \downarrow C \Downarrow c \\ &= \{\langle\langle L, t \downarrow C\rangle\rangle\} \Downarrow c \\ &= \{\langle\langle L, t \downarrow C\rangle\rangle\}\end{aligned}$$

since $\langle\rangle \ll c$. Now:

$$\begin{aligned}&\mathsf{Apply}(C, \mathbf{S}) = \\ &\quad \textbf{if } \mathsf{slice} \ll \mathsf{limit} \\ &\quad\quad \textbf{then } \mathsf{S}';\ \mathsf{slice} := \mathsf{slice} \mathbin{+\!\!+} \langle\langle L, \mathsf{state}(C(L))\rangle\rangle \textbf{ fi}\end{aligned}$$

After the (terminating) execution of $\mathbf{S}'$ on initial state $s$ we have: $\mathsf{state}(C(L)) = t \downarrow C$. So:

$$\boldsymbol{g}^{c,q}(s) = \{t \circledast \{\mathsf{limit} \mapsto c \oplus q, \mathsf{slice} \mapsto q \mathbin{+\!\!+} \langle\langle L, t \downarrow C\rangle\rangle\}\}$$

since $\langle\rangle \ll c$.

So: $\boldsymbol{h}^{c,q}(s) = \{q \mathbin{+\!\!+} \langle\langle L, t \downarrow C\rangle\rangle\}$ and the result follows.

If $\mathbf{S}'$ does not terminate, then: $\boldsymbol{f}^c(s) = D^*_{\mathcal{H}}(V) \downarrow C \Downarrow c = \{\langle\perp\rangle\}$ and $\boldsymbol{g}^{c,q}(s)$ aborts (since $\langle\rangle \ll c$), and the result follows as in Case (1).

This proves the theorem for all primitive statements. For the induction step, let $\mathbf{S}$ be a compound statement and suppose that the theorem holds for all statements smaller than $\mathbf{S}$ (in terms of the number of primitive statements) and with the same maximum depth of **while** loop nesting, and for all statements (however large) with a smaller depth of **while** loop nesting. In effect, we are using a double induction on:

1) Depth of **while** loop nesting, and;
2) Size of the statement.

**Case (3):** Suppose that $\mathbf{S} = \mathbf{S}_1;\ \mathbf{S}_2$. By the induction hypothesis, the theorem holds for both $\mathbf{S}_1$ and $\mathbf{S}_2$. Let $\boldsymbol{f}_1 = (\mathrm{Int}^{\mathrm{op}}_M(\mathbf{S}_1, V)) \downarrow C$, $\boldsymbol{g}_1 = \mathrm{Int}^{\mathrm{den}}_M(\mathsf{Apply}(C, \mathbf{S}_1)), V')$ and let $\boldsymbol{f}^c_1(s) = \boldsymbol{f}_1(s) \Downarrow c$ and:

$$\boldsymbol{g}^{c,q}_1(s) = \boldsymbol{g}_1(s \circledast \{\mathsf{limit} \mapsto c \oplus q, \mathsf{slice} \mapsto q\})$$

Define $\boldsymbol{h}^{c,q}_1$ from $\boldsymbol{g}^{c,q}_1$ as above. Define $\boldsymbol{f}_2, \boldsymbol{g}_2, \boldsymbol{f}^c_2, \boldsymbol{g}^{c,h}_2$ and $\boldsymbol{h}^{c,q}_2$ from $\mathbf{S}_2$ in the same way.

Let $t$ be any element of $\boldsymbol{f}(s)$. By definition, $t$ is a minimal history. We claim that $q \mathbin{+\!\!+} (t \downarrow C \Downarrow c) \in \boldsymbol{h}^{c,q}(s)$.

By definition, there exists a history $t_1 \in \boldsymbol{f}_1(s)$ and a history $t_2 \in \boldsymbol{f}_2(\mathsf{Final}(t_1))$ such that $t = t_1 \mathbin{+\!\!+} t_2$. Let $q_1 = q \mathbin{+\!\!+} (t_1 \downarrow C \Downarrow c)$ and $c_1 = c \ominus (t_1 \downarrow C \Downarrow c)$.

By Lemma 5.10 $c \oplus q = c_1 \oplus q_1$.

There are three subcases to consider:

**Subcase (i):** Suppose $t_1 \downarrow C \not\ll c$. Then $t \downarrow C \Downarrow c = (t_1 \downarrow C \mathbin{+\!\!+} t_2 \downarrow C) \Downarrow c = t_1 \downarrow C \Downarrow c \in \boldsymbol{f}^c_1(s)$. So, by the induction hypothesis for $\mathbf{S}_1$: $q \mathbin{+\!\!+} (t \downarrow C \Downarrow c) \in \boldsymbol{h}^{c,q}_1(s)$. This is a proper state, so we must have $\boldsymbol{g}^{c,q}_1(s)$ terminating with $q_1$ as a possible final value for $\mathsf{slice}$. Let $s'_1$ be a final state containing this value. Then: $\boldsymbol{g}_2(s'_1) = \{s'_1\}$ since the guarded statement terminates immediately. So $q \mathbin{+\!\!+} (t \downarrow C \Downarrow c) \in \boldsymbol{h}^{c_1,q_1}_2(s'_1) \subseteq \boldsymbol{h}^{c,q}(s)$ as required.

**Subcase (ii):** Suppose $t_1 \downarrow C \ll c$ and $t_1$ is non-terminating. Then $t_1 \downarrow C$ ends in $\perp$ and:

$$t \downarrow C \Downarrow c = (t_1 \downarrow C \mathbin{+\!\!+} t_2 \downarrow C) \Downarrow c = t_1 \downarrow C \Downarrow c \in \boldsymbol{f}^c_1(s)$$

So, by the induction hypothesis for $\mathbf{S}_1$: $q \mathbin{+\!\!+} (t \downarrow C \Downarrow c) \in \boldsymbol{h}^{c,q}_1(s)$. This ends in $\perp$, so $\boldsymbol{g}^{c,q,}_1(s)$ must abort (since this is the only way to get $\perp$ into $\boldsymbol{h}_1$), and there is some $c' < c$ such that $\boldsymbol{g}^{c',q}_1(s)$ terminates with the terminating history $q'_1 = \mathsf{butlast}(q \mathbin{+\!\!+} (t \downarrow C \Downarrow c))$ as a possible final value for $\mathsf{slice}$. Let $s_1$ be a final state in which $\mathsf{slice}$ has the value $q'_1$. As in the proof of Theorem 5.14 we have $q_1 \not\ll c' \oplus q$, since any increase in $c'$ leads to non-termination. So $\boldsymbol{g}^{c',q}_2(s_1) = \{s_1\}$ and therefore $s_1 \in \boldsymbol{g}^{c',q}(s)$. Since $\boldsymbol{g}$ aborts on any larger $c'$, in particular $c$, we must have $q_1 \mathbin{+\!\!+} \langle\perp\rangle \in \boldsymbol{h}^{c,q}(s)$. But $q_1 \mathbin{+\!\!+} \langle\perp\rangle = q \mathbin{+\!\!+} (t \downarrow C \Downarrow c)$, so we have $q \mathbin{+\!\!+} (t \downarrow C \Downarrow c) \in \boldsymbol{h}^{c,q}(s)$ as required.

**Subcase (iii):** Suppose $t_1 \downarrow C \ll c$ and $t_1$ is terminating. Then $(t \downarrow C) \Downarrow c = ((t_1 \downarrow C) \mathbin{+\!\!+} (t_2 \downarrow C)) \Downarrow c$ and $t_2 \in \boldsymbol{f}_2(\mathsf{Final}(t_1))$. Let $q_1 = q \mathbin{+\!\!+} (t \downarrow C) \Downarrow c$ and $c_1 = c \ominus (t_1 \downarrow C)$. Then:

$$\begin{aligned}q \mathbin{+\!\!+} (t \downarrow C) \Downarrow c &= (q \mathbin{+\!\!+} (t \downarrow C) \Downarrow c) \mathbin{+\!\!+} (t_2 \downarrow C) \Downarrow c_1 \\ &= q_1 \mathbin{+\!\!+} (t_2 \downarrow C) \Downarrow c_1\end{aligned}$$

By the induction hypothesis for $\mathbf{S}_1$, $q_1 \in \boldsymbol{h}^{c,q}_1(s)$. Let $s_1 = \mathsf{Final}(t_1)$. By the induction hypothesis for $\mathbf{S}_2$: $q_1 \mathbin{+\!\!+} (t_2 \downarrow C) \Downarrow c_1 \in \boldsymbol{h}^{q_1,c_1}_2(s_1)$. By examining the definitions of $\boldsymbol{h}$ and $\boldsymbol{g}$ on a sequential composition it is clear that $\boldsymbol{h}^{q_1,c_1}_2(s_1) \subseteq \boldsymbol{h}^{c,q}(s)$, so $q \mathbin{+\!\!+} (t \downarrow C) \Downarrow c \in \boldsymbol{h}^{c,q}(s)$ as required.

Putting these subcases together, we have proved that $q \mathbin{+\!\!+} \boldsymbol{f}^c(s) \subseteq \boldsymbol{h}^{c,q}(s)$.

For the converse, let $q \mathbin{+\!\!+} t$ be any element of $\boldsymbol{h}^{c,q}(s)$. (By Lemma 5.12, every element of $\boldsymbol{h}^{c,q}(s)$ is of this form). We claim that $t \in \boldsymbol{f}(s)$.

The final value of $\mathsf{slice}$ after executing $\mathsf{Apply}(C, \mathbf{S}_1)$ followed by $\mathsf{Apply}(C, \mathbf{S}_2)$ must be the initial value (which is $q$) plus some component from the execution of $\mathbf{S}_1$, plus a further component from the execution of $\mathsf{Apply}(C, \mathbf{S}_2)$. So we must have histories $t_1$ and $t_2$ such that $t = t_1 \mathbin{+\!\!+} t_2$ and $q \mathbin{+\!\!+} t_1 \in \boldsymbol{h}^{c,q}_1(s)$. Let $q_1 = q \mathbin{+\!\!+} t_1$ and $c_1 = c \ominus t_1$. By the induction hypothesis for $\mathbf{S}_1$, we have $t_1 \in \boldsymbol{f}^c_1(s)$. Let $t'_1 \in \boldsymbol{f}_1(s)$ be such that $t_1 = t'_1 \downarrow C \Downarrow c$.

Again, there are three subcases:

**Subcase (i):** Suppose $t_1 \not\ll c$. Then at some point in the execution of $\mathsf{Apply}(C, \mathbf{S}_1)$ we appended an element to $\mathsf{slice}$ which caused the condition $\mathsf{slice} \ll \mathsf{limit}$ to become false. Due to the guard, no further statements are executed, so the final state is the state which was appended to $\mathsf{slice}$. Let $s_1 = \mathsf{Final}(t_1)$. Then $\boldsymbol{g}_2^{c_1,q_1}(s_1) = \{s_1\}$ due to the guards. So $t_2 = \langle\rangle$ and $q \mathbin{+\!\!+} t = q_1$.

Let $t_2'$ be any element of $\boldsymbol{f}_2(\mathsf{Final}(t_1))$. Then: $(t_1' \mathbin{+\!\!+} t_2') \downarrow C \Downarrow c$ is in $\boldsymbol{f}^c(s)$. Now:

$$(t_1' \mathbin{+\!\!+} t_2') \downarrow C \Downarrow c = (t_1' \downarrow C) \Downarrow c$$

since $t_1' \downarrow C = t_1 \not\ll c$. So $q \mathbin{+\!\!+} t \in q \mathbin{+\!\!+} \boldsymbol{f}^c(s)$ as required.

**Subcase (ii):** Suppose $t_1 \ll c$ and $\boldsymbol{g}_1^{c,q}(s)$ aborts. Then there exists a maximal $c' < c$ such that $\boldsymbol{g}_1^{c',q}(s)$ terminates in a state where $\mathsf{slice}$ has the value $q \mathbin{+\!\!+} \mathsf{butlast}(t_1)$ and $t_1 = \mathsf{butlast}(t_1) \mathbin{+\!\!+} \langle\bot\rangle$. So $t_1'$ must be nonterminating and therefore $t_1' \in \boldsymbol{f}(s)$, so $t = t_1 \in \boldsymbol{f}^c(s)$ as required.

**Subcase (iii):** Suppose $t_1 \ll c$ and $\boldsymbol{g}_1^{c,q}(s)$ terminates. Let $s_1$ be a final state in $\boldsymbol{g}_1^{c,q}(s)$ such that $\mathsf{slice}$ has the value $q \mathbin{+\!\!+} t_1$ and such that $q \mathbin{+\!\!+} t_1 \mathbin{+\!\!+} t_2 \in \boldsymbol{h}_2^{q_1,c_1}(s_1)$. Therefore, by the induction hypothesis for $\mathbf{S}_2$, $t_2 \in \boldsymbol{f}_2^{c_1}(s_1)$, so $t_1 \mathbin{+\!\!+} t_2 \in \boldsymbol{f}^c(s)$ as required, *provided it is a minimal element*. To prove that $t_1 \mathbin{+\!\!+} t_2$ is minimal we need to carry out the proof of the equality of $q \mathbin{+\!\!+} \boldsymbol{f}^c(s)$ and $\boldsymbol{h}^{c,q}(s)$ by induction on the lengths of elements. Prove for each $k$ that the subset of elements of $q \mathbin{+\!\!+} \boldsymbol{f}^c(s)$ of length $k$ is equal to the subset of elements of $\boldsymbol{h}^{c,q}(s)$ of length $k$. Now, if $t_2 \in \boldsymbol{f}_2^{c_1}(s_1)$ then $q \mathbin{+\!\!+} t$ must be a minimal element of $q \mathbin{+\!\!+} \boldsymbol{f}^c(s)$ since otherwise it is an extension of a smaller element. By induction, this smaller element must be in $\boldsymbol{h}^{c,q}(s)$. But then $q \mathbin{+\!\!+} t$ would not be a minimal element of $\boldsymbol{h}^{c,q}(s)$, which contradicts Theorem 5.14.

This completes the proof for Case (3).

**Case (4):** Suppose that $\mathbf{S} = \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}$ If $\mathbf{B}$ is true initially (i.e. $s \in \mathrm{int}_M(\mathbf{B}, V)$) then by the induction hypothesis for $\mathbf{S}_1$:

$$q \mathbin{+\!\!+} \boldsymbol{f}^c(s) = q \mathbin{+\!\!+} \boldsymbol{f}_1^c(s) = \boldsymbol{h}_1^{c,q}(s) = \boldsymbol{h}^{c,q}(s)$$

and similarly if $\mathbf{B}$ is false initially. The result follows from the semantics for **if**.

**Case (5):** Suppose that $\mathbf{S} = \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S}_1 \textbf{ od}$. By the induction hypothesis, the result holds for every approximation of the **while** loop (since each approximation, although it may be larger than $\mathbf{S}$, has a lower depth of **while** loop nesting). For each $k \geqslant 0$ let:

$$\begin{aligned}
\boldsymbol{f}_k(s) &= \mathrm{Int}_M^{\mathrm{op}}(\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S}_1 \textbf{ od}^k)(s) \\
\boldsymbol{f}_k^c(s) &=_{\mathrm{DF}} \boldsymbol{f}_k(s) \downarrow C \Downarrow c \\
\boldsymbol{g}_k^{c,q}(s) &= \mathrm{Int}_M^{\mathrm{den}}(\mathsf{Apply}(C, \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S}_1 \textbf{ od}^k)), V') \\
&\qquad (s \circledast \{\mathsf{limit} \mapsto c \oplus q, \mathsf{slice} \mapsto q\})
\end{aligned}$$

and define $\boldsymbol{h}_k^{c,q}$ from $\boldsymbol{g}_k^{c,q}$ in the usual way.

By the induction hypothesis: $q \mathbin{+\!\!+} \boldsymbol{f}_k^c(s) = \boldsymbol{h}_k^{c,q}(s)$ for each $k \geqslant 0$. By the definition of the operational semantics for **while**:

$$\boldsymbol{f}^c(s) = \Big(\big(\bigsqcup_{k<\omega} \boldsymbol{f}_k(s)\big) \downarrow C\Big) \Downarrow c$$

By Lemma 5.5, $\boldsymbol{f}(s)$ is a finitely branching history set, so $\boldsymbol{f}(s) \downarrow C$ is also finitely branching. Each branch in $\boldsymbol{f}(s) \downarrow C \Downarrow c$ is finite (since there are a finite number of labels, and there can be no more than $c(L)$ elements of a history with label $L$). So $\boldsymbol{f}(s) \downarrow C \Downarrow c$ is a finite set of histories: since the corresponding tree is finitely branching with a finite maximum depth. Each history in $\boldsymbol{f}(s) \downarrow C \Downarrow c$ appears in $\boldsymbol{f}_k^c(s)$ for some $k$. Let $k_1$ be the largest such $k$. Then $\boldsymbol{f}^c(s) = \boldsymbol{f}_{k_1}^c(s)$ and this holds for all $k > k_1$.

For $\boldsymbol{g}^{c,q}(s)$, either $\bot \in \boldsymbol{g}_k^{c,q}(s)$ for all $k$, or there exists a $k_2$ such that $\boldsymbol{g}^{c,q}(s) = \boldsymbol{g}_{k_2}^{c,q}(s)$ and this holds for all $k > k_2$.

Pick any $k$ larger than both $k_1$ and $k_2$ and apply the induction hypothesis for this $k$. We have:

$$q \mathbin{+\!\!+} \boldsymbol{f}^c(s) = q \mathbin{+\!\!+} \boldsymbol{f}_k^c(s) = \boldsymbol{h}_k^{c,q}(s) = \boldsymbol{h}^{c,q}(s)$$

**Case (6):** Suppose that $\mathbf{S} = (\mathbf{S}_1 \sqcap \mathbf{S}_2)$. There are three subcases to consider:

**Subcase (i):** Suppose that $\langle\bot\rangle \in \boldsymbol{f}_1^c(s)$. Then $\boldsymbol{f}_1^c(s) = \{\langle\bot\rangle\}$ since it is a set of minimal elements. Then, by the induction hypothesis for $\mathbf{S}_1$, $\boldsymbol{h}_1^{c,q}(s) = q \mathbin{+\!\!+} \{\langle\bot\rangle\}$ and by the denotational semantics for choice we have $\boldsymbol{h}_{c,q}(s) = q \mathbin{+\!\!+} \{\langle\bot\rangle\}$. By the operational semantics for choice, we have $\langle\bot\rangle \in \boldsymbol{f}^c(s)$, and so $\boldsymbol{f}^c(s) = \{\langle\bot\rangle\}$ and the result follows.

**Subcase (ii):** Suppose that $\langle\bot\rangle \in \boldsymbol{f}_2^c(s)$. The result follows as for Subcase (i).

**Subcase (iii):** Otherwise, assume $\langle\bot\rangle \notin \boldsymbol{f}_1^c(s)$ and $\langle\bot\rangle \notin \boldsymbol{f}_2^c(s)$. We claim that $\boldsymbol{f}^c(s) = \boldsymbol{f}_1^c(s) \cup \boldsymbol{f}_2^c(s)$. To prove this it is sufficient to show that $\boldsymbol{f}_1^c(s) \cup \boldsymbol{f}_2^c(s)$ is a set of minimal elements. Assume for contradiction that there exists a terminating history $t$ and history $t_1$ such that $t \mathbin{+\!\!+} \langle\bot\rangle \in \boldsymbol{f}_1^c(s) \cup \boldsymbol{f}_2^c(s)$ and $t \mathbin{+\!\!+} t_1 \in \boldsymbol{f}_1^c(s) \cup \boldsymbol{f}_2^c(s)$. From the assumption, we must have $t \neq \langle\rangle$, so since $t$ is terminating it must have a first element. Let $\langle L, t\rangle = t[1]$. Assume, w.l.o.g., that $t \mathbin{+\!\!+} \langle\bot\rangle \in \boldsymbol{f}_1^c(s)$. At this point we make use of the crucial fact that the set of labels in $\mathbf{S}_1$ is disjoint from the set of labels in $\mathbf{S}_2$. Since $L$ is a label in $\mathbf{S}_1$, it cannot appear in $\mathbf{S}_2$. So, no history in $\boldsymbol{f}_2^c(s)$ can start with a state labelled with $L$. In particular, we cannot have $t \mathbin{+\!\!+} t_1 \in \boldsymbol{f}_2^c(s)$ so we must have $t \mathbin{+\!\!+} t_1 \in \boldsymbol{f}_1^c(s)$. But then, $t \mathbin{+\!\!+} t_1$ is not a minimal element of $\boldsymbol{f}_1^c(s)$ (since $t \mathbin{+\!\!+} \langle\bot\rangle \in \boldsymbol{f}_1^c(s)$) which is a contradiction.

Similarly, we can prove that $\boldsymbol{h}^{c,q}(s) = \boldsymbol{h}_1^{c,q}(s) \cup \boldsymbol{h}_2^{c,q}(s)$, using Theorem 5.14

This completes the proof. ■

## 6 Discussion

In this section we present some example programs and show how their slices are calculated. We also show how the new semantics is able to cope with the example programs which proved so problematic with the various other attempts to define a semantics for slicing.

### 6.1 Operational Semantics

Our operational semantics defines the meaning of a program to be a function which maps each initial state to the (finite or infinite) sequence of labelled states that the program passes through as it executes. Such a sequence is called a *history*. With this semantics we can distinguish between different non-terminating programs.

Our definition of slicing has the property that any observer who observes the value of the variable of interest at the slice point will not see any difference between the sequence of values produced by the original program and the sequence produced by the slice: with the exception that the slice may produce more values, and may terminate, in cases where the original program does not terminate. To see why the original program may produce more values, consider Weiser's example program in Section 2.1 and suppose that we are slicing on the value of X at line 3. Clearly the code labelled "perform any function not involving X here" can be deleted. If this code does not terminate, then the observer will not see any value of X at line 3 in the original program (since execution will never reach that line), but may observe X with the value 1 in the slice.

This process of observing the values of certain variables at certain points in the program, while abstracting away from the rest of the behaviour of the program, is precisely what is needed to analyse the interactive behaviour of a program. If we slice on all the variables involved in each interaction at each point in the program where the interaction occurs, then the slice will exhibit all the interactive behaviours of the original program. So we can use slicing to prove that an interactive program is a correct implementation of an interactive specification, or to derive the specification of a program from its source code using the methods given in [5,9,12].

To motivate the definition of the semantic relation (semi-refinement) on sequences of labelled states, consider the program **P**:

$$\textbf{while true do } L_1 : y := 1 \textbf{ od};\ L_2 : x := 2$$

For an initial state space of $\{y\}$ this program will have the following operational semantics:

$$\langle\langle L_1, \{y \mapsto 1\}\rangle, \langle L_1, \{y \mapsto 1\}\rangle, \langle L_1, \{y \mapsto 1\}\rangle, \dots\rangle$$

A proposed slice of **P** for $x$ at label $L_2$ is program **S**:

$$L_2 : x := 2$$

This has semantics $\langle\langle L_2, \{x \mapsto 2\}\rangle\rangle$. The projection of the semantics for **P** on $x$ at label $L_2$ will delete an infinite sequence of states, so this sequence is replaced by $\langle\bot\rangle$. This example shows why the slicing relation on histories must allow $\langle\langle L_2, \{x \mapsto 2\}\rangle\rangle$ as a valid slice of $\langle\bot\rangle$. More generally, since the assignment could be any statement, the relation must allow *any* history as a valid slice of $\langle\bot\rangle$. These considerations show that the semantic slicing relation is indeed *semi-refinement* (see Section 4.5.1).

### 6.2 Example Programs

We now consider the examples discussed earlier which proved problematical for previous definitions of a semantics for slicing.

For the Weiser example $W_1$ in Figure 2, the operational semantics will include the state $\langle L_2, X \mapsto 1\rangle$ when $X$ is true, and so $X$ will have the value 1 at the write statement on this initial state. So this statement cannot be sliced away. For example $W_2$ in Figure 3, the projection of the loop at $L_3$ is $\langle\bot\rangle$, so the statement at $L_4$ has no effect on the semantics and *can* be sliced away when we are slicing at the end of the program. This is in accordance with Weiser's discussion and confirms that semi-refinement does indeed correctly model Weiser slicing (unlike all the other semantic relations we have discussed).

#### 6.2.1 *Non-termination*

(a) Program $P_9$

```
while $y > 0$ do
  $L_2 : y := y + 1$;
  $L_3 : y := y - 2$ od;
$L : x := 1$
```

(b) Program $P_{10}$

```
while $y > 0$ do
  $L_2 : y := y + 1$ od;
$L : x := 1$
```

Figure 10. Labelled slicing examples

Now consider programs $P_1$ and $P_2$. The programs $P_9$ and $P_{10}$ in Figure 10 are $P_1$ and $P_2$ with labels added.

Suppose we are slicing on $x$ at label $L$. Program $P_9$ terminates, so every history consists of a finite sequence of states labelled $L_2$ or $L_3$, followed by a single state labelled $L$ in which $x$ has the value 1. Program $P_{10}$ does not terminate when $y > 0$ initially, so the history is an infinite sequence of states labelled $L_2$ and no state labelled $L$. The projection of this infinite sequence is therefore the singleton sequence $\langle\bot\rangle$.

So, $P_{10}$ is not a valid slice of $P_9$, but $P_9$ *is* a valid (semantic) slice for $P_{10}$, as required.

#### 6.2.2 *Unreachable code*

Figure 11 shows labelled copies of $P_7$ and $P8$ from Figure 8 where we are slicing on $y$ at $L$. For program $P_{11}$, the statement $L_2 : x := 1$ is never executed, so appears in none of the histories. This program terminates with the assignment $y := x$, so if we slice on this statement the projected history contains a single state in which $y$ contains the initial value of $x$. So, as with the lazy semantics, transfinite semantics and trajectory semantics, a valid slice is $L : y := x$.

(a) Program $P_{11}$

```
if true
  then $L_1$ : skip
  else $L_2 : x := 1$ fi;
$L : y := x$
```

(b) Program $P_{12}$

```
while true do
  $L_1$ : skip od;
$L_2 : x := 1$;
$L : y := x$
```

Figure 11. Unreachable Code

For $P_{12}$ the history is an infinite sequence of states labelled $L_1$. The projection of this history on $L$ is $\langle\perp\rangle$ so any program is a valid slice, including $L : y := x$. As explained in Section 3.2.5, in the various forms of lazy semantics, transfinite semantics and trajectory semantics, the statement $L_2 : x := 1$ must be included in all slices. For example, in the trajectory semantics, every finite trajectory includes the statements $L_2 : x := 1$ and $L : y := x$, so the projection of every trajectory on $L$ assigns the value 1 to $y$, so any syntactic slice must also ensure that $y$ is assigned the value 1 in the trajectory semantics. This is only possible if the slice also includes the statement $L_2 : x := 1$ since otherwise $y$ will be assigned an incorrect value. So, in the lazy or transfinite semantics, every slice must include the unreachable statement $L_2 : x := 1$. With our semantics, every history for $P_8$ consists of an infinite sequence of states labelled $L_1$ so the projection on $L$ is simply $\langle\perp\rangle$ and any program is a valid slice. In particular, the program $L : y := x$ is a valid slice on $y$ at $L$, and the unreachable statement $L_2 : x := 1$ can be deleted, as required.

A slightly more complex example is Figure 12. The only assignment to $x$ which can reach label $L$ is $L_3 : x := 4$, so a valid semantic slice in the our operational semantics is $L : y := 4$. Note that with the test of $y$ removed, the preceding code which assigns to $y$ can also be removed. With any of the lazy, transfinite or trajectory semantics the assignment $x := 3$ cannot be removed, so the test of $y$ and therefore the code which assigns to $y$ all have to be preserved in any slice.

```
code which assigns to $y$ here
**if** $y = 0$
  **then while true do** $L_1$ : **skip od**; $L_2 : x := 3$
  **else** $L_3 : x := 4$ **fi**;
$L : y := x$
```

Figure 12. Unreachable Code

### 6.3 Denotational Semantics

The WSL denotational semantics models a program as a function which maps each initial state to a final state, or set of final states. An interactive program can be modelled by defining the input as a sequence of events, and the output as a sequence of interactions. This model will work *provided that the program eventually terminates*. But there are many interactive programs which are designed to run forever without terminating normally. In effect, the output of the program is an infinite sequence of interactions with the environment. But the denotational semantic model cannot handle processing an infinite sequence of input data or generating an infinite sequence of output. This would appear to render the model unsuitable: but in reality, no program can actually process an infinite amount of data, so it is sufficient to consider the set of all *finite* sequences of interactions which can occur up to some (arbitrary) abnormal termination event. We cannot place any upper limit on the number of interactions, but if we can prove that the program correctly generates *all finite sequences* of interactions, then we can be satisfied that it will work in practice.

For a non-terminating slice of a non-terminating program, the sequence of values produced by the program and the sequence produced by the slice may both be infinite, but it is sufficient to examine all finite prefixes of the infinite sequences. If the observer terminates both programs after a certain (finite) number of interactions then the two finite sequences can be compared. If the results are correct for an observer with an arbitrarily large, but still finite, amount of patience, then we may assume that the slice is correct.

This discussion motivates our scheme of modifying the program by providing an extra parameter (the limit parameter) which indicates the number of interactions we are interested in. The modified program will terminate immediately on reaching that number of interactions. If this modified program gives correct results for *every* limit value, then we may assume that the original program is correct for all practical purposes.

Consider the following non-terminating program which reads a sequence of integers from the input port and prints the running totals for the sum and product of the integers read so far. The sum is send to output port1 and the product to port2.

```
sum := $0$;
prod := $1$;
**while true do**
  read(**var** input, $n$);
  sum := sum + $n$;
  print("Sum = ", sum **var** port1);
  prod := prod $*$ $n$;
  print("Product = ", prod **var** port2) **od**
```

The program interacts with the environment in three places: the statement read(input, $n$) reads a number from the input port and each of the two print statements writes to an output port.

Suppose we are interested in the interactions of this program with the ports input and port1. To analyse these interactions, we label the read statement L1 and label the first print statement L2 and use the following slicing criterion:

$$C = \{\mathsf{L1} \mapsto \{\mathsf{input}\}, \mathsf{L2} \mapsto \{\mathsf{port1}\}\}$$

The function Apply($C$, **S**), when applied to the program above, produces the following terminating program:

```
**if** slice $\ll$ limit **then** sum := $0$ **fi**;
**if** slice $\ll$ limit **then** prod := $1$ **fi**;
**while** slice $\ll$ limit **do**
  **if** slice $\ll$ limit
    **then** read(**var** input, $n$);
         slice := slice $+\!\!\!+$ $\langle\langle$L1, $\langle$"input", input$\rangle\rangle\rangle$ **fi**;
  **if** slice $\ll$ limit **then** sum := sum + $n$ **fi**;
  **if** slice $\ll$ limit
    **then** print("Sum = ", sum **var** port1);
         slice := slice $+\!\!\!+$ $\langle\langle$L2, $\langle$"port1", port1$\rangle\rangle\rangle$ **fi**;
  **if** slice $\ll$ limit **then** prod := prod $*$ $n$ **fi**;
  **if** slice $\ll$ limit
    **then** print("Product = ", prod **var** port2) **fi od**
```

The apply function adds a lot of code to the program, but most of this can be removed via automated transformations.

Applying FermaT's semantic slice transformation [49], and slicing on the final value of the variable slice, we get:

```
if slice $\ll$ limit then sum := $0$ fi;
while slice $\ll$ limit do
  read(var input, $n$);
  slice := slice ++ $\langle\langle L_1, \langle$"input", input$\rangle\rangle\rangle$;
  if slice $\ll$ limit then sum := $n$ + sum fi;
  if slice $\ll$ limit
    then print("Sum = ", sum var port1);
         slice := slice ++ $\langle\langle L_2, \langle$"port1", port1$\rangle\rangle\rangle$ fi od
```

Removing the annotations gives us a simplified non-terminating program which is semantically equivalent to the original program in its interactions via input and port1:

```
sum := $0$;
while true do
  read(var input, $n$);
  sum := sum + $n$;
  print("Sum = ", sum var port1) od
```

Our annotation process converts a non-terminating loop into a loop which may terminate: but the whole program terminates when any loop is forced to terminate. The finite trajectory semantics [50] also converts non-terminating loops to terminating loops, but then allows execution to continue after the loop. This means that code which appears after a non-terminating loop can affect the semantics and may have to be included in any slice (even though such code cannot be executed). In contrast, when our annotated code causes termination of an otherwise non-terminating loop, the whole program is terminated, so any non-executable code in the original program is still non-executable in the annotated program: this can be clearly seen in the examples in the Section 6.2.2. The finite trajectory semantics also allows a non-terminating program as a valid slice of a terminating program.

### 6.4 Slicing Nondeterministic Programs

Extending the programming language to include nondeterminism has some interesting results.

The operational semantics for the nondeterministic language maps each initial state to the set of possible histories, while the denotational semantics maps each initial state to the set of possible final states. If the denotational semantics includes $\perp$ in the set of final states, then it is defined to include all other states. The corresponding requirement for the operational semantics is: If the operational semantics includes a history ending in $\perp$ then it is defined to include all *extensions* of this history. In this section we discuss the interactions between slicing and nondeterministic choice. The statement $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$ will execute one of the statements $\mathbf{S}_1$ or $\mathbf{S}_2$.

Note that any program of the form ($\mathbf{S} \sqcap$ **abort**) is equivalent to **abort** in both operational and denotational semantics. This is because any component of a nondeterministic choice is a valid refinement of the choice. So ($\mathbf{S} \sqcap$ **abort**) is refined by **abort**. It is also a refinement of **abort** (because every statement is a refinement of **abort**), so therefore it must be semantically equivalent to **abort**.

**Example 1**

First, consider the program $L_1\colon x := 1;$ **abort**. Slicing on $x$ at $L_1$ gives this projected semantics:

$$\{\langle\langle L_1, \{x \mapsto 1\}\rangle, \perp\rangle\}$$

The slice therefore has to include the statement $L_1\colon x := 1$.

**Example 2**

Now consider the program:

$$(L_1\colon x := 1;\ \textbf{abort} \sqcap L_2\colon x := 2;\ \textbf{abort})$$

Slicing on $x$ at $L_1$ gives this projected semantics:

$$\{\langle\perp\rangle, \langle\langle L_1, \{x \mapsto 1\}\rangle, \perp\rangle\}$$

plus all the extensions of $\langle\perp\rangle$. The history $\langle\langle L_1, \{x \mapsto 1\}\rangle, \perp\rangle$ is an extension of $\langle\perp\rangle$, so this set is identical to $\{\langle\perp\rangle\}$, plus its extensions. So the projected semantics is identical to the semantics for **abort**, and the statement $L_1\colon x := 1$ does *not* have to be included in the slice. Similarly a slice on $x$ at $L_2$ does not have to include $L_2\colon x := 2$. However, if we slice on both labels simultaneously then the projected semantics is:

$$\{\langle\langle L_1, \{x \mapsto 1\}\rangle, \perp\rangle, \langle\langle L_2, \{x \mapsto 2\}\rangle, \perp\rangle\}$$

and any slice for this criterion *must* include both assignments.

This example illustrates the fact that a slice on two labels simultaneously may have to include statements which are not needed when slicing on either label separately. In general, taking the union of slices on two separate labels does not necessarily give a slice for both labels simultaneously.

**Example 3**

Consider the program:

$$(L_1\colon x := 1;\ \textbf{abort} \sqcap L_2\colon x := 2;\ L_3\colon x := 3;\ \textbf{abort})$$

The operational semantics gives this history set:

$$f(s) = \{\langle\langle L_1, \{x \mapsto 1\}\rangle, \perp\rangle,\\ \langle\langle L_2, \{x \mapsto 2\}\rangle, \langle L_3, \{x \mapsto 3\}\rangle, \perp\rangle\}$$

For any slicing criterion $C$ on a single label, the projection of the operational semantics for $f$ is $\{\langle\perp\rangle\}$. In other words, if we slice on just $L_1$, $L_2$ or $L_3$ then the result is **abort**. But if we slice on $L_1$ and $L_2$ the result is:

$$f(s) = \{\langle\langle L_1, \{x \mapsto 1\}\rangle, \perp\rangle, \langle\langle L_2, \{x \mapsto 2\}\rangle, \perp\rangle\}$$

Again, the result of slicing on two labels simultaneously is not the same as combining the slices for each label.

In the denotational semantics we annotate the program by appending a new value to the new variable slice at each slice point. The Apply function also inserts tests of the form slice $\ll$ limit at various points in the program.

This example illustrates the more precise test slice $\ll$ limit instead of the simple test $\ell(\mathsf{slice}) < N$. The problem is that a simple length limit will not be able to "detect" the statement at label $L_3$ when we are slicing on all three labels simultaneously.

If the maximum length is 1 then the statement will terminate as soon as $L_2$ has been executed, while if the length is 2 then the $L_1$ branch will cause the program to abort. So, with a simple integer length limit $\ell(\textsf{slice}) < N$ there is no initial value for $N$ for which the program:

$$(L_1\colon x := 1;\ \textbf{abort}\ \sqcap\ L_2\colon x := 2;\ \textbf{abort})$$

is different from the program:

$$(L_1\colon x := 1;\ \textbf{abort}\ \sqcap\ L_2\colon x := 2;\ L_3\colon x := 3;\ \textbf{abort})$$

This example shows why the annotation must add a test which can specify a different limit for each label in the slicing criterion. For example, the limit function $\{L_1 \mapsto 1, L_2 \mapsto 2, L_3 \mapsto 1\}$ will ensure that all three labels are included in states appended to slice, so the program with $L_3\colon x := 3$ deleted will *not* be considered as a valid slice when we are slicing on all three labels. The limit of 2 on $L_2$ means that the execution of $L_2 : x := 2$ will not cause an enforced termination when slice is tested just before $L_3$, so the program will execute $L_3 : x := 3$. The latter statement has a limit of one, so termination is enforced before the following **abort** can be executed. Note that with this limit function, the annotated program is guaranteed to terminate, so there is always a final state for which we can check the value of the variable slice.

### 6.5 More Non-terminating Programs

Define the infinite loop:

$$\textbf{loop}\ =_{\text{DF}}\ \textbf{while true do skip od}$$

In the denotational semantics world, **loop** is equivalent to **abort**, but the operational semantics of the two programs are different. The semantics of **loop** maps each initial state $s$ to the infinite sequence:

$$\langle\langle L_0, s\rangle, \langle L_0, s\rangle, \langle L_0, s\rangle, \dots\rangle$$

while the semantics of **abort** maps each initial state $s$ to the sequence $\langle\perp\rangle$. In both semantics, **abort** is refined by any program, but under operational semantics **loop** is only refined by itself.

As with **abort**, however, **loop** satisfies the relation: **loop**; **S** $\approx$ **loop** for any statement **S**: so (unlike the transfinite semantics, the lazy semantics and the finite trajectory semantics) there is no possibility of code *after* a non-terminating loop having any effect on the semantics of the program as a whole.

For any initial state $s$ and any slicing criterion $C$ which does not include $L_0$ in its domain, the projection of **loop** on $C$ is $\langle\perp\rangle$. So, on every slicing criterion which excludes $L_0$, the projection of **loop** is the same as the projection of **abort**.

The projection of the semantics of **loop**; end : **skip** on the criterion $C = \{\textsf{end} \mapsto X\}$ is $\langle\perp\rangle$ for every initial state. But the projection of end : **skip**, for initial state $s$ is $\langle\langle\textsf{end}, s\rangle\rangle$. If we want to be able to "slice away" the infinite loop, then the slicing relation must allow $\langle\langle\textsf{end}, s\rangle\rangle$ as a valid slice of $\langle\perp\rangle$. Of course, the **skip** statement could be an arbitrary statement, so the slicing relation must allow any statement as a valid slice of $\langle\perp\rangle$.

Another example is the program $\mathbf{S}_1$:

$$L_1\colon x := 4;\ \textbf{while true do}\ L_0\colon \textbf{skip od};\ L_2\colon x := 5$$

where we are slicing on the value of $x$ at $L_1$ and $L_2$. Again, we would like to "slice out" the middle loop and get $\mathbf{S}_2$:

$$L_1\colon x := 4;\ L_2\colon x := 5$$

as a valid slice. The projection of $\mathbf{S}_1$ is $\langle\langle L_1, \{x \mapsto 4\}\rangle, \perp\rangle$ for each initial state, while the projection of $\mathbf{S}_2$ is $\langle\langle L_1, \{x \mapsto 4\}\rangle, \langle L_2, \{x \mapsto 5\}\rangle\rangle$. Here, the projection of $\mathbf{S}_2$ is formed from the projection of $\mathbf{S}_1$ by replacing the trailing $\perp$ by a labelled state.

If we replace the statement $L_2\colon x := 5$ in both $\mathbf{S}_1$ and $\mathbf{S}_2$ by any arbitrary statement, then we can see that the semantics for the modified $\mathbf{S}_2$ could be *any* extension of the semantics for the modified $\mathbf{S}_1$. In other words, the slice could be any semi-refinement of the original program (as far as the semantic relation is concerned). This means that, just as in [20], the semantic relation for slicing is semi-refinement.

#### 6.5.1 Another Non-terminating Program

Our final example is slicing in the middle of a non-terminating program. This example is based on one published in [52], which was based on an example in [22]. See [53] for a detailed discussion of the original example. We have modified the program to convert it to a non-terminating program with the slice point in the middle.

Suppose we are slicing on the value of $x$ at label $L$ in the program **S**, which is:

```
while true do
  $i := y(x)$;
  $c := z(x)$;
  while $p(i)$ do
    if $q(c)$
      then $x := f$; $c := g(i)$ fi;
    $i := h(i)$ od;
  $L$: skip od
```

A dataflow algorithm for slicing will note that $x$ is assigned in an **if** statement with condition $q(c)$, so there is a control dependency on this condition. $c$ is also assigned in the **if** statement, which is inside a loop, so there is a data dependency on the assignment to $c$. So any dataflow based slicing algorithm will include the assignment to $c$ in the slice. Similarly, any slicing definition which is based on the dataflow will insist that the assignment to $c$ be included in *any* valid slice.

However, in practice the assignment to $c$ cannot affect the value of $x$. If $q(c)$ is false on the first iteration of the inner loop (assuming the loop executes at all), then $c$ does not get modified by the loop and so neither does $x$. On the other hand, if $q(c)$ is true, then $x$ gets assigned the constant value $f$. Any further assignments to $x$ are to the same value and so have no effect. So changes to the value of $q(c)$ on subsequent iterations of the loop will not affect the value of $x$ at $L$.

Our slicing criterion is: $C = \{L \mapsto \{x\}\}$, so the annotated program Apply$(C, \mathbf{S})$ is:

```
while slice $\ll$ limit $\wedge$ true do
  if slice $\ll$ limit then $i := y(x)$ fi;
  if slice $\ll$ limit then $c := z(x)$ fi;
  while slice ≪ limit ∧ $p(i)$ do
    if $q(c)$
      then if slice ≪ limit then $x := f;\ c := g$ fi fi;
    if slice ≪ limit then $i := h(i)$ fi;
    if slice ≪ limit
      then slice := slice ++ $\langle\langle L, \mathsf{state}(C(L))\rangle\rangle$ fi od
```

Use the invariant slice ≪ limit to simplify the loop body:

```
while slice ≪ limit do
  $i := y(x)$;
  $c := z(x)$;
  while slice ≪ limit ∧ $p(i)$ do
    if $q(c)$
      then $x := f;\ c := g(i)$ fi;
    $i := h(i)$ od;
  slice := slice ++ $\langle\langle L, x\rangle\rangle$ od
```

Applying FermaT's semantic slicer [10,49] to this program to slice on the final value of slice we get:

```
while slice ≪ limit do
  $i := y(x)$;
  $c := z(x)$;
  if slice ≪ limit ∧ $p(i)$ ∧ $q(c)$
    then $x := f$ fi;
  slice := slice ++ $\langle\langle L, x\rangle\rangle$ od
```

which corresponds to the following un-annotated program, which we will call $\mathbf{S}'$:

```
while true do
  $i := y(x)$;
  $c := z(x)$;
  if $p(i)$ ∧ $q(c)$
    then $x := f$ fi;
  $L$: skip od
```

Note that the inner **while** loop has been converted to a simple **if** statement, and that the assignment to $c$ inside the inner loop has been deleted.

Computing the operational semantics for **S** and $\mathbf{S}'$, projecting the semantics on $C$ and determining directly that the projected semantics for $\mathbf{S}'$ is a semi-refinement of the projected semantics for **S**, would appear to be a much harder task than computing the denotational semantic slice of the annotated program. This is partly because the denotational semantic slice process can make use of any valid WSL transformations: including loop unrolling and constant propagation.

## 7 Applications of Semantic Slicing

If we drop the syntactic requirement from Weiser's definition of slicing (to define a semantic slice), then any statement is a valid slice of a non-terminating statement. This "semantic freedom" is necessary for removing irrelevant code from programs. A practical example of removing irrelevant code is removing error handling code: suppose we are interested in determining the behaviour of a program under normal (non error) conditions. In commercial systems, such as large commercial assembler systems, error handling code can amount to 60% or more of the source code in a module [11, 49]. Before a program has been analysed, it can be difficult to determine which parts of the program are purely concerned with error handling, since the error handling code can add a large number of irrelevant control flow paths: this is especially true for unstructured assembler code. By the time control flow reaches an instruction which causes abnormal termination (ABEND), or code to display an error message, we can be certain that we are within the error handling code. If these statements are replaced by **abort** statements, then the result is an *abstraction* of the original program which is nevertheless equivalent to the original under "normal" (non error) input states. Subsequent semantic slicing is able to eliminate the error handling code and any preceding tests for error conditions. For example, consider the statement:

**if B then S else** ... error ... **fi**

where the condition **B** tests for a particular error. This is abstracted to:

**if B then S else** ... **abort** ... **fi**

which is equivalent to:

**if B then S else abort fi**

If we slice this particular **abort** statement to the statement **S** then we get:

**if B then S else S fi**

which is equivalent to **S**: and we have eliminated the error handling code *and* the preceding test for the error. In [49] we show that much of the above process can be automated and applied to unstructured assembler to produce an abstract version of the code: on average a 6,000 line assembler listing is condensed down to a 132 line high level language abstraction.

### 7.1 Analysing a Non-Terminating Interactive Program

By slicing on some or all of the points of interaction of an interactive program we can analyse the behaviour of a non-terminating program using the FermaT Maintenance Environment (FME) which can be downloaded from

http://www.gkc.org.uk/fermat.html

This example uses a regular action system and loops with multiple exits. In [2] we show that these constructs can be transformed into equivalent code using **while** loops, so the equivalence proved in Section 5 is still valid. A regular action system is denoted by the keywords **actions**... **endactions** and consists of the name of the starting action followed by a list of actions. Each action has a name and a body (a statement sequence). Execution of any action body will always lead to an action **call**: so no call can ever return and a **call** acts in the same way as a **goto** in other languages. A call of the special action $Z$ (which has no body) causes the whole action system to terminate immediately, with control passing to the next statement after the action system (if any).

```
var ⟨FL1 := 0, FL2 := 0⟩ :
  actions Inside :
  Loop1 ≡
    if msg = "" then call Again fi;
    if FL2 = 0 ∨ last ≠ msg
      then call Next
```

```
        else call More fi end
    Next ≡
      if FL1 = 1
        then !P Write(last, total var output) fi;
      FL1 := 1;
      total := 0;
      call More end
    More ≡
      total := total + count;
      FL2 := 1;
      call Loop end
    Loop ≡
      last := msg;
      !P Read( var msg, count, input);
      call Loop1 end
    Again ≡
      if FL2 = 1
        then !P Write(last, total var output) fi;
      call Inside end
    Inside ≡
      FL1 := 0;  FL2 := 0;
      last := msg;
      !P Read( var msg, count, input);
      call Loop1 end endactions end
```

This regular action system does not contain a **call** $Z$ so it will never terminate. The program interacts with the environment via the Read and Write operations (the !P denotes a call to a procedure which is external to this module). The FermaT maintenance environment is based on denotational semantics, so if we analyse this program using FME we can transform it to the program **abort**. This is because FME preserves the behaviour of the program on termination, but this program never terminates.

The program makes extensive use of labels and unstructured branches, and also uses two flag variables FL1 and FL2 to direct control flow. This is typical of assembler code, particularly embedded assembler.

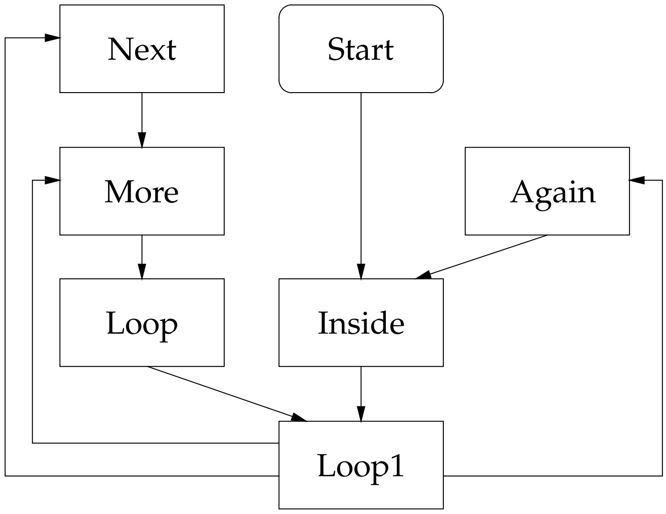


Figure 13. Action System Call Graph

We are interested in the outputs produced by the program on interaction with the environment, so we annotate the Write calls as follows:

```
!P Write(last, total var output);
slice := slice ++ ⟨⟨last, total⟩⟩;
if ℓ(slice) ⩾ limit then call Z fi
```

With these annotations, the denotational semantics of the annotated program preserves the interactive behaviour of the original program, as proved in Section 5.

The first step is to restructure the action system into a set of nested loops and **if** statements. FME includes a transformation Collapse_Action_System which uses heuristics to guide the application of other transformations to achieve this restructuring. The output produced by Collapse_Action_System is:

```
var ⟨FL1 := 0, FL2 := 0⟩ :
  do FL1 := 0;
     FL2 := 0;
     last := msg;
     !P Read( var msg, count, input);
     do if msg = “”
          then if FL2 = 1
                 then !P Write(last, total var output);
                      slice := slice ++ ⟨⟨last, total⟩⟩;
                      if ℓ(slice) ⩾ limit
                        then exit(2)
                        else exit(1) fi
                 else exit(1) fi fi;
        if FL2 = 0 ∨ last ≠ msg
          then if FL1 = 1
                 then !P Write(last, total var output);
                      slice := slice ++ ⟨⟨last, total⟩⟩;
                      if ℓ(slice) ⩾ limit
                        then exit(2) fi fi;
               FL1 := 1;
               total := 0 fi;
        total := (total + count);
        FL2 := 1;
        last := msg;
        !P Read( var msg, count, input) od od end
```

The next step is to further restructure the program in order to eliminate flag variables where possible. This is achieved by transforming the code to move flag tests closer to the places where the flag is set: duplicating small amounts of code if necessary. Then Constant_Propagation can be applied to eliminate tests and assignments where the flag variable has a known value. If all references to the flag can be eliminated in this way, then the flag variable can be removed: since it is a local variable.

A transformation Flag_Removal uses heuristics to guide the application of other transformations to eliminate flags. In this case, both flags could be removed, with the following result:

```
do last := msg;
   !P Read( var msg, count, input);
   if msg ≠ “”
     then total := 0;
          total := count;
          do last := msg;
             !P Read( var msg, count, input);
             if msg = “”
               then !P Write(last, total var output);
```

```
              slice := slice ++ ⟨⟨last, total⟩⟩;
              if $\ell$(slice) $\geqslant$ limit
                then exit(2)
                 else exit(1) fi fi;
      if last $\neq$ msg
        then !P Write(last, total var output);
              slice := slice ++ ⟨⟨last, total⟩⟩;
              if $\ell$(slice) $\geqslant$ limit
                then exit(2) fi;
              total := 0 fi;
      total := (count + total) od fi od
```

The result still has some duplicated statements, particularly the Write calls and assignments to slice and count. These were merged by applying further transformations using the FME graphical front end to select and apply each transformation. The transformation engine checks the validity of each transformation before it is applied. The result is:

```
do !P Read( var msg, count, input);
   do if msg = “”
          then exit(1) fi;
      total := count;
      last := msg;
      !P Read( var msg, count, input);
      while last = msg $\wedge$ msg $\neq$ “” do
         total := (count + total);
         !P Read( var msg, count, input) od;
      !P Write(last, total var output);
      slice := slice ++ ⟨⟨last, total⟩⟩;
      if $\ell$(slice) $\geqslant$ limit
         then exit(2) fi od od
```

We can now delete the annotations and convert the inner loop to a **while** loop to give a transformed non-terminating program which is guaranteed to be equivalent to the original program in all of its external behaviour:

```
do !P Read( var msg, count, input);
   while msg $\neq$ “” do
     total := count;
     last := msg;
     !P Read( var msg, count, input);
     while last = msg $\wedge$ msg $\neq$ “” do
       total := (count + total);
       !P Read( var msg, count, input) od;
     !P Write(last, total var output) od od
```

This restructured program can then be transformed into a specification, if required, (although such a process is beyond the scope of this paper). See [5,9,54].

The program reads a sequence of message and count values from the input stream. The stream of data may be split into groups where all the message for each element in the group is the same. Groups may also be separated by one or more empty messages (whose count values are ignored). The outermost loop processes each group separated by empty messages. The next inner loop processes a single group where all message values are the same, while the innermost loop processes each data element. The message and total count for each group is written to the output stream.

## 8 Conclusion

The interactive behaviour of a program can be modelled as a semantic slice, where the points of interest are the statements which interact with the environment and the variables of interest are all the variables involved in each interaction. Many interactive systems are designed to run continuously, without terminating: so there is a practical need for a clear formal definition of program slicing for non-terminating programs.

In this paper we have reviewed many attempts by a number of authors to define a suitable semantic relation which precisely captures the meaning of a program slice, and therefore captures the interactive behaviour of a system. None of these efforts succeeded in capturing Weiser's informal definition of a program slice: particularly where potentially non-terminating code and nondeterminism are concerned. We have developed a new operational semantics and a projection operator which defines the meaning of a program slice applied to interactive and potentially non-terminating programs. We have also developed an extension of the denotational semantics which also defines the meaning of a program slice for all these classes of program. This “extension” is simply the standard WSL denotational semantics applied to an annotated program: so all the WSL program analysis techniques developed over the last 25 years of research [2,3,4,5,6,7,8,9,10,11,12] can now be applied to the analysis of non-terminating interactive programs. We prove that the two different semantics are equivalent for slicing purposes, and this gives us confidence that we have captured the informal concept of a “program slice” for interactive, potentially non-terminating programs.

The FermaT program transformation system is distributed under the GNU General Public License (GPL). The FermaT Maintenance Environment is a graphical front end to the FermaT program transformation system which is also distributed under the GNU General Public License (GPL). Both of these tools can be downloaded from the following web site:

http://www.gkc.org.uk/fermat.html

The WSL source code for the example program in Section 7.1 is included in the distribution in the project directory `fme/paper`.

## References

[1] M. Weiser, “Program slices: formal, psychological, and practical investigations of an automatic program abstraction method,” University of Michigan, Ann Arbor, PhD Thesis, 1979.

[2] M. Ward, “Proving Program Refinements and Transformations,” Oxford University, DPhil Thesis, 1989, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/thesis⟩.

[3] M. Ward and K. H. Bennett, “A Practical Program Transformation System For Reverse Engineering,” presented at Working Conference on Reverse Engineering, May 21–23, 1993, Baltimore MA, 1993, doi:10.1109/WCRE.1993.287763.

[4] E. J. Younger and M. Ward, “Understanding Concurrent Programs using Program Transformations,” presented at Proceedings of the 1993 2nd Workshop on Program Comprehension, 8th-9th July, Capri, Italy, 1993, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/cap.ps.gz⟩.

[5] M. Ward, “Specifications from Source Code—Alchemists’ Dream or Practical Reality?,” presented at 4th Reengineering Forum, September 19-21, 1994, Victoria, Canada, Sept., 1994.

[6] M. Ward and K. H. Bennett, “Formal Methods to Aid the Evolution of Software,” *International Journal of Software Engineering and Knowledge Engineering*, 5, no. 1, pp. 25–47, 1995, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/evolution-t.ps.gz⟩ doi:dx.doi.org/10.1142/S0218194095000034.

[7] M. Ward, “The FermaT Assembler Re-engineering Workbench,” presented at International Conference on Software Maintenance (ICSM), 6th–9th November 2001, Florence, Italy, 2001.

[8] M. Ward, “The Formal Transformation Approach to Source Code Analysis and Manipulation,” presented at IEEE International Workshop on Source Code Analysis and Manipulation Florence, Italy, 10th November, Los Alamitos, California, USA, 2001.

[9] M. Ward, “Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations,” *Science of Computer Programming, Special Issue on Program Transformation*, 52, no. 1–3, pp. 213–255, 2004, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/migration-t.ps.gz⟩ doi:dx.doi.org/10.1016/j.scico.2004.03.007.

[10] M. P. Ward, H. Zedan and T. Hardcastle, “Conditioned Semantic Slicing via Abstraction and Refinement in FermaT,” presented at 9th European Conference on Software Maintenance and Reengineering (CSMR) Manchester, UK, March 21–23, 2005.

[11] M. Ward and H. Zedan, “Combining Dynamic and Static Slicing for Analysing Assembler,” *Science of Computer Programming*, 75, no. 3, pp. 134–175, Mar., 2010, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/combined-slicing-t.pdf⟩ doi:10.1016/j.scico.2009.11.001.

[12] ———, “Provably Correct Derivation of Algorithms Using FermaT,” *Formal Aspects of Computing*, 26, no. 5, pp. 993–1031, Sept., 2014, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/trans-prog-t.pdf⟩.

[13] M. Weiser, “Program slicing,” *IEEE Trans. Software Eng.*, 10, no. 4, pp. 352–357, July, 1984.

[14] ———, “Programmers use slices when debugging,” *Comm. ACM*, 25, no. 7, pp. 352–357, July, 1984.

[15] F. Tip, “A Survey of Program Slicing Techniques,” *Journal of Programming Languages,*, 3, no. 3, pp. 121–189, Sept., 1995.

[16] D. W. Binkley and K. B. Gallagher, “A Survey of Program Slicing,” *Advances in Computers*, 43, pp. 1–52, 1996.

[17] A. D. Lucia, “Program slicing: Methods and applications,” presented at First IEEE International Workshop on Source Code Analysis and Manipulation, Los Alamitos, California, USA, 2001.

[18] D. Binkley and M. Harman, “A Survey of Empirical Results on Program Slicing,” in *Advances in Computers*, vol. 62, M. Zelkowitz, Ed. San Diego, CA: Academic Press, pp. 105–178, 2004.

[19] J. Silva, “A Vocabulary of Program Slicing-Based Techniques,” *ACM Computing Surveys*, 44, no. 3, June, 2012, Article No. 12.

[20] M. Ward and H. Zedan, “Slicing as a Program Transformation,” *Trans. Programming Lang. and Syst.*, 29, no. 2, pp. 1–52, Apr., 2007, ⟨http://www.cse.dmu.ac.uk/∼mward/martin/papers/slicing-t.ps.gz⟩ doi:doi.acm.org/10.1145/1216374.1216375.

[21] R. Ettinger, “Refactoring via Program Slicing and Sliding,” Oxford University, DPhil Thesis, 2007, ⟨http://progtools.comlab.ox.ac.uk/members/rani/sliding_thesis_esub101006.pdf⟩.

[22] S. Danicic, “Dataflow Minimal Slicing,” London University, PhD Thesis, 1999.

[23] D. Binkley, M. Harman and S. Danicic, “Amorphous Program Slicing,” *Journal of Systems and Software*, 68, no. 1, pp. 45–64, Oct., 2003.

[24] D. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid.* New Yory New York, Basic Books, 1979, ISBN 978-0-465-02656-2.

[25] S. Horwitz, T. Reps and D. Binkley, “Interprocedural slicing using dependence graphs,” *Trans. Programming Lang. and Syst.*, 12, no. 1, pp. 26–60, Jan., 1990.

[26] D. Binkley, “Precise Executable Interprocedural Slices,” *ACM Letters on Programming Languages and Systems*, 2, pp. 31–45, Mar., 1993.

[27] J. Hatcliff, M. B. Dwyer and H. Zheng, “Slicing Software for Model Construction,” *Journal of Higher-order and Symbolic Computation*, 13, no. 4, pp. 315–353, Dec., 2000.

[28] B. Korel, I. Singh, L. Tahat and B. Vaysburg, “Slicing of State Based Models. ,” presented at IEEE International Conference on Software Maintenance (ICSM’03), Los Alamitos, California, USA, Sept., 2003.

[29] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer and J. Hatcliff, “A New Foundation For Control-Dependence and Slicing for Modern Program Structures,” *Proceedings of the European Symposium On Programming (ESOP’05), Edinburg, Scotland*, 3444, pp. 77–93, Apr., 2005.

[30] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, M. B. Dwyer, “A New Foundation For Control-Dependence and Slicing for Modern Program Structures,” *Trans. Programming Lang. and Syst.*, 29, no. 5, pp. 27:1–27:43 and N. definitionsforcontrol dependence, Aug., 2007.

[31] R. Halder and A. Cortei, “Abstract program slicing on dependence condition graphs,” *Science of Computer Programming*, 78, pp. 1240–1263, 2013, doi:10.1016/j.scico.2012.05.007.

[32] I. Mastroeni and D. Zanardini, “Data dependencies and program slicing: from syntax to abstract semantics,” presented at Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM ’08, San Francisco, California, USA, 2008.

[33] A. D. Lucia, M. Harman, R. Hierons and J. Krinke, "Unions of Slices are not Slices," presented at 7th European Conference on Software Maintenance and Reengineering Benevento, Italy March 26-28th, 2003.

[34] S. Horwitz, J. Prins and T. Reps, "Integrating non-interfering versions of programs," *Trans. Programming Lang. and Syst.*, 11, no. 3, pp. 345–387, July, 1989.

[35] G. A. Venkatesh, "The semantic approach to program slicing.," *SIGPLAN Notices*, 26, no. 6, pp. 107–119, 1991, Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, June 26-28.

[36] T. Reps and W. Yang, "The Semantics of Program Slicing," *Computer Sciences Technical Report*, 777, pp. 1–30, June, 1988.

[37] E. W. Dijkstra, "Guarded commands, non-determinacy and formal derivation of programs," *Comm. ACM*, 18, no. 8, pp. 453–457, 1975, EWD 472.

[38] ———, *A Discipline of Programming*. Englewood Cliffs, NJ, Prentice-Hall, 1976.

[39] R. Cartwright and M. Felleisen, "The Semantics of Program Dependence," *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation*, 24, no. 7, pp. 13–27, 1989, Publishes as SIGPLAN Notices.

[40] R. Giacobazzi and I. Mastroeni, "Non-Standard Semantics for Program Slicing," *Higher-Order and Symbolic Computation*, 16, no. 4, pp. 297–339, Dec., 2003.

[41] H. Nestra, "Fractional Semantics," in *Proceedings of AMAST 2006* (Lect. Notes in Comp. Sci.), vol. 4019, M. Johnson and V. Vene, Eds. New York–Heidelberg–Berlin: Springer-Verlag, pp. 278–292, 2006.

[42] S. Danicic, M. Harman, J. Howroyd and 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*, 72, no. 2, pp. 123–240, 2007.

[43] M. Kamkar, "Interprocedural Dynamic Slicing with Applications to Debugging and Testing," Linkoping University, S-581 83 Linkoping, Sweden, PhD Thesis, 1993.

[44] M. Harman and S. Danicic, "Amorphous Program Slicing," presented at 5th IEEE International Workshop on Program Comprehesion (IWPC'97), Dearborn, Michigan, USA, May 1997.

[45] M. Harman, L. Hu, M. Munro and X. Zhang, "GUSTT: An Amorphous Slicing System Which Combines Slicing and Transformation," presented at Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), Los Alamitos, California, USA, 2001.

[46] M. Harman, M. Munro, D. Binkley, S. Danicic, M. Aoudi and L. Ouarbya, "Syntax-Directed Amorphous Slicing," *Automated Software Engineering (ASE)*, 11, no. 1, pp. 27–61, 2004.

[47] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss and B. Korel, "A Formalisation of the Relationship between Forms of Program Slicing," *Science of Computer Programming*, 62, no. 3, pp. 228–252, 2006.

[48] M. Ward, "Properties of Slicing Definitions," presented at Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, Los Alamitos, California, USA, Sept., 2009, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/properties-final.pdf⟩ doi:10.1109/SCAM.2009.12.

[49] M. Ward, H. Zedan, M. Ladkau and S. Natelberg, "Conditioned Semantic Slicing for Abstraction; Industrial Experiment," *Software Practice and Experience*, 38, no. 12, pp. 1273–1304, Oct., 2008, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/slicing-paper-final.pdf⟩ doi:doi.wiley.com/10.1002/spe.869.

[50] R. W. Barraclough, D. Binkley, S. Danicic, M. Harman, R. M. Hierons, Á. Kiss, M. Laurence and L. Ouarbya, "A Trajectory-Based Strict Semantics for Program Slicing," *Theoretical Computer Science*, 411, no. 11-13, pp. 1372–1386, Mar., 2010, ISSN:0304-3975.

[51] D'enes Kőnig, *Theorie der Endlichen und Unendlichen Graphen: Kombinatorische Topologie der Streckenkomplexe*. Leipzig, Akad. Verlag, 1936.

[52] Anon, "Which Lines do not affect x?," presented at Ceramic Mug given to attendees of the First Source Code Analysis and Manipulation Workshop, Florence, Italy, 10th November, 2001.

[53] M. Ward, "Slicing the SCAM Mug: A Case Study in Semantic Slicing," presented at Third IEEE International Workshop on Source Code Analysis and Manipulation 26th–27th September, Los Alamitos, California, USA, 2003.

[54] M. Ward, "Reverse Engineering from Assembler to Formal Specifications via Program Transformations," presented at 7th Working Conference on Reverse Engineering, 23-25th November, Brisbane, Queensland, Australia, 2000, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/wcre2000-t.pdf⟩.