Slicing as a Program Transformation

MARTIN WARD
Martin.Ward@durham.ac.uk
De Montfort University
and
HUSSEIN ZEDAN
zedan@dmu.ac.uk
De Montfort University

The aim of this paper is to provide a unified mathematical framework for program slicing which places all slicing work, for sequential programs, on a sound theoretical foundation. The main advantage to a mathematical approach is that it is not tied to a particular representation. In fact the mathematics provides a sound basis for *any* particular representation. We use the WSL (Wide Spectrum Language) program transformation theory as our framework. Within this framework we define a new semantic relation, *semi-refinement* which lies between semantic equivalence and semantic refinement. Combining this semantic relation, a syntactic relation (called *reduction*) and WSL's **remove** statement, we can give mathematical definitions for backwards slicing, conditioned slicing, static and dynamic slicing and semantic slicing as program transformations in the WSL transformation theory. A novel technique of "encoding" operational semantics within a denotational semantics allows the framework to handle "operational slicing". The theory also enables the concept of slicing to be applied to nondeterministic programs. These transformations are implemented in the industry-strength FermaT transformation system.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques
General Terms: Slicing, Program Transformation, FermaT, Formal Methods, Reverse Engineering, Reengineering
Additional Key Words and Phrases: Slicing, Program Transformation, FermaT, Reverse Engineering, Reengineering

1. INTRODUCTION

Program slicing is a decomposition technique that extracts from a program those statements relevant to a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the value of variable v at statement s?" An observer cannot distinguish between the execution of a program and execution of the slice, when attention is focused on the value of v in statement s.

Slicing was first described by Mark Weiser [43] as a debugging technique [44], and has since proved to have applications in testing, parallelisation, integration, software safety, program understanding and software maintenance. Survey articles by Binkley and Gallagher [9] and Tip [28] include extensive bibliographies.

Since the publication of Weiser's paper, there have been many papers published which describe different algorithms for slicing and different slicing variants. Most

Authors' address: Software Technology Research Lab., De Montfort University, The Gateway, Leicester LE1 9BH, UK

of these papers give an informal definition of the meaning of a program slice and concentrate attention on defining and computing program dependencies (data dependencies, control dependencies and so on). This focus on dependency calculations confuses the *definition* of a slice with various *algorithms* for computing slices. In fact, it may come as a surprise to some readers that the definitions in this paper make no reference to data or control dependencies!

The aim of this paper is to provide a unified mathematical framework for program slicing which places all slicing work, for sequential programs, in a sound theoretical framework. This mathematical approach has many advantages: not least of which is that it is not tied to a particular representation. In fact the mathematics provides a sound basis for *any* sequential program representation and any program slicing technique. We use the WSL (Wide Spectrum Language) program transformation theory as our framework. WSL is a programming language based on first order infinitary logic. A first order logic language $\mathcal{L}$ is extended to a simple programming language by adding the kernel constructs of WSL (see Section 3.1). The kernel language is expanded into a powerful programming language in a series of language "layers" with the kernel as the base layer and each new layer defined in terms of the previous layer.

Within this framework we define a new semantic relation, *semi-refinement* which lies between semantic equivalence and semantic refinement. Combining this semantic relation, a syntactic relation (called *reduction*) and WSL's **remove** statement, we can give mathematical definitions for backwards slicing, conditioned slicing, static and dynamic slicing and semantic slicing as program transformations in the WSL transformation theory.

A *program transformation* is any operation on a program which generates a semantically equivalent program. Weiser defined a program slice $\mathbf{S}$ as a *reduced, executable program* obtained from a program $\mathbf{P}$ by removing statements, such that $\mathbf{S}$ replicates part of the behaviour of $\mathbf{P}$. A slice is not generally a transformation of the original program because a transformation has to preserve the *whole* behaviour of the program, while in the slice some statements which affect the values of some output variables (those not in the slice) may have been deleted.

Suppose we are slicing on the *end* of the program. The subset of the behaviour we want to preserve is simply the final values of one or more variables (the variables in the slicing criterion). If we modify both the original program and the slice to delete the unwanted variables from the state space, then the two modified programs *will* be semantically equivalent.

If we are interested in slicing on variables in the middle of a program, then we can "capture" the values of the variables at this point by assigning to a new variable, slice. Preserving the final value of slice ensures that we preserve the values of the variables of interest at the point of interest.

This discussion suggests that the operation of slicing can be formalised as a program transformation on a modification of the original program. A *syntactic slice* of a program $\mathbf{S}$ is any program $\mathbf{S}'$ formed by deleting statements from $\mathbf{S}$ such that if $\mathbf{S}$ and $\mathbf{S}'$ are modified by adding assignments to slice at the appropriate places and removing all other variables from the final state space, then the two modified programs are semantically equivalent.

A key insight of this formulation is that it defines the concept of slicing as a combination of two relations: a syntactic relation (statement deletion) and a semantic relation (which shows what subset of the semantics has been preserved).

This point may be illustrated by the following example:

(1) $x := y + 1;$
(2) $y := y + 4;$
(3) $x := x + z$

where we are interested in the final value of x. The assignment to y on line 2 can be sliced away:

(1) $x := y + 1;$
(2) $x := x + z$

These two programs are not equivalent, because they have different effects on the variable y. However, the **remove** statement in WSL can be used to remove y from the final state space, so its value does not form part of the semantics of the program. Informally, **remove**(y) at the end of a program has the effect of converting y to a local variable initialised to the same value as the corresponding global variable. So, if we modify both programs by appending a **remove**(y) statement, then the resulting programs are equivalent.

A *slicing criterion* usually consists of a set of variables plus a program point at which the values of the variables must be preserved by the slice. Suppose we want to slice on the value of i in at the top of the **while** loop body (just before line 3) in this program:

(1) $i := 0;\ s := 0;$
(2) **while** $i < n$ **do**
(3) $\quad s := s + i;$
(4) $\quad i := i + 1$ **od**;
(5) $i := 0$

Slicing on i at the end of the program (i.e. after line 5) would allow $i := 0$ as a valid slice, but we are interested in the sequence of values taken on by i at the top of the loop body. So we need some way to get this sequence of values to appear at end of the program. A simple way to do this is to add a new variable, slice, which records this sequence of values:

(1) $i := 0;\ s := 0;$
(2) **while** $i < n$ **do**
(3) $\quad \mathsf{slice} := \mathsf{slice} +\!\!+ \langle i \rangle;$
(4) $\quad s := s + i;$
(5) $\quad i := i + 1$ **od**;
(6) $i := 0;$

where $\mathsf{slice} := \mathsf{slice} +\!\!+ \langle i \rangle$ appends the value of i to the list of values in slice.

Slicing on slice at the end of the program is equivalent to slicing on i at the top of the loop. If we add the statement **remove**(i, s, n) to remove all the other output variables, then the result can be transformed into the equivalent program:

(1) $i := 0;$
(2) **while** $i < n$ **do**

(3) $\quad \mathsf{slice} := \mathsf{slice} \mathbin{+\!\!+} \langle i \rangle;$
(4) $\quad i := i + 1$ **od**;
(5) **remove**(i, s, n)

which yields the sliced program:

(1) $i := 0;$
(2) **while** $i < n$ **do**
(3) $\quad i := i + 1$ **od**

Binkley et al [8] define a slice as a combination of a syntactic ordering (any computable, transitive, reflexive relation on programs) and a semantic requirement which is any equivalence relation on a projection of program semantics. It turns out in practice that semantic *equivalence* is too strong a requirement to place on the definition of a slice. In fact we will show in Section 6.2 that there is *no* equivalence relation which is suitable for defining program slicing!

The WSL refinement relation is also shown to be unsuitable as the semantic requirement for program slicing. Instead, we define a new semantic relation, *semi-refinement*, which captures precisely what we need for the formal mathematical definition of slicing.

1.1 Outline of the Paper

In Section 2 we give a brief introduction to refinement and equivalence which forms the basis for the FermaT transformation theory. In Section 3 we introduce the WSL kernel language and specification statement and describe the foundations of transformation theory, concluding with some example transformations. Section 4 describes the extensions to the kernel language which make up the full WSL language. In Section 5 we discuss how to "encode" an operational semantics in FermaT's denotational semantics. This allows the framework to handle "operational slicing". Section 6 defines program slicing as a WSL transformation, and discusses the various generalisations and extensions which arise from this definition. Section 7 describes the FermaT transformation system and the various slicing algorithms that are implemented in FermaT. Section 8 gives some slicing examples, including a conditioned semantic slicing example, and Section 9 concludes. The Appendix describes the implementation of the simple slicer in $\mathcal{META}$WSL.

2. REFINEMENT AND EQUIVALENCE

In this section we give a brief introduction to program transformation theory, sufficient to give a rigorous definition of a program slice in terms of the theory.

The way to get a rigourous proof of the correctness of a transformation is to first define precisely when two programs are "equivalent", and then show that the transformation in question will turn any suitable program into an equivalent program. To do this, we need to make some simplifying assumptions: for example, we usually ignore the execution time of the program. This is not because we don't care about efficiency—far from it—but because we want to be able to use the theory to prove the correctness of optimising transformations: where a program is transformed into a more efficient version.

More generally, we ignore the internal sequence of state changes that a program carries out: we are only interested in the initial and final states (but see Section 5 for a discussion of operational semantics).

Our mathematical model defines the semantics of a program as a function from states to sets of states. For each initial state s, the function f returns the set of states $f(s)$ which contains all the possible final states of the program when it is started in state s. A special state $\perp$ indicates nontermination or an error condition. If $\perp$ is in the set of final states, then the program might not terminate for that initial state. If two programs are both potentially nonterminating on a particular initial state, then we consider them to be equivalent on that state. (A program which might not terminate is no more useful than a program which never terminates: we are just not interested in whatever else it might do). So we define our semantic functions to be such that whenever $\perp$ is in the set of final states, then $f(s)$ must include every other state. This restriction also simplifies the definition of semantic equivalence and refinement (see below). These "semantic functions" are used in the denotational semantics of Tennet [27] and Stoy [26].

If two programs have the same semantic function then they are said to be *equivalent*. A *transformation* is an operation which takes any program satisfying its applicability conditions and returns an equivalent program.

A generalisation of equivalence is the notion of *refinement*: one program is a refinement of another if it terminates on all the initial states for which the original program terminates, and for each such state it is guaranteed to terminate in a possible final state for the original program. In other words, a refinement of a program is *more defined* and *more deterministic* than the original program. If program $\mathbf{S}_1$ has semantic function f_1 and $\mathbf{S}_2$ has semantic function f_2, then we say that $\mathbf{S}_1$ *is refined by* $\mathbf{S}_2$ (or $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$), and write $\mathbf{S}_1 \leq \mathbf{S}_2$ if for all initial states s we have $f_2(s) \subseteq f_1(s)$. If $\mathbf{S}_1$ may not terminate for initial state s, then by definition $f_1(s)$ contains $\perp$ and every other state, so $f_2(s)$ can be anything at all and the relation is trivially satisfied. The program **abort** (which terminates on no initial state) can be refined to *any* other program. Insisting that $f(s)$ include every other state whenever $f(s)$ contains $\perp$ ensures that refinement can be defined as a simple subset relation.

3. TRANSFORMATION THEORY

In this section we describe the kernel language and transformation theory.

3.1 The Kernel Language

The Kernel language is based on infinitary first order logic, originally developed by Carol Karp [20]. Infinitary logic is an extension of ordinary first order logic which allows conjunction and disjunction over (countably) infinite lists of formulae, but quantification over finite lists of variables.

We need just four primitive statements and three compound statements to define the whole kernel language. Let $\mathbf{P}$ and $\mathbf{Q}$ be any infinitary logical formulae and $\boldsymbol{x}$ and $\boldsymbol{y}$ be any finite lists of variables. The primitive statements are:

(1) **Assertion:** $\{\mathbf{P}\}$ is an assertion statement which acts as a partial **skip** statement. If the formula $\mathbf{P}$ is true then the statement terminates immediately with-

out changing any variables, otherwise it aborts (we treat abnormal termination and non-termination as equivalent, so a program which aborts is equivalent to one which never terminates);

(2) **Guard:** $[\mathbf{Q}]$ is a guard statement. It always terminates, and enforces $\mathbf{Q}$ to be true at this point in the program *without changing the values of any variables*. It has the effect of restricting previous nondeterminism to those cases which will cause $\mathbf{Q}$ to be true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore all the final states will satisfy any desired condition (including $\mathbf{Q}$);

(3) **Add variables:** $\mathsf{add}(\boldsymbol{x})$ ensures that the variables in $\boldsymbol{x}$ are in the state space (by adding them if necessary) and assigns arbitrary values to the variables in $\boldsymbol{x}$. The arbitrary values may be restricted to particular values by a subsequent guard;

(4) **Remove variables:** $\mathsf{remove}(\boldsymbol{y})$ ensures that the variables in $\boldsymbol{y}$ are *not* present in the state space (by removing them if necessary).

while the compound statements are:

(1) **Sequence:** $(\mathbf{S}_1;\ \mathbf{S}_2)$ executes $\mathbf{S}_1$ followed by $\mathbf{S}_2$;

(2) **Nondeterministic choice:** $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$ chooses one of $\mathbf{S}_1$ or $\mathbf{S}_2$ for execution, the choice being made nondeterministically;

(3) **Recursion:** $(\mu X.\mathbf{S}_1)$ where X is a *statement variable* (a symbol taken from a suitable set of symbols). The statement $\mathbf{S}_1$ may contain occurrences of X as one or more of its component statements. These represent recursive calls to the procedure whose body is $\mathbf{S}_1$.

Some of these constructs, particularly the guard statement, may be unfamiliar to many programmers, while other more familiar constructs such as assignments and conditional statements appear to be missing. It turns out that assignments and conditionals, which used to be thought of as "atomic" operations, can be constructed out of these more fundamental constructs. On the other hand, the guard statement by itself is unimplementable in any programming language: for example, the guard statement [**false**] is guaranteed to terminate in a state in which **false** is true. In the semantic model this is easy to achieve: the semantic function for [**false**] has an *empty* set of final states for each proper initial state. As a result, [**false**] is a valid refinement for *any* program. Morgan [23] calls this construct "miracle". Such considerations have led to the Kernel language constructs being described as "the Quarks of Programming": mysterious entities which cannot be observed in isolation, but which combine to form what were previously thought of as the fundamental particles.

Assignments can be constructed from a sequence of **add** statements and guards. For example, the assignment $x := 1$ is constructed by adding x (thereby nondeterministically scrambling any existing value in x) and then restricting the nondeterminism to give a particular value: $(\mathsf{add}(\langle x\rangle);\ [x = 1])$. For an assignment such as $x := x + 1$ we need to record the new value of x in a new variable, x' say, before copying it into x. So we can construct $x := x + 1$ as follows:

$$\mathsf{add}(\langle x'\rangle);\ [x' = x + 1];\ \mathsf{add}(\langle x\rangle);\ [x = x'];\ \mathsf{remove}(x')$$

Conditional statements are constructed by combining guards with nondeterministic choice. For example, **if B then $\mathbf{S}_1$ else $\mathbf{S}_2$ fi** can be constructed as

$$([\mathbf{B}];\ \mathbf{S}_1)\ \sqcap\ ([\neg\mathbf{B}];\ \mathbf{S}_2)$$

3.2 Semantics of the Kernel Language

Let V and W be finite sets of variables, called *state spaces*, and $\mathcal{H}$ be a set of values. A *state* is either the special state $\perp$ which indicates nontermination or error, or is a function from V to $\mathcal{H}$. The set of all states on V is denoted $D_{\mathcal{H}}(V)$ where $D_{\mathcal{H}}(V) =_{\text{DF}} \{\perp\} \cup \mathcal{H}^V$. A *state predicate* is a set of proper states (i.e. states other than $\perp$), with the set of all state predicates denoted $E_{\mathcal{H}}(V)$, so $E_{\mathcal{H}}(V) =_{\text{DF}} \wp(\mathcal{H}^V)$. A *state transformation* is a function which maps a state in $D_{\mathcal{H}}(V)$ to a set of states in $D_{\mathcal{H}}(W)$, where $\perp$ maps to $D_{\mathcal{H}}(W)$ and if $\perp$ is in the output, then so is every other state. The set of all state transformations from V to W may therefore be defined as:

$$F_{\mathcal{H}}(V, W) =_{\text{DF}} \{\, f : D_{\mathcal{H}}(V) \to (E_{\mathcal{H}}(W) \cup \{D_{\mathcal{H}}(W)\}) \mid f(\perp) = D_{\mathcal{H}}(W) \,\}$$

The non-recursive kernel language statements are defined as state transformations as follows:

$$\begin{aligned}
\{e\}(s) &=_{\text{DF}} \begin{cases} \{s\} & \text{if } s \in e \\ D_{\mathcal{H}}(W) & \text{otherwise} \end{cases} \\
[e](s) &=_{\text{DF}} \begin{cases} \{s\} & \text{if } s \in e \\ \varnothing & \text{otherwise} \end{cases} \\
\textsf{\textbf{add}}(\boldsymbol{x})(s) &=_{\text{DF}} \{\, s' \in D_{\mathcal{H}}(W) \mid \forall y \in W.(y \notin \boldsymbol{x} \Rightarrow s'(y) = s(y)) \,\} \\
\textsf{\textbf{remove}}(\boldsymbol{y})(s) &=_{\text{DF}} \{\, s' \in D_{\mathcal{H}}(W) \mid \forall y \in W.(s'(y) = s(y)) \,\} \\
(f_1;\ f_2)(s) &=_{\text{DF}} \bigcup \{\, f_2(s') \mid s' \in f_1(s) \,\} \\
(f_1\ \sqcap\ f_2)(s) &=_{\text{DF}} f_1(s) \cup f_2(s)
\end{aligned}$$

Note that for **add**$(\boldsymbol{x})$ we must have $W = V \cup \boldsymbol{x}$ and for **remove**$(\boldsymbol{y})$ we must have $W = V \setminus \boldsymbol{y}$. Also note that if $\perp \in f_1(s)$ then $\perp \in (f_1;\ f_2)(s)$.

Three fundamental state transformations in $F_{\mathcal{H}}(V, V)$ are: Ω, Θ and Λ. These give the semantics of the statements **abort**, **null** and **skip**, where **abort** is defined as $\{$**false**$\}$, **null** is defined as $[$**false**$]$ and **skip** is defined as $\{$**true**$\}$. For each proper $s \in D_{\mathcal{H}}(V)$:

$$\Omega(s) =_{\text{DF}} D_{\mathcal{H}}(V) \qquad \Theta(s) =_{\text{DF}} \varnothing \qquad \Lambda(s) =_{\text{DF}} \{s\}$$

We define recursion in terms of a function on state transformations:

Definition 3.1. *Recursion:* Suppose we have a function $\mathcal{F}$ which maps the set of state transformations $F_{\mathcal{H}}(V, V)$ to itself. We want to define a recursive state transformation from $\mathcal{F}$ as the limit of the sequence of state transformations $\mathcal{F}(\Omega)$, $\mathcal{F}(\mathcal{F}(\Omega))$, $\mathcal{F}(\mathcal{F}(\mathcal{F}(\Omega)))$, … With the definition of state transformation given above, this limit $(\mu.\mathcal{F})$ has a particularly simple and elegant definition:

$$(\mu.\mathcal{F}) =_{\text{DF}} \bigsqcup_{n<\omega} \mathcal{F}^n(\Omega)$$

where $\bigsqcup$ on a set of state transformations is defined by pointwise intersection:

$$(\bigsqcup X)(s) \;=_{\text{DF}}\; \bigcap \{\, f(s) \mid f \in X \,\}$$

We say $\mathcal{F}^n(\Omega)$ is the "nth truncation" of $(\mu.\mathcal{F})$: as n increases the truncations get closer to $(\mu.\mathcal{F})$. The later truncations provide more information about $(\mu.\mathcal{F})$—more initial states for which it terminates and a restricted set of final states. The $\bigsqcup$ operation collects together all this information to form $(\mu.\mathcal{F})$.

With this definition, $(\mu.\mathcal{F})$ is well defined for *every* function $\mathcal{F} : F_{\mathcal{H}}(V, V) \to F_{\mathcal{H}}(V, V)$. But if we want our recursive state transformations to satisfy the property $\mathcal{F}((\mu.\mathcal{F})) = (\mu.\mathcal{F})$ (in other words, to be a *fixed point* of the $\mathcal{F}$ function) then we need to put some further restrictions on $\mathcal{F}$.

To prove this claim we need The "flat" order on states [26] $s \sqsubseteq t$ is defined as true when $s = \perp$ or $s = t$. It induces an order on state transformations as follows:

Definition 3.2. *If $f, g \in F_{\mathcal{H}}(V, W)$ are state transformations then $f_1 \sqsubseteq f_2$ iff:*

$$\forall s \in D_{\mathcal{H}}(V).\, (\forall t_1 \in f_1(s).\, \exists t_2 \in f_2(s).\, t_1 \sqsubseteq t_2 \;\wedge\; \forall t_2 \in f_2(s).\, \exists t_1 \in f_1(s).\, t_1 \sqsubseteq t_2)$$

An equivalent formulation is: $f_1 \sqsubseteq f_2$ iff $\forall s \in D_{\mathcal{H}}(V).\, (\perp \in f_1(s) \;\vee\; f_1(s) = f_2(s))$.

A function $\mathcal{F}$ on state transformations is *monotonic* if $\forall f \in F_{\mathcal{H}}(V, V).\, f \sqsubseteq \mathcal{F}(f)$. If in addition, $\mathcal{F}(\bigsqcup F) = \bigsqcup \{\, \mathcal{F}(f) \mid f \in F \,\}$ for every *directed* set $F \subseteq F_{\mathcal{H}}(V, V)$ then $\mathcal{F}$ is *continuous*. (A directed set F is such that for every $f_1, f_2 \in F$ there exists $g \in F$ such that $f_1 \sqsubseteq g$ and $f_2 \sqsubseteq g$.)

For *continuous* functions $\mathcal{F}$, we have $\mathcal{F}((\mu.\mathcal{F})) = (\mu.\mathcal{F})$. So the state transformation $(\mu.\mathcal{F})$ is a fixed point for the continuous function $\mathcal{F}$; it is easily shown to be the *least* fixed point. All functions generated by the kernel language constructs are continuous.

A state transformation can be thought of as either a specification of a program, or as a (partial) description of the behaviour of a program. If f is a specification, then for each initial state s, $f(s)$ is the allowed set of final states. If $\perp \in f(s)$ then the specification does not restrict the program in any way for initial state s, since every other state is also in $f(s)$.

Similarly, if f is a program description, then $\perp \notin f(s)$ means that the program is guaranteed to terminate in some state in $f(s)$ when started in state s.

Program f *satisfies* specification g precisely when $\forall s.\, (f(s) \subseteq g(s))$.

A program f_2 is a *refinement* of program f_1 if f_2 satisfies every specification satisfied by f_1, i.e. $\forall g.\, (\forall s.\, (f_1(s) \subseteq g(s)) \Rightarrow \forall s.\, (f_2(s) \subseteq g(s)))$. It is easy to see that refinement and satisfaction, as defined above, are identical relations. So from now on we only talk about refinement, with the understanding that anything we say about refinement applies equally well to satisfaction of specifications.

Definition 3.3. *Refinement:* Given two state transformations f_1 and f_2 in $F_{\mathcal{H}}(V, W)$ we say that f_2 *refines* f_1, or f_1 *is refined by* f_2, and write $f_1 \;\leq\; f_2$ when f_2 is at least as defined and at least as deterministic as f_2. More formally:

$$f_1 \;\leq\; f_2 \iff \forall s \in D_{\mathcal{H}}(V).\, f_2(s) \subseteq f_1(s)$$

Note that if $\perp \in f_1(s)$ then $f_1(s) = D_{\mathcal{H}}(W)$ and f_2 can be any state transformation.

If we fix on a particular set of values and 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 3.4. 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 $\mathcal{H}$. A structure M for $\mathcal{L}$ defines an *interpretation* of each formula $\mathbf{B}$ as a state predicate $\mathrm{int}_M(\mathbf{B}, V) \in E_{\mathcal{H}}(V)$, consisting of the states which satisfy the formula, and also interprets each statement $\mathbf{S}$ as a state transformation $\mathrm{int}_M(\mathbf{S}, V, W) \in F_{\mathcal{H}}(V, W)$.

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:

$$\{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$$

Given a countable set Δ of sentences (formulae with no free variables), a *model* for Δ is any structure M such that every formula in Δ is interpreted as true under M. If $\mathbf{S}_1$ and $\mathbf{S}_2$ are statements such that $\mathrm{int}_M(\mathbf{S}_1, V, W) \leq \mathrm{int}_M(\mathbf{S}_2, V, W)$ for every model M of Δ then we say that $\mathbf{S}_2$ is a semantic refinement of $\mathbf{S}_1$ and write $\Delta \models \mathbf{S}_1 \leq \mathbf{S}_2$.

3.3 The Specification Statement

For our transformation theory to be useful for both forward and reverse engineering it is important to be able to represent abstract specifications as part of the language and this motivates the definition of the *Specification statement*. Then the refinement of a specification into an executable program, or the reverse process of abstracting a specification from executable code, can both be carried out within a single language. Specification statements are also used in semantic slicing (see Section 6.9).

Informally, a specification describes *what* a program does without defining exactly *how* the program is to work. This can be formalised by defining a specification as a list of variables (the variables whose values are allowed to change) and a formula defining the relationship between the old values of the variables, the new values, and any other required variables.

With this in mind, we define the notation $\boldsymbol{x} := \boldsymbol{x}'.\mathbf{Q}$ where $\boldsymbol{x}$ is a sequence of variables and $\boldsymbol{x}'$ the corresponding sequence of "primed variables", and $\mathbf{Q}$ is any formula. This assigns new values to the variables in $\boldsymbol{x}$ so that the formula $\mathbf{Q}$ is true where (within $\mathbf{Q}$) $\boldsymbol{x}$ represents the old values and $\boldsymbol{x}'$ represents the new values. If there are no new values for $\boldsymbol{x}$ which satisfy $\mathbf{Q}$ then the statement aborts. The formal definition is:

$$\boldsymbol{x} := \boldsymbol{x}'.\mathbf{Q} \;=_{\text{DF}}\; \{\exists \boldsymbol{x}'.\,\mathbf{Q}\};\ \textsf{add}(\boldsymbol{x}');\ [\mathbf{Q}];\ \textsf{add}(\boldsymbol{x});\ [\boldsymbol{x} = \boldsymbol{x}'];\ \textsf{remove}(\boldsymbol{x}')$$

Note that the specification statement is never null (the final set of states is non-empty for every initial state), and so obey's Dijkstra's "Law of Excluded Miracles" [14] (see Section 4).

3.4 Weakest Preconditions

Dijkstra introduced the concept of weakest preconditions [14] as a tool for reasoning about programs. For a given program $\mathbf{P}$ and condition $\mathbf{R}$ on the final state space, the weakest precondition $\mathrm{WP}(\mathbf{P}, \mathbf{R})$ is the weakest condition on the initial state such that if $\mathbf{P}$ is started in a state satisfying $\mathrm{WP}(\mathbf{P}, \mathbf{R})$ then it is guaranteed to terminate in a state satisfying $\mathbf{R}$.

Given any statement $\mathbf{S}$ and any formula $\mathbf{R}$ whose free variables are all in the final state space for $\mathbf{S}$ and which defines a condition on the final states for $\mathbf{S}$, we define the *weakest precondition* $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ to be the weakest condition on the initial states for $\mathbf{S}$ such that if $\mathbf{S}$ is started in any state which satisfies $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ then it is guaranteed to terminate in a state which satisfies $\mathbf{R}$. By using an infinitary logic, it turns out that $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ has a simple definition for all kernel language programs $\mathbf{S}$ and all (infinitary logic) formulae $\mathbf{R}$:

$$
\begin{aligned}
\mathrm{WP}(\{\mathbf{P}\}, \mathbf{R}) &=_{\text{DF}} \mathbf{P} \wedge \mathbf{R} \\
\mathrm{WP}([\mathbf{Q}], \mathbf{R}) &=_{\text{DF}} \mathbf{Q} \Rightarrow \mathbf{R} \\
\mathrm{WP}(\textsf{\textbf{add}}(\boldsymbol{x}), \mathbf{R}) &=_{\text{DF}} \forall \boldsymbol{x}.\, \mathbf{R} \\
\mathrm{WP}(\textsf{\textbf{remove}}(\boldsymbol{x}), \mathbf{R}) &=_{\text{DF}} \mathbf{R} \\
\mathrm{WP}((\mathbf{S}_1;\ \mathbf{S}_2), \mathbf{R}) &=_{\text{DF}} \mathrm{WP}(\mathbf{S}_1, \mathrm{WP}(\mathbf{S}_2, \mathbf{R})) \\
\mathrm{WP}((\mathbf{S}_1 \sqcap \mathbf{S}_2), \mathbf{R}) &=_{\text{DF}} \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \wedge \mathrm{WP}(\mathbf{S}_2, \mathbf{R}) \\
\mathrm{WP}((\mu X.\mathbf{S}), \mathbf{R}) &=_{\text{DF}} \bigvee_{n<\omega} \mathrm{WP}((\mu X.\mathbf{S})^n, \mathbf{R})
\end{aligned}
$$

The statement $(\mu X.\mathbf{S})^n$ is called the *nth truncation* of $\mathbf{S}$ and is defined as follows:

$$(\mu X.\mathbf{S})^0 = \textsf{\textbf{abort}} = \{\textsf{\textbf{false}}\} \qquad \text{and} \qquad (\mu X.\mathbf{S})^{n+1} = \mathbf{S}[(\mu X.\mathbf{S})^n / X]$$

where the notation $\mathbf{S}[\mathbf{T}'/\mathbf{T}]$ represents the result of replacing every occurrence of $\mathbf{T}$ in $\mathbf{S}$ by $\mathbf{T}'$. (Read $\mathbf{S}[\mathbf{T}'/\mathbf{T}]$ as $\mathbf{S}$ *with* $\mathbf{T}'$ *instead of* $\mathbf{T}$).

Note that the weakest precondition for **remove** is identical to the postcondition: the effect of a **remove**($\boldsymbol{x}$) is to ensure that the variables in $\boldsymbol{x}$ do not appear in W (the final state space). All free variables in $\mathbf{R}$ must appear in W.

Technically, we define the trinary predicate $\mathbf{S} : V \to W$ on $\mathbf{S}$, V and W to be true precisely when V is a valid initial state space for $\mathbf{S}$ and W is the corresponding valid final state space. A program such as $(\textsf{\textbf{add}}(\langle x \rangle) \sqcap \textsf{\textbf{remove}}(\langle x \rangle))$ has *no* valid final state space for any initial state space: such a program is considered to be syntactically incorrect. The semantics of a program is defined in terms of the program plus a valid initial and final state space.

With the recursive statement we see the advantage of using infinitary logic: the weakest precondition for this statement is defined as a countably infinite disjunction of weakest preconditions of statements, each of which has a smaller depth of nesting of recursive constructs. So ultimately, the weakest precondition of a statement involving recursion is defined in terms of weakest preconditions of statements with no recursion.

For the specification statement $\boldsymbol{x} := \boldsymbol{x}'.\mathbf{Q}$ we have, by applying the definition from Section 3.3:

$$\begin{aligned}
&\mathrm{WP}(\boldsymbol{x} := \boldsymbol{x}'.\mathbf{Q}, \mathbf{R})\\
&\quad\iff \mathrm{WP}((\{\exists \boldsymbol{x}'.\, \mathbf{Q}\};\ (\textsf{\textbf{add}}(\boldsymbol{x}');\ ([\mathbf{Q}];\ (\textsf{\textbf{add}}(\boldsymbol{x});\ ([\boldsymbol{x} = \boldsymbol{x}'];\ \textsf{\textbf{remove}}(\boldsymbol{x}')))))), \mathbf{R})\\
&\quad\iff \exists \boldsymbol{x}'\mathbf{Q} \wedge \mathrm{WP}((\textsf{\textbf{add}}(\boldsymbol{x}');\ ([\mathbf{Q}];\ (\textsf{\textbf{add}}(\boldsymbol{x});\ ([\boldsymbol{x} = \boldsymbol{x}'];\ \textsf{\textbf{remove}}(\boldsymbol{x}'))))), \mathbf{R})\\
&\quad\iff \exists \boldsymbol{x}'\mathbf{Q} \wedge \forall \boldsymbol{x}'.\ \mathrm{WP}([\mathbf{Q}];\ (\textsf{\textbf{add}}(\boldsymbol{x});\ ([\boldsymbol{x} = \boldsymbol{x}'];\ \textsf{\textbf{remove}}(\boldsymbol{x}')))), \mathbf{R})\\
&\quad\iff \exists \boldsymbol{x}'\mathbf{Q} \wedge \forall \boldsymbol{x}'.\ (\mathbf{Q} \Rightarrow \mathrm{WP}(\textsf{\textbf{add}}(\boldsymbol{x});\ ([\boldsymbol{x} = \boldsymbol{x}'];\ \textsf{\textbf{remove}}(\boldsymbol{x}'))), \mathbf{R})\\
&\quad\iff \exists \boldsymbol{x}'\mathbf{Q} \wedge \forall \boldsymbol{x}'.\ (\mathbf{Q} \Rightarrow \forall \boldsymbol{x}.\, ([\boldsymbol{x} = \boldsymbol{x}'];\ \textsf{\textbf{remove}}(\boldsymbol{x}')), \mathbf{R})\\
&\quad\iff \exists \boldsymbol{x}'\mathbf{Q} \wedge \forall \boldsymbol{x}'.\ (\mathbf{Q} \Rightarrow \forall \boldsymbol{x}.\, (\boldsymbol{x} = \boldsymbol{x}' \Rightarrow \mathbf{R}))\\
&\quad\iff \exists \boldsymbol{x}'\mathbf{Q} \wedge \forall \boldsymbol{x}'.\ (\mathbf{Q} \Rightarrow \mathbf{R}[\boldsymbol{x}'/\boldsymbol{x}])
\end{aligned}$$

(recall that since the variables $\boldsymbol{x}$' have been removed, they cannot occur free in $\mathbf{R}$).

Morgan's specification statement [23] is written $\boldsymbol{x}\colon [\mathsf{Pre}, \mathsf{Post}]$, where $\boldsymbol{x}$ is a list of variables, and Pre and Post are formulae. If the initial state does not satisfy Pre then the statement aborts, otherwise it is guaranteed to terminate in a state which satisfies Post and which differs from the initial state only on variables in $\boldsymbol{x}$. In other words, for all initial states satisfying Pre it must terminate in a state which satisfies Post and may only change the values of variables in $\boldsymbol{x}$. It may be defined in WSL as:

$$\boldsymbol{x}\colon [\mathsf{Pre}, \mathsf{Post}] \ =_{\mathrm{DF}}\ \{\mathsf{Pre}\};\ \textsf{\textbf{add}}(\boldsymbol{x});\ [\mathsf{Post}]$$

The weakest precondition is therefore:

$$\mathrm{WP}(\boldsymbol{x}\colon [\mathsf{Pre}, \mathsf{Post}], \mathbf{R}) \iff \mathsf{Pre} \wedge \forall \boldsymbol{x}.\, (\mathsf{Post} \Rightarrow \mathbf{R})$$

The Hoare predicate (defining partial correctness): $\{\mathsf{Pre}\}\mathbf{S}\{\mathsf{Post}\}$ is true if whenever $\mathbf{S}$ terminates after starting in an initial state which satisfies Pre then the final state will satisfy Post. We can express this in terms of WP as the formula: $(\mathsf{Pre} \wedge (\mathrm{WP}(\mathbf{S}, \textsf{\textbf{true}})) \Rightarrow \mathrm{WP}(\mathbf{S}, \mathsf{Post})$.

For the **if** statement:

$$\begin{aligned}
&\mathrm{WP}(\textsf{\textbf{if}}\ \mathbf{B}\ \textsf{\textbf{then}}\ \mathbf{S}_1\ \textsf{\textbf{else}}\ \mathbf{S}_2\ \textsf{\textbf{fi}}, \mathbf{R})\\
&\quad\iff \mathrm{WP}((([\mathbf{B}];\ \mathbf{S}_1)\ \sqcap\ ([\neg\mathbf{B}];\ \mathbf{S}_2)), \mathbf{R})\\
&\quad\iff \mathrm{WP}(([\mathbf{B}];\ \mathbf{S}_1), \mathbf{R}) \wedge \mathrm{WP}(([\neg\mathbf{B}];\ \mathbf{S}_2), \mathbf{R})\\
&\quad\iff \mathrm{WP}([\mathbf{B}], \mathrm{WP}(\mathbf{S}_1, \mathbf{R})) \wedge \mathrm{WP}([\neg\mathbf{B}], \mathrm{WP}(\mathbf{S}_2, \mathbf{R}))\\
&\quad\iff (\mathbf{B} \Rightarrow \mathrm{WP}(\mathbf{S}_1, \mathbf{R})) \wedge (\neg\mathbf{B} \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R}))
\end{aligned}$$

Similarly, for the Dijkstra guarded command:

$$\begin{aligned}
&\mathrm{WP}(\textsf{\textbf{if}}\ \mathbf{B}_1 \ \rightarrow\ \mathbf{S}_1 \ \square\ \mathbf{B}_2 \ \rightarrow\ \mathbf{S}_2\ \textsf{\textbf{fi}}, \mathbf{R})\\
&\quad\iff (\mathbf{B}_1 \vee \mathbf{B}_2) \wedge (\mathbf{B}_1 \Rightarrow \mathrm{WP}(\mathbf{S}_1, \mathbf{R})) \wedge (\mathbf{B}_2 \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R}))
\end{aligned}$$

The motivation for considering weakest preconditions is that refinement and transformations can be characterised using weakest preconditions and consequently, the proof of correctness of a refinement or transformation can be carried out as a first order logic proof on weakest preconditions.

3.5 Proof-Theoretic Refinement

We can define a notion of refinement using weakest preconditions as follows: $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ if and only if the formula

$$\mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$$

can be proved for *every* formula $\mathbf{R}$. Back and von Wright [4] and Morgan [23,24] use a second order logic to carry out this proof. In a second order logic we can quantify over boolean predicates, so the formula to be proved is:

$$\forall \mathbf{R}.\, \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$$

This approach has a serious drawback: second order logic is *incomplete* which means that there is not necessarily a proof for every valid transformation. Back [2, 3] gets round this difficulty by extending the logic with a new predicate symbol to represent the postcondition and carrying out the proof in the extended first order logic.

However, it turns out that these exotic logics and extensions are not necessary because there are two simple postconditions which completely characterise the refinement relation. We can define a refinement relation using weakest preconditions on these two postconditions:

DEFINITION 3.5. Let $\boldsymbol{x}$ be a sequence of the variables in the final state space of $\mathbf{S}_1$ and $\mathbf{S}_2$ and let $\boldsymbol{x}'$ be a sequence of new variables the same length as $\boldsymbol{x}$. If the formulae $\mathrm{WP}(\mathbf{S}_1, \boldsymbol{x} \neq \boldsymbol{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \boldsymbol{x} \neq \boldsymbol{x}')$ and $\mathrm{WP}(\mathbf{S}_1, \mathsf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathsf{true})$ are provable from a given set Δ of *sentences* (formulae with no free variables), then we say that $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ and write: $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$. (Two sequences are equal when they have the same number of elements, and all corresponding elements are equal).

This definition of refinement (using weakest preconditions) is equivalent to the definition of refinement in terms of semantic functions (see [32] for the proof). As a result, we have two very different methods available for proving the correctness of a program transformation: and both methods are used in practice.

If both $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$ and $\Delta \vdash \mathbf{S}_2 \leq \mathbf{S}_1$ then we say that $\mathbf{S}_1$ and $\mathbf{S}_2$ are equivalent, and write $\Delta \vdash \mathbf{S}_1 \approx \mathbf{S}_2$. A *transformation* is any operation which takes a statement $\mathbf{S}_1$ and transforms it into an equivalent statement $\mathbf{S}_2$ (where Δ is the set of *applicability conditions* for the transformation).

An example of an "applicability condition" is a property of the function or relation symbols which a particular transformation depends on. For example, the statements $x := a \oplus b$ and $x := b \oplus a$ are equivalent when $\oplus$ is a commutative operation. We can write this transformation as:

$$\{\forall a, b.\, a \oplus b = b \oplus a\} \vdash x := a \oplus b \;\approx\; x := b \oplus a$$

3.6 Example Transformations

To see how we use weakest preconditions to prove the validity of transformations we will take a very simple example: reversing an **if** statement. To prove the transformation:

$$\Delta \vdash \textsf{\textbf{if}}\ \mathbf{B}\ \textsf{\textbf{then}}\ \mathbf{S}_1\ \textsf{\textbf{else}}\ \mathbf{S}_2\ \textsf{\textbf{fi}} \;\approx\; \textsf{\textbf{if}}\ \neg\mathbf{B}\ \textsf{\textbf{then}}\ \mathbf{S}_2\ \textsf{\textbf{else}}\ \mathbf{S}_1\ \textsf{\textbf{fi}}$$

we simply need to show that the corresponding weakest preconditions are equivalent:

$$\begin{array}{rl} & \mathrm{WP}(\textsf{if } \mathbf{B} \textsf{ then } \mathbf{S}_1 \textsf{ else } \mathbf{S}_2 \textsf{ fi}, \mathbf{R}) \\ \iff & \mathbf{B} \Rightarrow \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \ \wedge\ \neg\mathbf{B} \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R}) \\ \iff & (\neg\mathbf{B}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R}) \ \wedge\ \neg(\neg\mathbf{B}) \Rightarrow \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \\ = & \mathrm{WP}(\textsf{if } \neg\mathbf{B} \textsf{ then } \mathbf{S}_2 \textsf{ else } \mathbf{S}_1 \textsf{ fi}, \mathbf{R}) \end{array}$$

Another simple transformation is merging two assignments to the same variable:

$$\Delta \vdash x := e_1;\ x := e_2 \ \approx\ x := e_2[e_1/x]$$

The assignment $x := e$ is defined as:

$$\textsf{add}(\langle x' \rangle);\ [x' = e];\ \textsf{add}(\langle x \rangle);\ [x = x'];\ \textsf{remove}(\langle x' \rangle)$$

so the weakest precondition is:

$$\begin{aligned} \mathrm{WP}(x := e, \mathbf{R}) &= \mathrm{WP}(\textsf{add}(\langle x' \rangle);\ [x' = e], \forall x.\, (x = x' \Rightarrow \mathbf{R})) \\ &= \mathrm{WP}(\textsf{add}(\langle x' \rangle);\ [x' = e], \mathbf{R}[x'/x]) \\ &= \forall x'.\, (x' = e \Rightarrow \mathbf{R}[x'/x]) \\ &= \mathbf{R}[x'/x][e/x'] \\ &= \mathbf{R}[e/x] \end{aligned}$$

So to prove the transformation we simply calculate the weakest preconditions:

$$\begin{aligned} \mathrm{WP}(x := e_1;\ x := e_2, \mathbf{R}) &= \mathrm{WP}(x := e_1, \mathrm{WP}(x := e_2, \mathbf{R})) \\ &= \mathrm{WP}(x := e_1, \mathbf{R}[e_2/x]) \\ &= \mathbf{R}[e_2/x][e_1/x] \\ &= \mathbf{R}[(e_2[e_1/x])/x] \\ &= \mathrm{WP}(x := e_2[e_1/x], \mathbf{R}) \end{aligned}$$

Another simple transformation is Expand Forwards:

$$\begin{aligned} &\Delta \vdash \textsf{if } \mathbf{B}_1 \textsf{ then } \mathbf{S}_1 \ldots \textsf{ elsif } \mathbf{B}_n \textsf{ then } \mathbf{S}_n \textsf{ fi};\ \mathbf{S} \\ &\qquad\qquad \approx\ \textsf{if } \mathbf{B}_1 \textsf{ then } \mathbf{S}_1;\ \mathbf{S} \ldots \textsf{ elsif } \mathbf{B}_n \textsf{ then } \mathbf{S}_n;\ \mathbf{S} \textsf{ fi} \end{aligned}$$

For more complex transformations involving recursive constructs, we have a useful induction rule which is not limited to a single recursive procedure, but can be used on statements containing one or more recursive components (including nested recursion). For any statement $\mathbf{S}$, define $\mathbf{S}^n$ to be $\mathbf{S}$ with each recursive statement replaced by its nth truncation.

LEMMA 3.6. *The General Induction Rule for Recursion:* If $\mathbf{S}$ is any statement with bounded nondeterminacy, and $\mathbf{S}'$ is another statement such that $\Delta \vdash \mathbf{S}^n \ \leq\ \mathbf{S}'$ for all $n < \omega$, then $\Delta \vdash \mathbf{S} \ \leq\ \mathbf{S}'$.

Here, "bounded nondeterminacy" means that in each specification statement there is a finite number of possible values for the assigned variables. See [30] for the proof.

An example transformation which is proved using the generic induction rule is *loop merging*. If $\mathbf{S}$ is any statement and $\mathbf{B}_1$ and $\mathbf{B}_2$ are any formulae such that $\mathbf{B}_1 \Rightarrow \mathbf{B}_2$ then:

$$\Delta \vdash \textbf{while } \mathbf{B}_1 \textbf{ do S od}; \textbf{ while } \mathbf{B}_2 \textbf{ do S od} \approx \textbf{while } \mathbf{B}_2 \textbf{ do S od}$$

where the while loop **while B do S od** is defined in terms of a tail recursion ($\mu X.$ **if B then S**; X **fi**)

To prove loop merging it is sufficient to prove by induction that for each n there exists an m such that:

$$\Delta \vdash \textbf{while } \mathbf{B}_1 \textbf{ do S od}^n; \textbf{ while } \mathbf{B}_2 \textbf{ do S od}^n \leq \textbf{while } \mathbf{B}_2 \textbf{ do S od}^m$$

and for each m there exists an n_1 and n_2 such that:

$$\Delta \vdash \textbf{while } \mathbf{B}_2 \textbf{ do S od}^m \leq \textbf{while } \mathbf{B}_1 \textbf{ do S od}^{n_1}; \textbf{ while } \mathbf{B}_2 \textbf{ do S od}^{n_2}$$

(See [30] for the induction proofs). The result then follows from the general induction rule.

4. EXTENSIONS TO THE KERNEL LANGUAGE

The WSL language is built up in a set of layers, starting with the kernel language. The first level language includes specification statements, assignments, **if** statements, **while** and **for** loops, Dijkstra's guarded commands [14] and simple local variables.

The local variable clause **var** $\langle v := e \rangle$: **S end**, where v is a variable, e is an expression and **S** is a statement, is defined as follows:

$$\mathsf{v\_stack} := \langle v \rangle \mathbin{+\!\!+} \mathsf{v\_stack};\ v := e;\ \mathbf{S};\ v := \mathsf{v\_stack}[1];\ \mathsf{v\_stack} := \mathsf{v\_stack}[2..]$$

Here, the $+\!\!+$ operator concatenates two sequences. If $\mathsf{v\_stack}$ contains the sequence $\langle v_1, \dots, v_n \rangle$ then $\mathsf{v\_stack}[i]$ returns the ith element of the sequence, v_i, and $\mathsf{v\_stack}[i..]$ returns the subsequence $\langle v_i, \dots, v_n \rangle$.

In other words, we save the current value of v on a stack $\mathsf{v\_stack}$ (which is a new variable), assign the new value to the variable v, execute the body, then restore v by popping the value off the stack.

We use the notation **var** $\langle v := \perp \rangle$: **S end** to indicate that the local variable is assigned an arbitrary value. It is defined as follows:

$$\mathsf{v\_stack} := \langle v \rangle \mathbin{+\!\!+} \mathsf{v\_stack};\ \textbf{add}(\langle v \rangle);\ \mathbf{S};\ v := \mathsf{v\_stack}[1];\ \mathsf{v\_stack} := \mathsf{v\_stack}[2..]$$

The second level language adds **do** ... **od** loops, action systems and local variables.

A **do** ... **od** loop is an unbounded or "infinite" loop which can only be terminated by executing an **exit**(n) statement (where the n must be a simple integer, not a variable or expression). This statement terminates the n enclosing **do** ... **od** loop.

An action system is a set of parameterless procedures of the form:

(1) **actions** A_1 :
(2) $A_1 \equiv S_1$.
(3) ...
(4) $A_n \equiv S_n$.
(5) **endactions**

where A_1 is the starting action and within each $\mathbf{S}_i$ a statement of the form **call** A_j is a call to another procedure. The special statement **call** Z (where Z is not one of the A_i) causes the whole action system to terminate immediately, without executing any further statements. In particular, none of the "pending" action call returns will take place.

See [42] or [45] for the formal definition of **do** ... **od**, **exit**(n) and action systems in terms of the WSL kernel language.

These loops are very valuable for restructuring unstructured code *without* introducing extra "flag" variables, or duplicating blocks of code. In the worst case, code duplication causes an exponential growth in program size. This will *not* improve the understandability or maintainability of the code. Having these loops as part of the WSL language, together with an extensive set of transformations for manipulating these loops, is essential for the practical applications of the theory in assembler analysis and migration.

A *proper sequence* is any program such that every **exit**(n) is contained within at least n enclosing loops. If $\mathbf{S}$ contains an **exit**(n) within less than n loops, then this **exit** could cause termination of a loop enclosing $\mathbf{S}$. The set of terminal values of $\mathbf{S}$, denoted $\mathsf{TVs}(\mathbf{S})$ is the set of integers $n - d \geqslant 0$ such that there is an **exit**(n) within d nested loops in $\mathbf{S}$ which could cause termination of $\mathbf{S}$. $\mathsf{TVs}(\mathbf{S})$ also contains 0 if $\mathbf{S}$ could terminate normally (i.e. not via an **exit** statement).

For example, if $\mathbf{S}$ is the program:

(1) **do if** $x = 0$ **then exit**(3)
(2) **elsif** $x = 1$ **then exit**(2) **fi**;
(3) $x := x - 2$ **od**

Then $\mathsf{TVs}(\mathbf{S}) = \{1, 2\}$. Any proper sequence has $\mathsf{TVs}(\mathbf{S}) \subseteq \{0\}$.

We earlier remarked on the remarkable properties of the guard statement, in particular [**false**] is a valid refinement for any program or specification. This is because the set of final states is empty for every (proper) initial state. A program which may have an empty set of final states is called a *null program* and is inherently unimplementable in any programming language. So it is important to avoid inadvertantly introducing a null program as the result of a refinement process. Morgan [23] calls the program [**false**] a "miracle", after Dijkstra's "Law of Excluded Miracles" [14]:

$$\mathrm{WP}(\mathbf{S}, \mathsf{\textbf{false}}) = \mathsf{\textbf{false}}$$

Part of the motivation for our specification statement is to exclude null programs (Morgan leaves it to the programmer to ensure that null programs are not introduced by accident). WSL has been designed so that any WSL program which does not include guard statements will obey the Law of Excluded Miracles.

5. OPERATIONAL SEMANTICS

The correctness proofs of WSL transformations only look at the external behaviour of the programs. If we want to know which transformations preserve the actual sequence of internal operations, then it would appear that a new definition of the semantics of programs is required: one which defines the meaning of a program to be a function from the initial state to the possible sequences of internal states culminating in the final state of the program: in other words, an *operational semantics*. We would then need to attempt to re-prove the correctness of all the transformations under the new semantics, in order to find out which ones are still valid. But we would not have the benefit of the weakest precondition approach, and we would not be able to re-use any existing proofs. Xingyuan Zhang [47] has defined an operational semantics for WSL and started to prove the correctness of some transformations in terms of the operational semantics.

It turns out that this extra work is not essential: instead the operational semantics can be "encoded" in the denotational semantics. We add a new variable, seq, to the program which will be used to record the sequence of state changes: at each point where an assignment takes place we append the list of variable names and assigned values to seq. The operation of annotating a program, adding assignments to seq at the appropriate places can be defined recursively as an $\mathsf{annotate}$ function which maps a program to the corresponding annotated program:

$$\begin{aligned}
\mathsf{annotate}(\mathbf{S}_1;\ \mathbf{S}_2) &=_{\text{DF}} \mathsf{annotate}(\mathbf{S}_1);\ \mathsf{annotate}(\mathbf{S}_2)\\
\mathsf{annotate}(\textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}) &=_{\text{DF}} \textit{(1)}\ \textbf{if } \mathbf{B} \textbf{ then } \mathsf{annotate}(\mathbf{S}_1)\\
&\qquad \textit{(2)}\qquad \textbf{else } \mathsf{annotate}(\mathbf{S}_2) \textbf{ fi}\\
\mathsf{annotate}(v := e) &=_{\text{DF}} \mathsf{seq} := \mathsf{seq} +\!\!+ \langle\langle \text{"v"}, e\rangle\rangle;\ v := e
\end{aligned}$$

and so on for the other constructs.

Given a transformation which turns $\mathbf{S}_1$ into the equivalent program $\mathbf{S}_2$, if we want to prove that the transformation preserves operational semantics it is sufficient to prove that the annotated program $\mathsf{annotate}(\mathbf{S}_1)$ is equivalent to $\mathsf{annotate}(\mathbf{S}_2)$.

The "reverse **if**" transformation of Section 3.6 is an example of a transformation which preserves operational semantics, while "merge assignments" does not.

6. SLICING

Weiser [43] defined a program slice $\mathbf{S}$ as a *reduced, executable program* obtained from a program $\mathbf{P}$ by removing statements, such that $\mathbf{S}$ replicates part of the behaviour of $\mathbf{P}$.

6.1 Reduction

Recall that we will be defining a slice as a combination of a syntactic relation and a semantic relation. The syntactic relation we use is called *reduction*. This relation defines the result of replacing certain statements in a program by **skip** or **exit** statements. We define the relation $\mathbf{S}_1 \sqsubseteq \mathbf{S}_2$, read "$\mathbf{S}_1$ is a reduction of $\mathbf{S}_2$", on WSL programs as follows:

$$\mathbf{S} \sqsubseteq \mathbf{S} \quad \text{for any program } \mathbf{S}$$

$$\textbf{skip} \sqsubseteq \mathbf{S} \quad \text{for any proper sequence } \mathbf{S}$$

If $\mathbf{S}$ is not a proper sequence and $n > 0$ is the largest integer in $\mathsf{TVs}(\mathbf{S})$ then:

$$\textbf{exit}(n) \sqsubseteq \mathbf{S}$$

If $\mathbf{S}'_1 \sqsubseteq \mathbf{S}_1$ and $\mathbf{S}'_2 \sqsubseteq \mathbf{S}_2$ then:

$$\textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}'_1 \textbf{ else } \mathbf{S}'_2 \textbf{ fi} \ \sqsubseteq\ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}$$

If $\mathbf{S}' \sqsubseteq \mathbf{S}$ then:

$$\textbf{while } \mathrm{B} \textbf{ do } \mathrm{S}' \textbf{ od} \ \sqsubseteq \ \textbf{while } \mathrm{B} \textbf{ do } \mathrm{S} \textbf{ od}$$

$$\textbf{var } \langle v := e \rangle : \mathbf{S}' \textbf{ end} \ \sqsubseteq \ \textbf{var } \langle v := e \rangle : \mathbf{S} \textbf{ end}$$

$$\textbf{var } \langle v := \perp \rangle : \mathbf{S}' \textbf{ end} \ \sqsubseteq \ \textbf{var } \langle v := e \rangle : \mathbf{S} \textbf{ end}$$

This last case will be used when the variable v is used in $\mathbf{S}$, but the initial value e is not used.

If $\mathbf{S}'_i \sqsubseteq \mathbf{S}_i$ for $1 \leqslant i \leqslant n$ then:

$$\mathbf{S}'_1;\ \mathbf{S}'_2;\ \ldots;\ \mathbf{S}'_n \ \sqsubseteq \ \mathbf{S}_1;\ \mathbf{S}_2;\ \ldots;\ \mathbf{S}_n$$

Note that the reduction relation does not allow actual deletion of statements: only replacing a statement by a **skip**. This makes it easier to match up the original program with the reduced version: the position of each statement in the reduced program is the same as the corresponding statement in the original program. Deleting the extra **skip**s is a trivial step.

In effect, we have expanded Weiser's "reduction" process into a two stage process: reduction (of statements to either **skip** or **exit** as appropriate), followed by deletion (of redundant **skip**s and loops).

Three important properties of the reduction relation are:

LEMMA 6.1. Transitivity: If $\mathbf{S}_1 \sqsubseteq \mathbf{S}_2$ and $\mathbf{S}_2 \sqsubseteq \mathbf{S}_3$ then $\mathbf{S}_1 \sqsubseteq \mathbf{S}_3$.

LEMMA 6.2. Antisymmetry: If $\mathbf{S}_1 \sqsubseteq \mathbf{S}_2$ and $\mathbf{S}_2 \sqsubseteq \mathbf{S}_1$ then $\mathbf{S}_1 = \mathbf{S}_2$.

LEMMA 6.3. The *Replacement Property*: If any component of a program is replaced by a reduction, then the result is a reduction of the whole program.

6.2 Semi-Refinement

In this subsection we will discuss the selection of a suitable semantic relation for the definition of slicing.

Initially we will consider the special case where the slicing point s is the end point of the program, but we will generalise the variable v to a set X of variables. (As we will see in Section 6.11, slicing at a point or points within the program does not introduce any further complications.) If X does not contain all the variables in the final state space of the program, then the sliced program will *not* be equivalent to the original program. However, consider the set $W \setminus X$, where W is the final state space. These are the variables whose values we are *not* interested in. By removing these variables from the final state space we can get a program which is equivalent to the sliced program. If a program $\mathbf{S}$ maps state spaces V to W, then the effect of slicing $\mathbf{S}$ at its end point on the variables in X is to generate a program equivalent to $\mathbf{S}$; **remove**$(W \setminus X)$.

Binkley et al. [8] define a "slice" as a combination of a syntactic ordering and a semantic equivalence relation, which can be *any* equivalence relation on a projection of the program semantics. In WSL terms, this suggests defining a slice of $\mathbf{S}$ on X to be any program $\mathbf{S}' \sqsubseteq \mathbf{S}$, such that:

$$\Delta \vdash \mathbf{S}';\ \textbf{remove}(W \setminus X) \ \approx \ \mathbf{S};\ \textbf{remove}(W \setminus X)$$

However, the requirement that the slice be strictly equivalent to the original program is too strict in some cases. Consider the program:

$$\mathbf{S};\ x := 0$$

where **S** does not contain any assignments to x. If we are slicing on x then we would like to reduce the whole of **S** to a **skip**: but the two programs

$$\textbf{skip};\ x := 0;\ \textbf{remove}(W \setminus \{x\}) \qquad \text{and} \qquad \mathbf{S};\ x := 0;\ \textbf{remove}(W \setminus \{x\})$$

are only equivalent provided that **S** always terminates. But most slicing researchers see no difficulty in slicing away potentially non-terminating code.

So, WSL equivalence is not suitable for defining program slicing. In fact, there is *no* semantic equivalence relation which is suitable for defining a useful program slice! Consider the two programs **abort** and **skip**. Any possible semantic relation must either treat **abort** as equivalent to **skip**, or must treat **abort** as not equivalent to **skip**.

(1) Suppose **abort** is not equivalent to **skip**. Then the slicing relation will not allow deletion of non-terminating, or potentially non-terminating, code. So this is not suitable;

(2) On the other hand, suppose **abort** is equivalent to **skip**. Then the slicing relation will allow deletion of statements which turn a terminating program into a non-terminating program. For example, in the program:

$$x := 0;\ x := 1;\ \textbf{while}\ x = 0\ \textbf{do skip od}$$

we could delete the statement $x := 1$ to give a syntactically smaller, semantically "equivalent" but non-terminating program. Few slicing researchers are happy to allow a non-terminating program as a valid slice of a terminating program!

Another semantic relation which has been proposed [36] is to allow any *refinement* of a program, which is also a reduction, as a valid slice. This would allow slicing away nonterminating code, since **skip** is a refinement of any nonterminating program. But such a definition of slicing is counter-intuitive, in the sense that slicing is intuitively an *abstraction* operation (an operation which throws away information), while refinement is the opposite of abstraction. A more important consideration is that we would like to be able to analyse the sliced program and derive facts about the original program (with the proviso that the original program might not terminate in cases where the slice does). If the sliced program assigns a particular value to a variable in the slice, then we would like to deduce that the original program assigns the *same* value to the variable. But with the refinement definition of a slice, the fact that the slice sets x to 1, say, tells us only that 1 is one of the *possible* values given to x by the original program.

These considerations led to the development of the concept of a *semi-refinement*:

DEFINITION 6.4. A *semi-refinement* of **S** is any program $\mathbf{S}'$ such that

$$\Delta \vdash \mathbf{S}\ \approx\ \{\mathrm{WP}(\mathbf{S}, \textbf{true})\};\ \mathbf{S}'$$

It is denoted $\Delta \vdash \mathbf{S}\ \preccurlyeq\ \mathbf{S}'$.

A semi-refinement can equivalently be defined in terms of the weakest preconditions:

$$\begin{array}{lll}\Delta \vdash \mathbf{S} \preccurlyeq \mathbf{S}' & \text{iff} & \Delta \vdash \mathrm{WP}(\mathbf{S}, \boldsymbol{x} \neq \boldsymbol{x}') \Leftrightarrow (\mathrm{WP}(\mathbf{S}, \textsf{true}) \wedge \mathrm{WP}(\mathbf{S}', \boldsymbol{x} \neq \boldsymbol{x}')) \\ & \text{and} & \Delta \vdash \mathrm{WP}(\mathbf{S}, \textsf{true}) \Leftrightarrow (\mathrm{WP}(\mathbf{S}, \textsf{true}) \wedge \mathrm{WP}(\mathbf{S}', \textsf{true}))\end{array}$$

The assertion in the semi-refinement relation shows that program equivalence is only required where the original program terminates. So we define a slice of $\mathbf{S}$ on X to be any reduction of $\mathbf{S}$ which is also a semi-refinement:

Definition 6.5. A *Syntactic Slice* of $\mathbf{S}$ on a set X of variables is any program $\mathbf{S}'$ with the same initial and final state spaces such that $\mathbf{S}' \sqsubseteq \mathbf{S}$ and

$$\Delta \vdash \mathbf{S};\ \textsf{remove}(W \setminus X) \preccurlyeq \mathbf{S}';\ \textsf{remove}(W \setminus X)$$

where W is the final state space for $\mathbf{S}$ and $\mathbf{S}'$.

The assertion $\{\mathrm{WP}(\mathbf{S}, \textsf{true})\}$ is a **skip** whenever $\mathbf{S}$ is guaranteed to terminate and an **abort** whenever $\mathbf{S}$ aborts. So in the case when $\mathbf{S}$ aborts, $\mathbf{S}'$ can be anything: in particular, setting $\mathbf{S}'$ to **skip** will trivially satisfy $\mathbf{S}' \sqsubseteq \mathbf{S}$. So this definition allows us to slice away nonterminating code.

If $\Delta \vdash \mathbf{S} \preccurlyeq \mathbf{S}'$ then $\Delta \vdash \mathbf{S} \leq \mathbf{S}'$ (since deleting an assertion is a valid refinement), but the converse does not hold. So the relationship lies somewhere between a refinement and an equivalence.

Like refinement and reduction, semi-refinement also satisfies the *replacement property*: if any component of a program is replaced by a semi-refinement then the result is a semi-refinement of the whole program.

Semi-refinement also allows deletion of any assertions:

Lemma 6.6. $\{\mathbf{Q}\} \preccurlyeq \textsf{skip}$

Proof: $\mathrm{WP}(\{\mathbf{Q}\}, \textsf{true}) = \mathbf{Q} \quad \text{so} \quad \{\mathbf{Q}\} \approx \{\mathrm{WP}(\{\mathbf{Q}\}, \textsf{true})\};\ \textsf{skip}$ ■

Semi-refinement has a counterpart to the general induction rule for recursion:

Lemma 6.7. *The General Induction Rule for Semi-Refinement:* If $\mathbf{S}$ is any statement with bounded nondeterminacy, and $\mathbf{S}'$ is another statement such that $\Delta \vdash \mathbf{S}^n \preccurlyeq \mathbf{S}'$ for all $n < \omega$, then $\Delta \vdash \mathbf{S} \preccurlyeq \mathbf{S}'$.

With this definition of slicing, a slice can be computed purely by applying program transformation operations which duplicate and move the **remove** statements through the program and then use the **remove** statements to transform components of the program to **skip** statements.

As an example of this process, consider the following program:

$x := y + 1;$
if $x > 0$ **then** $x := 2;\ z := y$ **fi**;
remove$(\{x, y\})$

Expand the **if** statement forwards:

$x := y + 1;$
if $x > 0$ **then** $x := 2;\ z := y;$ **remove**$(\{x, y\})$ **else remove**$(\{x, y\})$ **fi**

The **then** clause can be transformed to:

$x := 2$; **remove**$(\{x\})$; $z := y$; **remove**$(\{x, y\})$

which is equivalent to:

skip; **remove**$(\{x\})$; $z := y$; **remove**$(\{x, y\})$

We now push the **remove** statements forwards again and merge them to give:

$x := y + 1$;
if $x > 0$ **then skip**; $z := y$ **fi**;
remove$(\{x, y\})$

This is clearly a reduction of the original program, so it is a valid slice.

What we have done is apply the following steps:-

(1) Duplicate the **remove** statements;
(2) Pull each **remove** backwards through the program as far as possible (moving it inside compound statements);
(3) Use each **remove** to simplify preceding statements;
(4) Finally, push the **remove**s forwards and combine them into a single statement at the end of the program.

This technique is developed into a simple slicing algorithm in the next section.

6.3 A Simple Slicing Algorithm

In this section we will develop a simple slicing algorithm for a small subset of WSL which includes the following:-

(1) Simple assignments;
(2) Assertions;
(3) Statement sequences;
(4) **if** statements;
(5) **while** loops;
(6) Local variables;

Note that **skip** and **abort** are defined as the assertions $\{$**true**$\}$ and $\{$**false**$\}$ respectively.

The basic approach is to use program transformations and semi-refinements to duplicate and "pull" the **remove** statement backwards through the program $\mathbf{S}$ to generate the sliced program $\mathbf{S}'$, and then "push" the **remove** statement forwards through $\mathbf{S}'$ to the end of the program again.

For conciseness, we write (X) for **remove**(X) in what follows.

The following set of lemmas show how to slice each type of statement by pushing the **remove** statement backwards through the structure, reducing the statement wherever possible, and then pulling it forwards through the reduced statement. These lemmas will be used as the basis for a simple slicing algorithm.

Consider the simple assignment $x := e$. If $x \in X$ then we need to keep the assignment since the final value of x is required. Before the assignment, we don't need the value of x (unless x appears in the expression e), but we do need all the variables which appear in e. Conversely, if $x \notin X$ then x is not in the final state space, and the assignment is equivalent to a **skip**.

So we have:

LEMMA 6.8. *Assignment.* If $x \in X$ then:

$$\Delta \vdash x := e;\ (W \setminus X)\ \preccurlyeq\ (W \setminus ((X \setminus \{x\}) \cup \mathsf{vars}(e)));\ x := e;\ (W \setminus X)$$

If $x \notin X$ then:

$$\Delta \vdash x := e;\ (W \setminus X)\ \preccurlyeq\ (W \setminus X);\ \textsf{skip};\ (W \setminus X)$$

LEMMA 6.9. *Assertion.* The semi-refinement relation allows any assertion to be replaced by a **skip**:

$$\Delta \vdash \{\mathbf{Q}\};\ (W \setminus X)\ \preccurlyeq\ (W \setminus X);\ \textsf{skip};\ (W \setminus X)$$

LEMMA 6.10. *Statement Sequence.* Suppose:

$$\Delta \vdash \mathbf{S}_2;\ (W \setminus X)\ \preccurlyeq\ (W \setminus X');\ \mathbf{S}_2';\ (W \setminus X)\ \approx\ \mathbf{S}_2';\ (W \setminus X)$$

and

$$\Delta \vdash \mathbf{S}_1;\ (W \setminus X')\ \preccurlyeq\ (W \setminus X'');\ \mathbf{S}_1';\ (W \setminus X')\ \approx\ \mathbf{S}_1';\ (W \setminus X')$$

Then:

$$\begin{aligned}\Delta \vdash \mathbf{S}_1;\ \mathbf{S}_2;\ (W \setminus X)\ &\preccurlyeq\ (W \setminus X'');\ \mathbf{S}_1';\ (W \setminus X');\ \mathbf{S}_2';\ (W \setminus X)\\ &\approx\ \mathbf{S}_1';\ \mathbf{S}_2';\ (W \setminus X)\end{aligned}$$

LEMMA 6.11. *Conditional statement.* Suppose $\mathbf{S}_1;\ (W \setminus X)\ \preccurlyeq\ (W \setminus X_i);\ \mathbf{S}_1'$ and $\mathbf{S}_2;\ (W \setminus X)\ \preccurlyeq\ (W \setminus X_i);\ \mathbf{S}_2'$ Then:

$$\begin{aligned}&\Delta \vdash \textsf{if } \mathbf{B}_1 \textsf{ then } \mathbf{S}_1 \textsf{ else } \mathbf{S}_2 \textsf{ fi};\ (W \setminus X)\\ &\quad\preccurlyeq\ \textsf{if } \mathbf{B}_1 \textsf{ then } \mathbf{S}_1;\ (W \setminus X) \textsf{ else } \mathbf{S}_2;\ (W \setminus X) \textsf{ fi};\ (W \setminus X)\end{aligned}$$

by Expand Forward

$$\preccurlyeq\ \textsf{if } \mathbf{B}_1 \textsf{ then } (W \setminus X_1);\ \mathbf{S}_1' \textsf{ else } (W \setminus X_2);\ \mathbf{S}_2' \textsf{ fi};\ (W \setminus X)$$

by the premise

$$\preccurlyeq\ (W \setminus X');\ \textsf{if } \mathbf{B}_1 \textsf{ then } (W \setminus X_1);\ \mathbf{S}_1' \textsf{ else } (W \setminus X_2);\ \mathbf{S}_2' \textsf{ fi};\ (W \setminus X)$$

where $X' = X_1 \cup \mathsf{vars}(\mathbf{B}_1) \cup X_2 \cup \mathsf{vars}(\mathbf{B}_2)$

This lemma can be trivially extended to multi-way **if** statements.

LEMMA 6.12. *While loop.* Let $\mathbf{S}'$ and X' be such that $X \subseteq X'$ and $\mathsf{vars}(\mathbf{B}) \subseteq X'$ and:

$$\Delta \vdash \mathbf{S};\ (W \setminus X')\ \preccurlyeq\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X')$$

Then:

$$\begin{aligned}&\Delta \vdash \textsf{while } \mathbf{B} \textsf{ do } \mathbf{S} \textsf{ od};\ (W \setminus X)\\ &\qquad\preccurlyeq\ (W \setminus X');\ \textsf{while } \mathbf{B} \textsf{ do } (W \setminus X');\ \mathbf{S}';\ (W \setminus X') \textsf{ od};\ (W \setminus X)\end{aligned}$$

Proof: The proof uses the induction rule for iteration (Lemma 3.6). We will prove by induction on n that:

$$\begin{aligned}&\Delta \vdash \textsf{while } \mathbf{B} \textsf{ do } \mathbf{S} \textsf{ od}^n;\ (W \setminus X)\\ &\qquad\preccurlyeq\ (W \setminus X');\ \textsf{while } \mathbf{B} \textsf{ do } (W \setminus X');\ \mathbf{S}';\ (W \setminus X') \textsf{ od};\ (W \setminus X)\end{aligned}$$

The result then follows from the induction rule for semi-refinement (Lemma 6.7).

The base case is trivial, since **while B do S od**^{0} is **abort** (by definition). For the induction step, we have:

$$\begin{aligned}
&\textbf{while B do S od}^{n+1};\ (W \setminus X)\\
&\quad = \ \textbf{if B then S};\ \textbf{while B do S od}^{n}\ \textbf{fi};\ (W \setminus X)\\
&\quad \preccurlyeq \ \textbf{if B then S};\ \textbf{while B do S od}^{n};\ (W \setminus X)\ \textbf{fi};\ (W \setminus X)
\end{aligned}$$

by duplicating and absorbing the **remove** statement.

$$\begin{aligned}
\preccurlyeq\ &\textbf{if B then S};\ (W \setminus X');\\
&\qquad \textbf{while B do}\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X')\ \textbf{od}^{n};\ (W \setminus X)\ \textbf{fi};\\
&(W \setminus X)
\end{aligned}$$

by the induction hypothesis.

$$\begin{aligned}
\preccurlyeq\ &\textbf{if B then}\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X');\\
&\qquad \textbf{while B do}\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X')\ \textbf{od}^{n};\ (W \setminus X)\ \textbf{fi};\\
&(W \setminus X)
\end{aligned}$$

by the premise.

$$\begin{aligned}
\preccurlyeq\ &(W \setminus X');\\
&\textbf{if B then}\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X');\\
&\qquad \textbf{while B do}\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X')\ \textbf{od}^{n};\ (W \setminus X)\ \textbf{fi};\\
&(W \setminus X)
\end{aligned}$$

by taking out the **remove** statement.

$$= \ (W \setminus X');\ \textbf{while B do}\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X')\ \textbf{od}^{n+1};\ (W \setminus X)$$

This proves the induction step. So by induction, the result holds for every truncation of the loop, and by Lemma 6.7 it therefore holds for the whole loop.

For the local variable statement **var** $\langle v := e \rangle$: **S end**, suppose that X' and $\mathbf{S}' \sqsubseteq \mathbf{S}$ are such that:

$$\Delta \vdash \mathbf{S};\ (W \setminus X)\ \preccurlyeq\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X)$$

There are two cases to consider:

(1) If $v \in X'$ then we need the value that v is initialised to at the top of the block. We therefore also need all the variables in e. Finally, if $v \in X$ then we need the (global) v, while if $v \notin X$ then we don't need to add v to X'. Both of these cases are catered for by adding $X \cap \{v\}$ to X';

(2) If $v \notin X'$ then we do not need the initial value of v. So the reduced **var** block can initialise v to $\perp$ to avoid unnecessary references to any variables in e.

So we have:

LEMMA 6.13. *Local variable.* Let $\mathbf{S}' \sqsubseteq \mathbf{S}$ and:

$$\Delta \vdash \mathbf{S};\ (W \setminus X)\ \preccurlyeq\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X)$$

Then:

$$\begin{aligned}\Delta \vdash \textsf{var}\ \langle v := e\rangle : \mathbf{S}\ \textsf{end};\ (W \setminus X) \\ \preccurlyeq\ & (W \setminus ((X' \setminus \{v\}) \cup \mathsf{vars}(e) \cup (X \cap \{v\})); \\ & \textsf{var}\ \langle v := e\rangle : \mathbf{S}'\ \textsf{end}; \\ & (W \setminus X)\end{aligned}$$

if $v \in X'$ and

$$\begin{aligned}\Delta \vdash \textsf{var}\ \langle v := e\rangle : \mathbf{S}\ \textsf{end};\ (W \setminus X) \\ \preccurlyeq\ & (W \setminus ((X' \setminus \{v\}) \cup (X \cap \{v\})); \\ & \textsf{var}\ \langle v := \perp\rangle : \mathbf{S}'\ \textsf{end}; \\ & (W \setminus X)\end{aligned}$$

if $v \notin X'$.

Lemma 6.14. *Reduction of a whole statement.* Finally, there is an optimisation for the case where all the variables assigned in $\mathbf{S}$ are removed by the **remove** statement, i.e. when $\mathsf{assigned}(\mathbf{S}) \cap X = \varnothing$. In this case:

$$\Delta \vdash \mathbf{S};\ (W \setminus X)\ \preccurlyeq\ (W \setminus X);\ \textsf{skip};\ (W \setminus X)$$

The proof is by induction on the structure of $\mathbf{S}$. For the **while** loop case we use the induction rule for iteration (the base case for this induction uses the fact that $\Delta \vdash \textsf{abort}\ \preccurlyeq\ \textsf{skip}$).

These lemmas can be put together to derive a simple slicing algorithm for this subset of WSL. We define a function @Slice which takes a WSL program plus the set X of variables and returns the sliced program plus the modified variable set. So if:

$$\text{@}\mathsf{Slice}(\mathbf{S}, X) = \langle \mathbf{S}', X'\rangle$$

then:

$$\Delta \vdash \mathbf{S};\ (W \setminus X)\ \preccurlyeq\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X)$$

The only difficult case is the while loop **while B do S od**: Lemma 6.12 gives no indication of how to compute X'. Consider the following program:

$y := x_0$;
while $i \neq 0$ **do**
  $y := x_1$;
  $x_1 := x_2$;
  $x_2 := x_3$;
  $i := i - 1$ **od**;
$(W \setminus \{y\})$

Here, we are interested in the final value of y. If the loop is skipped ($i = 0$ initially) then y depends on x_0. If the loop body is executed once ($i = 1$), then y depends on x_1. If the loop is executed twice, the y depends on x_2. Finally, if the loop is executed three or more times, then y depends on x_3. If we have no information on the number of iterations, then we must assume that y depends on all of $\{x_0, x_1, x_2, x_3\}$ the slice must contain the whole program and the **remove** statement inserted in front of the program is:

$(W \setminus \{i, x_0, x_1, x_2, x_3\})$;
$y := x_0$;
while $i \neq 0$ **do**
  $y := x_1$;
  $x_1 := x_2$;
  $x_2 := x_3$;
  $i := i - 1$ **od**;
$(W \setminus \{y\})$

A simple solution is to repeatedly call @Slice on the body of the loop, taking the output set of variables and adding the previous X' set plus $\mathsf{vars}(\mathbf{B})$ to create the new X' until the result converges to a stable value. More formally, we define the sequence of sets X_i where $X_0 = X$ and for each i, let:

$$\langle \mathbf{S}'_i, X'_i \rangle = \mathsf{@Slice}(\mathbf{S}, X_i)$$

so that:

$$\Delta \vdash \mathbf{S};\ (W \setminus X_i)\ \preccurlyeq\ (W \setminus X'_i);\ \mathbf{S}'_i;\ (W \setminus X_i)$$

Now let:

$$X_{i+1} = \mathsf{vars}(\mathbf{B}) \cup X_i \cup X'_i$$

The sequence X_i is increasing ($X_i \subseteq X_{i+1}$ for all i) and bounded above by the finite set W, so the sequence must converge in a finite number of steps: i.e. there is an X_n such that $X_m = X_n$ for all $m \geqslant n$. (Once we have $X_{n+1} = X_n$ then all subsequent X_m values will be identical). The result of $\mathsf{@Slice}(\textbf{while B do S od}, X)$ is defined as $\langle \textbf{while B do S}' \textbf{ od}, X' \rangle$ unless Lemma 6.14 applies.

Putting all these results together we can implement a simple slicing operation as a FermaT transformation. The @Slice function at the heart of the transformation is described in the following pseudocode. For a statement $\mathbf{S}$ and set of variables X, $\mathsf{@Slice}(\mathbf{S}, X)$ returns a list $\langle \mathbf{S}', X' \rangle$ such that $\Delta \vdash \mathbf{S};\ (W \setminus X)\ \preccurlyeq\ (W \setminus X');\ \mathbf{S}';\ (W \setminus X)$

In the following program, the variable R contains the value returned by the function on the last line:

funct $\mathsf{@Slice}(I, X)\ \equiv$
  var $\langle R := \langle\rangle, \mathbf{S}' := \langle\rangle, X' := \langle\rangle \rangle$:
    if I is a statement sequence $\mathbf{S}_1;\ \ldots;\ \mathbf{S}_n$
      then for $\mathbf{S} \in \langle \mathbf{S}_n, \ldots, \mathbf{S}_1 \rangle$ **do**
          $R := \mathsf{@Slice}(\mathbf{S}, X)$;
          $\mathbf{S}' := \langle R[1] \rangle +\!\!+ \mathbf{S}'$;
          $X := R[2]$ **od**;
        $R := \langle \mathbf{S}', X \rangle$
    elsif I is **abort**
        then $R := \langle I, \langle\rangle \rangle$
    elsif No variable assigned in I is in X
        then $R := \langle \textbf{skip}, X \rangle$

elsif I is an assignment statement
  then $R := \langle I, (X \setminus \mathsf{assigned}(I)) \cup \mathsf{used}(I)\rangle$
elsif I is **if** $\mathbf{B}$ **then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**
  then var $\langle R_1 := @\mathsf{Slice}(\mathbf{S}_1, X),$
      $R_2 := @\mathsf{Slice}(\mathbf{S}_2, X)\rangle$
    $R := \langle$**if** $\mathbf{B}$ **then** $R_1[1]$ **else** $R_1[2]$ **fi**,
      $R_1[2] \cup R_2[2] \cup \mathsf{vars}(\mathbf{B})\rangle$ **end**
elsif I is **while** $\mathbf{B}$ **do** $\mathbf{S}$ **od**
  then var $\langle X' := X\rangle$:
    do $R := @\mathsf{Slice}(\mathbf{S}, X')$;
      $R[2] := R[2] \cup X' \cup \mathsf{vars}(\mathbf{B})$;
      if $R[2] = X'$ **then exit**(1) **fi**;
      $X' := R[2]$ **od end**;
    $R := \langle$**while** $\mathbf{B}$ **do** $R[1]$ **od**, $R[2]\rangle$
elsif I is **var** $\langle v := e\rangle$: $\mathbf{S}$ **end**
  then var $\langle X' := \langle\rangle\rangle$:
    $R := @\mathsf{Slice}(\mathbf{S}, X \setminus \{v\})$;
    $X' := (R[2] \setminus \{v\}) \cup (\{v\} \cap X)$;
    if $v \in R[2]$
      then $R := \langle$**var** $\langle v := e\rangle : R[1]$ **end**, $X' \cup \mathsf{vars}(e)\rangle$
      else $R := \langle$**var** $\langle v := \perp\rangle : R[1]$ **end**, $X'\rangle$ **fi end fi**;
(R).

This is essentially how the Simple_Slice transformation is implemented in FermaT, see the Appendix for the actual $\mathcal{META}$WSL source code (which is not much bigger that the pseudocode above). The following examples show the results produced by FermaT.

A simple example is the following program:

$y := z$;
if $y = 1$ **then** $x := 1$
elsif $y = 2$ **then** $\mathsf{x1} := 2$
  else $z := 99$ **fi**

Slicing on the final value of z we get:

$y := z$;
if $y = 1$ **then skip**
elsif $y = 2$ **then skip**
  else $z := 99$ **fi**

Another example:

$y := \mathsf{x0}$;
while $i \neq 0$ **do**
  $y := \mathsf{x1}$;
  $\mathsf{x1} := \mathsf{x2}$;
  $\mathsf{x2} := \mathsf{x3}$;
  $i := i - 1$ **od**

Slicing on $\mathsf{x1}$ gives:

skip;
while $i \neq 0$ **do**
  skip;
  x1 := x2;
  x2 := x3;
  $i := i - 1$ **od**

The next few subsections explore some of the advantages and applications of the formal definition of slicing in terms of WSL transformation theory.

6.4 Slicing Unstructured Programs

Note that we can apply arbitrary transformation in the process of slicing, provided that the final program satisfies all the conditions of Definition 6.5 (in particular, the slice is a reduction of the original program: in other words, it can be constructed from the original program by deleting replacing statements by **skip**s or **exit**s). So we can implement a slicing algorithm as the sequence:

$$\mathsf{transform} \;\rightarrow\; \mathsf{reduce} \;\rightarrow\; \mathsf{transform}$$

provided that the reduction step is also a semi-refinement and the final transformation step "undoes" the effect of the initial transformation. This step is facilitated by the fact that the reduction relation preserves the positions of sub-components in the program. In practice, the final transform step is implemented by tracking the movement of components in the initial transform step, noting which components are reduced in the reduce step and replacing these by **skip**s directly in the original program.

This suggests that our algorithm for slicing structured WSL can easily be extended to unstructured code: first restructure the program to use only **if** statements and **while** loops (for example, using Bohm and Jacopini's algorithm [10]), then slice the structured program, then determine which simple statements have been reduced, and apply the corresponding reduction to the original program (or equivalently, undo the restructuring transformation).

The simplest restructuring algorithm converts the whole program to a single **while** loop whose body is a multi-way conditional controlled by a single variable next which stores the number of the next block to be executed:

while next $\neq 0$ **do**
  if next $= 1$ **then** $\mathbf{S}_1$; **if** $\mathbf{B}_1$ **then** next := n_{11} **else** next := n_{12} **fi**
  ...
  elsif next $= i$ **then** $\mathbf{S}_i$; **if** $\mathbf{B}_i$ **then** next := n_{i1} **else** next := n_{i2} **fi**
  ... **fi od**

At the end of each branch of the conditional, next is conditionally or unconditionally assigned an integer value representing transfer of control to that block. Block i will be followed by block n_{i1} or n_{i2} depending in the result of the condition $\mathbf{B}_i$.

If such a program is sliced on any variable assigned anywhere in the program, then next will have to be included in the slice (since every statement is control dependent on next). All the variables in all of the $\mathbf{B}_i$ conditions are control variables for next (because next is control dependent on all these variables), so these variables will be included in the slice. Then all statements which assign to any variables that affect any $\mathbf{B}_i$ will also be included in the slice. This is likely to be most of the program.

FermaT's solution is to *destructure* the program to an action system in "basic blocks" format. To slice the action system, FermaT computes the Static Single Assignment (SSA) form of the program, and the control dependencies of each basic block using Bilardi and Pingali's algorithms [7,25]. FermaT tracks control and data dependencies to determine which statements can be deleted from the blocks. Tracking data dependencies is trivial when the program is in SSA form. FermaT links each basic block to the corresponding statement in the original program, so it can determine which statements from the original program have been deleted (in effect, this will "undo" the destructuring step). This algorithm is implemented as the Syntactic_Slice transformation in FermaT.

6.5 Generalisations of Statement Deletion

Some generalisations of the reduction relation will allow more precise slices, possibly at the expense of making it more difficult to track the connection between the original program and its slice.

For example, the statement **if B then S else S fi** is equivalent to **S**, so deleting the structure *around* **S** will remove dependencies on the variables in **B**. Programmers do not usually write **if** statements with identical branches, but such a statement might result from deleting other statements in the branches of the **if** statement. For example:

(1) $y := y + 1;$
(2) **if** $y > 0$
(3) **then** $z := 3;$
(4) $x := 2;$
(5) $y := 0$
(6) **else** $z := 4;$
(7) $x := 2;$
(8) $y := 1$ **fi**

Slicing on the final value of x using Simple_Slice gives:

(1) $y := y + 1;$
(2) **if** $y > 0$
(3) **then** $x := 2$
(4) **else** $x := 2$ **fi**

The assignment to y is not needed in the slice, but the Simple_Slice transformation sees the reference to y in the condition of the **if**, and assumes that the statement is needed. See Section 6.9 for a different approach to slicing this example.

Extending the reduction relation to allow:

$$\mathbf{S} \sqsubseteq \textbf{if B then S else S fi}$$

would allow the **if** statement to be sliced to $x := 2$, which in turn would allow Simple_Slice to delete the assignment to y giving $x := 2$ as the sliced program.

Another example is deleting an action that consists of a single call to another action.

On the other hand, there are some statements that we probably do *not* want to delete even though they have no effect on the variables we are slicing on: comments are statements in WSL and we will usually want to keep all the comments except for those inside deleted structures. The FermaT transformation system treats comments as a special case.

6.6 Interprocedural Slicing

For interprocedural slicing (see Section 7) we allow deletion of unused parameters in procedures: again, this is to prevent the creation of extra dependencies. For example, in the following program:

```
(1)  begin
(2)    sum := sum_0;
(3)    i := 1;
(4)    while i ⩽ 10 do
(5)      A( var sum, i) od;
(6)    PRINT("sum = ", sum)
(7)  where
(8)  proc A( var x, y) ≡
(9)    Add(y var x);
(10)   Inc( var y) end
(11) proc Add(b var a) ≡
(12)   a := a + b end
(13) proc Inc( var z) ≡
(14)   Add(1 var z) end
(15) end
```

If we slice on the value of z on line 14 in the body of Inc, the Syntactic_Slice transformation correctly recognises that the first parameter to A is redundant, and therefore the variable sum can be eliminated:

```
begin
  i := 1;
  while i ⩽ 10 do A( var i) od
where
proc A( var y) ≡
  Inc( var y) end
proc Add(b var a) ≡
  a := a + b end
proc Inc( var z) ≡
  Add(1 var z) end
end
```

A detailed discussion of interprocedural slicing is beyond the scope of this paper, but see Section 7.1 for a brief description of the implementation in FermaT.

6.7 Slicing Nondeterministic Programs

Binkley and Gallagher's definition of a slice [9] is as follows:

DEFINITION 6.15. For statement s and variable v, the slice $\mathbf{S}$ of program $\mathbf{P}$ with respect to the slicing criterion $\langle s;\ v\rangle$ is any executable program with the following properties:

(1) $\mathbf{S}$ can be obtained by deleting zero or more statements from $\mathbf{P}$.

(2) If $\mathbf{P}$ halts on input I, then the value of v at statement s each time s is executed in $\mathbf{P}$ is the same in $\mathbf{P}$ and $\mathbf{S}$. If $\mathbf{P}$ fails to terminate normally s may execute more times in $\mathbf{S}$ that in $\mathbf{P}$, but $\mathbf{P}$ and $\mathbf{S}$ compute the same values each time s is executed by $\mathbf{P}$.

This definition does not work with nondeterministic programs. Consider the program $\mathbf{P}$:

if true $\rightarrow x := 1$
$\Box$ **true** $\rightarrow x := 2$ **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 $\mathbf{P}$! Even $\mathbf{P}$ itself is not a valid slice: since the value of x at $y := x$ may be different each time $y := x$ is executed. Our definition of slicing avoids this flaw and is capable of handling nondeterministic statements. 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 abstraction 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.

6.8 Minimal Slices

Both Weiser's definition and ours allow the whole program as a valid slice for *any* slicing criterion, however restrictive that criterion is in comparison to the final state space. For program understanding and debugging, small slices are more useful than large slices, so it would appear to be a reasonable requirement to place on any slicing algorithm that the slices generated by the algorithm should be *minimal:* either in the sense of minimising the total number of statements, or at least in the weaker sense that no further statements can be deleted. So we define:

DEFINITION 6.16. A *minimal slice* of $\mathbf{S}$ on X is any syntactic slice $\mathbf{S}'$ such that if $\mathbf{S}'' \sqsubseteq \mathbf{S}'$ is also a syntactic slice, then $\mathbf{S}'' = \mathbf{S}'$.

Note that a minimal slice, according to this definition, is not necessarily unique and is not necessarily a slice with the smallest number of statements. Consider the program $\mathbf{S}$:

(1) $x := 2;$
(2) $x := x + 1;$
(3) $x := 3$

A syntactic slice can be obtained from $\mathbf{S}$ by deleting line 3 to give $\mathbf{S}'$:

(1) $x := 2;$
(2) $x := x + 1$

This program is a minimal slice (according to Definition 6.16), because neither of the remaining statements can be deleted. But there is another minimal slice of **S**, namely $x := 3$, which has fewer statements than $\mathbf{S}'$.

The case of a minimal slice with the minimal number of statements may also not be unique. Consider the program:

(1) $x := 1;$
(2) $x := x + 2;$
(3) $x := 2;$
(4) $x := x + 1$

This has two different minimal slices, both of which have two statements, namely:

(1) $x := 1;$
(2) $x := x + 2$

and

(1) $x := 2;$
(2) $x := x + 1$

Although of theoretical interest, demanding that the slices be minimal is too restrictive a requirement to place on a slicing algorithm. This is because, as Weiser pointed out [43], an algorithm for finding minimal slices can be converted into an algorithm for solving the halting problem. The halting problem is non-computable, hence the minimal slicing problem is also non-computable.

THEOREM 6.17. *Any minimal slicing algorithm can solve the halting problem.*

Proof: Let **S** be any program and consider the program **S**; $x := 0$, where x is any variable which does not appear in **S**. There are two cases to consider:

(1) If **S** may be nonterminating, then for any valid syntactic slice of **S**; $x := 0$ on $\{x\}$, if the slice still contains the assignment $x := 0$ then that statement can be deleted and the result will still be a valid slice of **S**; $x := 0$. Therefore, any minimal slice of **S**; $x := 0$ will not contain the assignment.
(2) If **S** always terminates, then the sliced program has to set x to zero, so the final assignment must appear in any valid slice of **S**; $x := 0$ on $\{x\}$.

So if we had a program which computes minimal syntactic slices, then we could solve the halting problem for any program **S** by computing the minimal slice of the program **S**; $x := 0$ on $\{x\}$ and simply observing if the result ends in the statement $x := 0$. If it does, then **S** terminates, while if it doesn't then **S** does not terminate. ■

6.9 Semantic and Amorphous Slicing

The definition of a syntactic slice immediately suggests a generalisation: why not keep the semantic relation and drop the syntactic relation? In other words, why not drop the requirement that $\mathbf{S}' \sqsubseteq \mathbf{S}$?

Harman and Danicic [16,18] coined the term "amorphous program slicing" for a combination of slicing and transformation of executable programs. So far the transformations have been restricted to restructuring snd simplifications, but the definition of an amorphous slice allows any transformation (in any transformation theory) of executable programs.

We define a "semantic slice" to be any semi-refinement in WSL, so the concepts of semantic slicing and amorphous slicing are distinct but overlapping. A semantic slice is defined in the context of WSL transformation theory, while an amorphous slice is defined in terms of executable programs (WSL allows nonexecutable statements including abstract specification statements and guard statements). Also, amorphous slices are restricted to finite programs, while WSL programs (and hence, semantic slices) can include infinitary formulae. To summarise:

(1) Amorphous slicing is restricted to finite, executable programs. Semantic slicing applies to any WSL programs including non-executable specification statements, non-executable guard statements, and programs containing infinitary formulae;
(2) Semantic slicing is defined in the particular context of the WSL language and transformation theory: amorphous slicing applies to any transformation theory or definition of program equivalence on executable programs.

The relation between a WSL program and its semantic slice is a purely semantic one: compare this with a "syntactic slice" where the relation is primarily a syntactic one with a semantic restriction.

Definition 6.18. A *semantic slice* of $\mathbf{S}$ on X is any program $\mathbf{S}'$ such that:

$$\Delta \vdash \mathbf{S};\ \textsf{\textbf{remove}}(W \setminus X) \ \preccurlyeq\ \mathbf{S}';\ \textsf{\textbf{remove}}(W \setminus X)$$

Note that while there are only a finite number of different syntactic slices (if $\mathbf{S}$ contains n statements then there are at most 2^n different programs $\mathbf{S}'$ such that $\mathbf{S}' \sqsubseteq \mathbf{S}$) there are infinitely many possible semantic slices for a program: including slices which are actually *larger* than the original program. Although one would normally expect a semantic slice to be no larger than the original program, [37,38] discuss cases where a high-level abstract specification can be larger than the program while still being arguably easier to understand and more useful for comprehension and debugging. A program might use some very clever coding to re-use the same data structure for more than one purpose. An equivalent program which internally uses two data structures might contain more statements and be less efficient while still being easier to analyse and understand. See [37] and [38] for a discussion of the issues.

Semantic refinement is implemented in FermaT via a process of *abstraction* and *refinement*. The *Representation Theorem* of WSL shows that for *any* WSL program there exists a WSL specification (containing a single specification statement) which implements that program:

Theorem 6.19. *The Representation Theorem*
Let $\mathbf{S}\colon V \to V$, be any kernel language statement and let $\boldsymbol{x}$ be a list of all the variables in V. Then for any countable set Δ of sentences:

$$\Delta \vdash \mathbf{S} \ \approx\ [\neg \mathrm{WP}(\mathbf{S}, \textsf{\textbf{false}})];\ \boldsymbol{x} := \boldsymbol{x}'.(\neg \mathrm{WP}(\mathbf{S}, \boldsymbol{x} \neq \boldsymbol{x}') \ \wedge\ \mathrm{WP}(\mathbf{S}, \textsf{\textbf{true}}))$$

Proof: Let $V' = V \cup \boldsymbol{x}'$, where let M be any model for Δ, and let $f = \text{int}_M\,(\mathbf{S}, V, V)$ and $f' = \text{int}_M\,(\mathbf{S}, V', V')$.

Now, f' is the program f with its initial and final state spaces extended to include $\boldsymbol{x}$', hence if s is any state in $D_{\mathcal{H}}(V)$ and $t \in f(s)$ and s' is any extension of the state s to $\boldsymbol{x}$' then there exists $t' \in f'(s')$, the corresponding extension of t, with $t'(z) = s'(z)$ for every $z \in \boldsymbol{x}'$ and $t'(z) = t(z)$ for every $z \notin \boldsymbol{x}'$.

If $\mathbf{S}$ is null for some initial state then $\text{WP}(\mathbf{S}, \textsf{false})$ is true for that state, hence the guard is false and the specification is also null. If $\mathbf{S}$ is undefined then $\text{WP}(\mathbf{S}, \textsf{true})$ is false, and the specification statement is also undefined. So we only need to consider initial states for which $\mathbf{S}$ is both defined and non-null. Fix s as any element of $D_{\mathcal{H}}(V)$ such that $f(s)$ is non-empty and $\perp \notin f(s)$. Then for any extension s' of s, $f'(s')$ is non-empty and $\perp \notin f'(s')$. Let:

$$g = \text{int}_M\,(\textsf{add}(\boldsymbol{x}');\ [\neg\text{WP}(\mathbf{S}, \boldsymbol{x} \neq \boldsymbol{x}')], V, V')$$

For each $t \in D_{\mathcal{H}}(V)$ we define $s_t \in D_{\mathcal{H}}(V)'$ to be the extension of s which assigns to $\boldsymbol{x}$' the same values which t assigns to $\boldsymbol{x}$. We will prove the following Lemma:

LEMMA 6.20. *For every $t \in f(s)$ there is a corresponding $s_t \in g(s)$ and every element of $g(s)$ is of the form s_t for some $t \in f(s)$.*

Proof: Let t be any element of $f(s)$. t gives a value to each variable in $\boldsymbol{x}$ and therefore can be used to define an extension $s_t \in D_{\mathcal{H}}(V)'$ of s where the values assigned to $\boldsymbol{x}$' by s_t are the same values which t assigns to $\boldsymbol{x}$. (The values given to any other variables are the same in s, t, and s_t since $\mathbf{S}$ can only affect the values of variables in $\boldsymbol{x}$.) Then we claim:

$$s_t \notin \text{int}_M\,(\text{WP}(\mathbf{S}, \boldsymbol{x} \neq \boldsymbol{x}'), V')$$

To prove this we note that a possible final state for f' on initial state s_t is the extension $t' \in D_{\mathcal{H}}(V)'$ of the state t, where t' gives the same values to $\boldsymbol{x}$' as s_t (the initial state). But these values are the same as the values t (and hence t') gives to $\boldsymbol{x}$, so t' does not satisfy the condition $\text{int}_M\,(\boldsymbol{x} \neq \boldsymbol{x}', V')$. So not all final states for f' on initial state s_t satisfy the condition, so s_t does not satisfy $\text{int}_M\,(\text{WP}(\mathbf{S}, \boldsymbol{x} \neq \boldsymbol{x}'), V')$. Hence:

$$s_t \in \text{int}_M\,(\neg\text{WP}(\mathbf{S}, \boldsymbol{x} \neq \boldsymbol{x}'), V')$$

and hence $s_t \in g(s)$.

Conversely, every final state $t' \in g(s)$ leaves the values of $\boldsymbol{x}$ the same as in s and assigns values to $\boldsymbol{x}$' such that $t' \notin \text{wp}(f', \text{int}_M\,(\boldsymbol{x} \neq \boldsymbol{x}', V'))$ is satisfied. For each such t' we can define a state $t \in D_{\mathcal{H}}(V)$ which assigns to $\boldsymbol{x}$ the values t' assigns to $\boldsymbol{x}$'. Then $t' = s_t$ where s_t is as defined above. Suppose $t \notin f(s)$, then every terminal state in $f(s)$ must have values assigned to $\boldsymbol{x}$ which differ from those s_t assigns to $\boldsymbol{x}$'. But then every terminal state of $f'(s_t)$ would satisfy $\text{int}_M\,(\boldsymbol{x} \neq \boldsymbol{x}', V')$ and hence s_t, and therefore t', would satisfy $\text{wp}(f', \text{int}_M\,(\boldsymbol{x} \neq \boldsymbol{x}', V'))$ which is a contradiction. So $t \in f(s)$ as required. This completes the proof of the Lemma. ■

To complete the main proof, we note that the state transformation

$$g' = \text{int}_M\,(\textsf{add}(\boldsymbol{x});\ [\boldsymbol{x} = \boldsymbol{x}'], V', V')$$

maps each s_t to the set $\{t\}$. Hence $f(s) = (g;\ g')(s)$ and this holds for all initial states s on which $\mathbf{S}$ is defined and determinate. Hence $\mathbf{S}$ is equivalent to the given specification. ■

For a general statement $\mathbf{S}\colon V \to W$ we have the corollary:

COROLLARY 6.21. *Let $\mathbf{S}\colon V \to W$, be any kernel language statement and let $\boldsymbol{x}$ be a list of all the variables in W. Without loss of generality we may assume that $W \subseteq V$ (Any variables added by $\mathbf{S}$ are already in the initial state space). Let $\boldsymbol{y}$ be a list of the variables removed by $\mathbf{S}$, so $\boldsymbol{x} \cap \boldsymbol{y} = \varnothing$ and $\boldsymbol{x} \cup \boldsymbol{y} = V$. Then $\mathbf{S}$ is equivalent to:*

(1) $[\neg \mathrm{WP}(\mathbf{S}, \textsf{false})]$;
(2) $\boldsymbol{x} := \boldsymbol{x}'.(\neg \mathrm{WP}(\mathbf{S}, \boldsymbol{x} \neq \boldsymbol{x}') \ \wedge\ \mathrm{WP}(\mathbf{S}, \textsf{true}))$;
(3) $\textsf{remove}(\boldsymbol{y})$

This theorem gives us an alternative representation for the weakest precondition of a statement:

COROLLARY 6.22. *For any statement $\mathbf{S}$:*

$$\begin{aligned}\mathrm{WP}(\mathbf{S}, \mathbf{R}) \iff & \\ & \mathrm{WP}(\mathbf{S}, \textsf{false}) \ \vee\ \big(\exists \boldsymbol{x}'.\, \neg \mathrm{WP}(\mathbf{S}, \boldsymbol{x} \neq \boldsymbol{x}') \ \wedge\ \mathrm{WP}(\mathbf{S}, \textsf{true}) \\ & \qquad\qquad \wedge\ \forall \boldsymbol{x}'.\, (\neg \mathrm{WP}(\mathbf{S}, \boldsymbol{x} \neq \boldsymbol{x}') \Rightarrow \mathbf{R}[\boldsymbol{x}'/\boldsymbol{x}])\big)\end{aligned}$$

where $\boldsymbol{x}$ is the variables assigned to by $\mathbf{S}$ as above.

Proof: Convert $\mathbf{S}$ to its specification equivalent using Theorem 6.19, take the weakest precondition for $\mathbf{R}$ and simplify the result. ■

The point of this corollary is that it expresses the weakest precondition of a statement for *any* postcondition as a simple formula containing a single occurrence of postcondition itself plus some weakest preconditions of fixed formulae.

In [41] we describe a partial implementation of the representation theorem in the form of a program transformation called Prog_To_Spec. This is used in combination with a syntactic slicing algorithm plus other transformations to develop a powerful conditioned semantic slicing algorithm. A *conditioned slice* [11,17] is a program slice which makes use of assertions added to the code to simplify the slice. The slicer applies the abstraction transformation Prog_To_Spec to blocks of code which do not contain loops, it then uses FermaT's condition simplifier to simplify the resulting specification. The simplifier can make use of assertions to simplify the specification, thus generating conditioned slices. A syntactic slicing algorithm is applied to the resulting program (with some semantic slicing extensions). Further simplification transformations, such as Constant_Propagation, are applied and any remaining specification statements are refined (using the Refine_Spec transformation) into combinations of assertions, assignments and IF statements, where possible.

For example, the statement:

var $\langle x := x\rangle$:
  if $p = q$

then $x := 18$
else $x := 17$ **fi**;
if $p \neq q$
then $y := x$
else $y := 2$ **fi** **end**

is abstracted, via Prog_To_Spec, to the specification statement:

$\langle y \rangle := \langle y' \rangle.(y' = 2 \ \wedge\ p = q \ \vee\ y' = 17 \ \wedge\ p \neq q)$

and subsequently refined, via Refine_Spec to:

if $p = q$ **then** $y := 2$ **else** $y := 17$ **fi**

The abstraction and refinement process can be used to simplify statements based on preceding *and subsequent* assertions: semantic slices can delete code which would falsify a later assertion if executed. This will be particularly important when we consider Conditioned Slicing in Section 6.14.

Returning to the example in Section 6.5, applying FermaT's Semantic_Slice transformation to the program:

(1) $y := y + 1$;
(2) **if** $y > 0$
(3) **then** $x := 2$
(4) **else** $x := 2$ **fi**

to slice on the final value of x gives $x := 2$ as the result.

6.10 Operational Slicing

An intermediate option between syntactic slicing and full semantic slicing is to restrict the transformations to preserve *operational* semantics, using the technique in Section 5.

DEFINITION 6.23. Program $\mathbf{S}'$ is an *operational slice* of $\mathbf{S}$ on X if there exists a sequence of statements $\mathbf{S}_1, \dots, \mathbf{S}_n$ such that $\mathbf{S}_1 = \mathbf{S}$, $\mathbf{S}_n = \mathbf{S}'$ and for each $1 \leq i < n$: either $\mathbf{S}_{i+1}$ is a syntactic slice of $\mathbf{S}_i$ on X, or:

$$\Delta \vdash \mathsf{annotate}(\mathbf{S}_i) \ \preccurlyeq\ \mathsf{annotate}(\mathbf{S}_{i+1})$$

It might be thought that the following is a simpler definition (which does not require a sequence of intermediate programs):

$$\Delta \vdash \mathsf{annotate}(\mathbf{S});\ \textbf{remove}(W \setminus X) \ \preccurlyeq\ \mathsf{annotate}(\mathbf{S}');\ \textbf{remove}(W \setminus X)$$

But this is incorrect because the seq variable (recording the sequence of states) is one of the variables removed by the statement **remove**$(W \setminus X)$, which means that all of the annotations are redundant code! On the other hand, if we add seq to X to stop it from being removed, then the suggested definition is much too restrictive: no statements can be deleted since they all contribute to the value of seq. This is why we define operational slicing in terms of a finite sequence of syntactic slicing and operational transformation steps. Some of the operational transformations may enable further slicing, and the slicing step may enable further transformation, so it is not sufficient to define operational slicing in terms of a single transformation step followed by a single slicing step if we want a definition of slicing that is *transitive*. (In other words: if we want a slice of a slice to be itself a valid slice). The example in Section 8 illustrates this point.

An operational slice is therefore a combination of syntactic slicing and operational transformations (such as restructuring). The implementation of operational slicing can iterate the slicing and transformation steps until the result converges (the convergence is guaranteed provided the operational transformations either leave the program unchanged or return a strictly smaller program).

6.11 Slicing At Any Position

To slice at an arbitrary position in the program we need to preserve the sequence of values taken on by the given variables at that point in the program. To do this, we simply insert an assignment to a new variable slice at the required position which records the current values of the variables on a list. If $X = \{x_1, \ldots, x_n\}$ is the set of variables we are interested in then we insert the statement:

$$\mathsf{slice} := \mathsf{slice} \mathbin{+\!\!+} \langle\langle x_1, \ldots, x_n \rangle\rangle$$

at the point of interest, in order to record the current values of the variables at that point. Then we slice at the end of the program on the single variable slice. (Note that we *append* to the list in slice, rather than simply assign to it, because we are interested in the whole sequence of values, not just the last set of values taken on by the variables.)

This process can be generalised to slicing at several points in the program, perhaps with a different set of "variables of interest" at each point, simply by inserting the slice assignments at the appropriate places.

One peculiarity of this definition is that if we slice at a point in the program which is within a compound statement that does not modify any of the variables in the slicing criteria, then we can end up with slices which appear to be larger than necessary. For example, suppose that we slice on x within this **if** statement at the point just before the assignment to z on line 3:

(1) $x := g(z)$;
(2) $y := f(z)$;
(3) **if** $y = 0$ **then** $z := 1$ **fi**

The annotated program is:

(1) $x := g(z)$;
(2) $y := f(z)$;
(3) **if** $y = 0$ **then** $\mathsf{slice} := \mathsf{slice} \mathbin{+\!\!+} \langle\langle x \rangle\rangle$; $z := 1$ **fi**;
(4) **remove**(x, y, z)

This is equivalent to:

(1) $x := g(z)$;
(2) $y := f(z)$;
(3) **if** $y = 0$ **then** $\mathsf{slice} := \mathsf{slice} \mathbin{+\!\!+} \langle\langle x \rangle\rangle$ **fi**;
(4) **remove**(x, y, z)

We cannot delete the assignment $y := f(z)$ on line 2 because it determines which branch of the **if** statement is taken, and this affects the final value of slice (although it does not affect the value of x). According to our definition, the slice has to preserve the test $y = 0$ and therefore preserve any previous modifications to y. In effect, by slicing at a particular position we are insisting that the given *position* should also appear in the sliced program. This is arguably correct in the sense that, if the slice has to preserve the sequence of values taken on by x at a particular point in the program, then a *corresponding point* (at which x takes on the *same* sequence of values) must appear in the slice. But if the **if** statement in the above example is deleted, then x may take on a different sequence of values! To be precise, there is *no* point in the new program at which x takes on the *same* sequence of values as at the slice point in the original program.

However, if it is not required to preserve the slice point then a simple solution is to allow the slicing algorithm to move all the assignments to slice upwards out of any enclosing structures as far as possible, before carrying out the slicing operation itself.

6.12 Dynamic Slicing

Although the term "dynamic program slice" was first introduced by Korel and Laski [21], it may be regarded as a non-interactive version of Balzer's notion of flowback analysis [5]. In flowback analysis, one is interested in how information flows through a program to obtain a particular value: the user interactively traverses a graph that represents the data and control dependences between statements in the program.

A dynamic slice of a program $\mathbf{P}$ is a reduced executable program $\mathbf{S}$ which replicates part of the behaviour of $\mathbf{P}$ on a particular initial state. We can define this initial state by means of an assertion. Suppose $V = \{v_1, v_2, \dots, v_n\}$ is the set of variables in the initial state space for $\mathbf{P}$, and $V_1, V_2, \dots, V_n$ are the initial values of these variables in the state of interest. Then the condition

$$\mathbf{A} \ =_{\text{DF}} \ v_1 = V_1 \ \wedge \ v_2 = V_2 \ \wedge \ \cdots \ \wedge \ v_n = V_n$$

is true for this initial state and false for every other initial state. The assertion $\{\mathbf{A}\}$ is **abort** for every initial state other than the specified one. So we define:

DEFINITION 6.24. A *Dynamic Syntactic Slice* of $\mathbf{S}$ with respect to a formula $\mathbf{A}$ of the form

$$v_1 = V_1 \ \wedge \ v_2 = V_2 \ \wedge \ \cdots \ \wedge \ v_n = V_n$$

where $V = \{v_1, v_2, \dots, v_n\}$ is the initial state space of $\mathbf{S}$ and V_i are constants, and the set of variables X is a subset of the final state space W of $\mathbf{S}$, is any program $\mathbf{S}' \sqsubseteq \mathbf{S}$ such that:

$$\Delta \vdash \{\mathbf{A}\};\ \mathbf{S};\ \textsf{\textbf{remove}}(W \setminus X) \ \preccurlyeq \ \{\mathbf{A}\};\ \mathbf{S}';\ \textsf{\textbf{remove}}(W \setminus X)$$

The following is an example of a dynamic slice, from Danicic's thesis [13] which is based on an example by Agrawal and Horgan [1], which we have translated into WSL:

```
(1) while i < n do
(2)   read( var x, input);
(3)   if x < 0
(4)     then y := f1(x)
```

```
      else y := f2(x) fi;
   z := f3(y);
   write(z var output);
   i := i + 1 od
```

where read(**var** x, input) removes the first element of the list in input and stores it in x, and write(z **var** output) appends the value of z to the list in output.

The initial state has: $i = 1, n = 3, \mathsf{input} = \langle -4, 3\rangle$, and we are slicing on the final value of z. Unrolling the first two iterations of the **while** loop and simplifying gives:

```
i = 1 ∧ n = 3 ∧ input = ⟨−4, 3⟩;
read( var x, input);
if x < 0
   then y := f1(x)
    else y := f2(x) fi;
z := f3(y);
write(z var output);
i := 2;
read( var x, input);
if x < 0
   then y := f1(x)
    else y := f2(x) fi;
z := f3(y);
write(z var output);
i := 3;
while i < n do ... od
```

The final loop is equivalent to **skip**, since $n = 3$. So the loop body can be transformed in any way we please: in particular, by deleting the assignment $y := \mathsf{f1}(x)$. Outside the loop, the first occurrence of $y := \mathsf{f1}(x)$ is redundant because the value of y is only used in the first assignment to z, which is overwritten by the next assignment to z. The second occurrence of $y := \mathsf{f1}(x)$ is not executed, so can also be deleted. The write calls can also be deleted, since it only affects variable output which is not in the slicing criterion. We can then roll up the loop again to give:

```
i = 1 ∧ n = 3 ∧ input = ⟨−4, 3⟩;
while i < n do
   read( var x, input);
   if x < 0
      then skip
       else y := f2(x) fi;
   z := f3(y);
   i := i + 1 od
```

Agrawal and Horgan [1]. describe an algorithm for computing this slice by computing a *dynamic dependency graph* from an execution of the program on the given initial state. This contains a node for every execution of every statement in the program. An equivalent, but more efficient, structure is the *reduced dynamic dependency graph* in which a node is created only if another node with the same transitive dependencies does not already exist. This still requires executing the program on the given initial state: in fact, a considerable amount of computation is required as each statement in the program is executed, so the algorithm can take a long time to execute on a long running program.

The program transformation approach gives a way to compute a dynamic slice without necessarily executing the program and recording all or part of the sequence of states taken on. Consider the following slight change to Agrawal and Horgan's example:

```
while i < n do
  read( var x, input);
  if true → y := f1(x)
  □ true → y := f2(x) fi;
  z := f3(y);
  write(z var output);
  i := i + 1 od
```

where the conditional statement has been replaced by a nondeterministic choice. When the program is executed, it may choose the second branch of the **if** on both iterations of the loop body: in which case, the dynamic dependency graph will *not* include the first branch of the **if**, and it will be excluded from the slice. The transformation approach gives the correct result:

```
while i < n do
  read( var x, input);
  if true → y := f1(x)
  □ true → y := f2(x) fi;
  z := f3(y);
  i := i + 1 od
```

The problem is that for a nondeterministic program there may be *no* unique sequence of states, even when the program is started in a unique initial state. So it is not possible to extend *any* of the dynamic slicing algorithms which actually execute the program to an algorithm for slicing nondeterministic programs.

6.13 Minimal Dynamic Slices for Deterministic Programs

If we restrict our attention to deterministic programs, then it would appear to be possible, at least in theory, to compute the minimal dynamic slice for a given initial state. Recall that we are interested in the set of statements which are essential to computing the given output (the value(s) of the variable(s) in the slicing criterion). Surely we can simply keep deleting statements and re-running the program to see if we still get the same result? There are two problems with this "brute force" approach to computing minimal dynamic slices:

(1) It is not sufficient to delete statements one at a time until the program stops working. Consider the program:

$x := 1;$
$x := x + 2;$
$x := x - 2$

Deleting any individual statement will lead to a non-equivalent program, but deleting both the second and third statements gives the obvious minimal slice. The "brute force" algorithm is therefore to re-run the program for each of the $O(2^n)$ possible reductions of the original program (where the original program had n statements).

(2) It may be possible to delete statements from a program to yield a minimal slice that takes *arbitrarily longer to compute* than the original program. For example, consider slicing on the final value of y in:

while $x > 1000$ **do** $x := x - 1000$ **od**;
while $x > 0$ **do**
  $y := -y$;
  $x := x - 1$ **od**

Deleting the first **while** loop will not affect the final value of y but could dramatically increase the running time. Potential slices which increase the running time have to be ignored by the brute force algorithm, since otherwise it cannot distinguish between a non-terminating program and one which is simply taking arbitrarily longer to compute the same answer (this is the Halting Problem again).

Beszédas et al [6] claim that the union of all minimal dynamic slices will be a minimal static slice. Given an efficient algorithm for computing minimal dynamic slices, it would therefore be possible to compute a reasonable selection of minimal dynamic slices and therefore get at least an approximation to a minimal static slice (it is not possible to compute *all* the minimal dynamic slices, of course). One problem with this approach is that the union of two slices is not necessarily a slice [22]. A more serious problem, as the next theorem shows, is that the union of all minimal dynamic slices is not necessarily a minimal static slice.

THEOREM 6.25. *The union of all minimal dynamic slices is not necessarily a minimal static slice.*

Proof: Consider the program:

(1) $x := 2;$
(2) $x := y;$
(3) $x := 2 * x;$

where we are slicing on the final value of x. If $y = 1$ initially, then the minimal dynamic slice is just line 1 $x := 2$. On the other hand, if $y = 2$ initially, then the minimal dynamic slice is lines 2 and 3 $x := y;\ x := 2 * x$. The union of these two slices is the whole program, which is strictly larger than the minimal static slice $x := y;\ x := 2 * x$. ■

6.14 Conditioned Slicing

Researchers have generalised dynamic slicing and combined static and dynamic slicing in various ways. For example: some researchers allow a finite set of initial states, or a *partial* initial state which restricts a subset of the initial variables to particular values [29]. In our formalism, all of these generalisations are subsumed under the obvious generalisation of dynamic slicing: why restrict the initial assertion to be of the particular form $\{v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n\}$?

If we allow any initial assertion, then the result is called a *conditioned slice*:

DEFINITION 6.26. A *Conditioned Syntactic Slice* of $\mathbf{S}$ with respect to any formula $\mathbf{A}$ and set of variables X is any program $\mathbf{S}' \sqsubseteq \mathbf{S}$ such that:

$$\Delta \vdash \{\mathbf{A}\};\ \mathbf{S};\ \textsf{\textbf{remove}}(W \setminus X) \ \preccurlyeq\ \{\mathbf{A}\};\ \mathbf{S}';\ \textsf{\textbf{remove}}(W \setminus X)$$

Conditioned slicing is thus a generalisation of both static slicing (where the condition $\mathbf{A}$ is **true**) and dynamic slicing (where $\mathbf{A}$ takes on the form $v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n$ and the set $\{v_1, v_2, \ldots, v_n\}$ lists all the variables in the program). One algorithm for computing a conditioned slice is to use the initial condition to simplify the program before applying a syntactic slicing algorithm. Danicic et al [17] describe a tool called ConSIT, for slicing a program at a particular point, given that the initial state satisfies a given condition.

The ConSIT tool works on an intraprocedural subset of C using a three phase approach:

(1) Symbolically Execute: to propagate assertions through the program where possible;
(2) Produce Conditioned Program: eliminate statements which are never executed under the given conditions;
(3) Perform Static Slicing: using a traditional (syntactic) slicing method.

In ConSIT, the slicing condition can be given in the form of `ASSERT` statements scattered through the program: the authors [17] claim that these `ASSERT` statements are equivalent to a single condition on the initial state, but in general this requires assertions to be formulae of *infinitary* logic. This is because the general case of moving an assertion "backwards" over or out of a loop breaks down into a countably infinite sequence of cases depending on the number of possible iterations of the loop. Fortunately, the assertion statements in WSL are already expressed in infinitary logic, so this is not a problem in our framework.

In our transformation framework, the `ASSERT` statements are simply WSL assertions. The symbolic execution and producing the conditioned program are examples of transformations which can be applied to the WSL program plus assertions. In [30] we provide a number of transformations for propagating assertions and eliminating dead code.

THEOREM 6.27. *A set of assertions scattered through a program can be replaced by an equivalent assertion at the beginning of the program (in the sense that the two programs are equivalent).*

Proof: Let $\mathbf{S}$ be any program and let $\mathbf{S}'$ be constructed from $\mathbf{S}$ by deleting assertions (i.e. replacing the assertions with **skip** statements). Each assertion deletion is a semi-refinement, so by the Replacement Property (Section 6), **S**' is a semi-refinement of $\mathbf{S}$. So, by the definition of semi-refinement (Definition 6.4):

$$\Delta \vdash \mathbf{S} \ \approx\ \{\mathrm{WP}(\mathbf{S}, \textsf{\textbf{true}})\};\ \mathbf{S}'$$

which proves the theorem. ■

For example, to delete the assertion from $\mathbf{S};\ \{\mathbf{Q}\}$ we have:

$$\begin{aligned}\Delta \vdash \mathbf{S};\ \{\mathbf{Q}\} &\approx \{\mathrm{WP}(\mathbf{S};\ \{\mathbf{Q}\}, \textsf{\textbf{true}})\};\ \mathbf{S} \\ &\approx \{\mathrm{WP}(\mathbf{S}, \mathrm{WP}(\{\mathbf{Q}\}, \textsf{\textbf{true}}))\};\ \mathbf{S} \\ &\approx \{\mathrm{WP}(\mathbf{S}, \mathbf{Q})\};\ \mathbf{S}\end{aligned}$$

So, for example, $x := y + 1;\ \{x > 0\}$ becomes $\{y + 1 > 0\};\ x := y + 1$.

For an assertion in a loop we have:

$$\begin{aligned}&\Delta \vdash \textsf{\textbf{while}}\ \mathbf{B}\ \textsf{\textbf{do}}\ \{\mathbf{Q}\};\ \mathbf{S}\ \textsf{\textbf{od}} \\ &\quad\approx\ \{\textstyle\bigwedge_{n>0}\big(\bigwedge_{i<n} \mathrm{WP}((\mathbf{S};)^i, \mathbf{B}) \Rightarrow \mathrm{WP}((\mathbf{S};)^n, \mathbf{Q})\big)\}; \\ &\qquad \textsf{\textbf{while}}\ \mathbf{B}\ \textsf{\textbf{do}}\ \mathbf{S}\ \textsf{\textbf{od}}\end{aligned}$$

where $(\mathbf{S};)^0$ is **skip** and $(\mathbf{S};)^{n+1}$ is $\mathbf{S};\ (\mathbf{S};)^n$.

6.15 Conditioned Semantic Slicing

Again, a generalisation is suggested: why restrict ourselves to the assertion moving and dead code removal transformations of ConSIT? A conditioned semantic slice can be defined by simply removing the syntactic condition from the definition of a syntactic slice (i.e. the condition that $\mathbf{S}' \sqsubseteq \mathbf{S}$):

DEFINITION 6.28. *Suppose we have a program* $\mathbf{S}$ *and a slicing criterion, defined from* $\mathbf{S}$ *by inserting assertions and assignments to the* slice *variable to form* $\mathbf{S}'$. *A* conditioned semantic slice *of* $\mathbf{S}$ *with respect to this criterion is any program* $\mathbf{S}''$ *such that:*

$$\Delta \vdash \mathbf{S}';\ \textsf{\textbf{remove}}(W)\ \preccurlyeq\ \mathbf{S}'';\ \textsf{\textbf{remove}}(W)$$

Any syntactic slice is also a semantic slice (but not vice versa), so the conditioned semantic slice is a generalisation of syntactic, semantic, dynamic, conditioned and operational slicing in the sense that any of these slices is also a conditioned semantic slice.

7. SLICING IN FERMAT

Our transformation theory was developed in roughly the following stages:

(1) Start with a very simple and tractable kernel language;
(2) Develop proof techniques based on set theory and mathematical logic, for proving the correctness of transformations in the kernel language;
(3) Extend the kernel language by definitional transformations which introduce new constructs (the result is the WSL wide spectrum language);
(4) Develop a catalogue of proven WSL transformations: each transformation is proved correct by appealing to already proven transformations, or by translating to the kernel language and applying the proof techniques directly.
(5) Tackle some challenging program development and reverse engineering tasks to demonstrate the validity of this approach [31,33,37,39,42,46];
(6) Extend WSL with constructs for implementing program transformations (the result is called $\mathcal{METAWSL}$);

(7) Implement an industrial strength transformation engine in $\mathcal{META}$WSL with translators to and from existing programming languages. This allowed us to test our theories on large scale legacy systems (including systems written in IBM Assembler [34,35,40]).

FermaT is an industrial strength program transformation system, the result of over eighteen years of research and development, which has recently been released under the GNU GPL (General Public Licence). It is available for downloading from the following sites:

http://www.dur.ac.uk/martin.ward/fermat.html
http://www.cse.dmu.ac.uk/∼mward/fermat.html

FermaT's transformations include three slicers:

—Simple_Slice, which implements the algorithm in Section 6.3. The source code for Simple_Slice is given in the Appendix;
—Syntactic_Slice which is an interprocedural syntactic slicer which can handle unstructured programs; and
—Semantic_Slice which is a semantic slicer which uses abstraction and refinement.

FermaT implements a large number of powerful program transformations, these combined with syntactic slicing make it possible to use FermaT for general conditioned semantic slicing.

The FermaT syntactic slicer has the following features:

—Handles arbitrary control flow (including WSL code translated from assembler language) via "action systems";
—Interprocedural slicing, which handles the "calling context" problem correctly (see [19] and Section 7.1) combined with action systems;
—Efficient algorithms for handling large and complex programs;

Intermediate steps in the algorithm are not restricted to being a reduction of the original program, so long as the final result is a reduction. So FermaT uses a "destructuring" algorithm (the opposite of restructuring) to convert the program to "basic blocks" form. Then we slice on the basic blocks form of the program, using a dataflow algorithm. Then determine which statements in the original program are present in the sliced blocks program: any statements not still present can be deleted.

—Construct the Basic Blocks file by analysing control flow;
—Construct control dependencies using the optimal algorithm of Pingali and Bilardi [25].
—Construct the Static Single Assignment form [7,12] of the basic blocks file: this is a concise representation of all data dependencies in the file. We use the near-linear algorithm given in [7].
—Track control and data dependencies backwards from the given starting points and variables using a simple graph reachability algorithm. Mark all node/variable pairs reached.

—Delete all statements where none of the modified variables in the statement were marked. The deletion is carried out as a two-stage process: first paste **skip** or **exit** statements over the unwanted statements (this preserves the positions of all statements in the program and constructs the reduced program), then use FermaT transformations to delete all **skip**s and all redundant actions in action systems.

7.1 Interprocedural Slicing in FermaT

Weiser's original algorithm for interprocedural slicing [43] constructed dataflow links from every procedure call to the top of the procedure body, and from the end of the procedure body to every procedure return point. His algorithm therefore does not take into account the calling context, and this leads to two problems:

—If a slice includes one call-site on a procedure then the slice includes all call-sites on the procedure, and

—If a slice includes one parameter of a procedure, it must include all parameters.

FermaT solves both of these problems by generating *procedure summaries*.

A procedure summary contains a list of the variables modified (directly or indirectly) by the procedure body. For each modified variable, the summary lists all the input parameters and global variables which contribute to the new value of the variable. So to calculate a procedure summary, all we need to do is slice backwards from each output variable from the end of the procedure body back to the beginning and see which input parameters are included in the slice. For this slicing procedure to work accurately, we need to already have available summaries of all procedures called by the current procedure: so the summaries are calculated in a "bottom up" traversal of the call graph. Procedures which call no other procedures are summarised first, and so on. For recursive procedures, or mutually-recursive groups of procedures the recursion is ignored and an initial "empty" set of summaries is provided: i.e. the system assumes that the outputs do not depend on any inputs. Then the whole procedure summary calculation is iterated to convergence. This process is guaranteed to terminate because each iteration can only add dependencies to the summary. The result is guaranteed to be correct because any dependency which appears via a recursive call must ultimately resolve to a set of dependencies which do not depend on recursive calls. So by starting with the "non recursive" dependencies and iterating we are guaranteed to include all dependencies.

A different approach is given by Forgacs and Gyimothy [15] which works by finding the set of *Strongly Connected Components* (SCC) in the call graph. A strongly connected component is a maximal subgraph in which every vertex is reachable from every other vertex. A single recursive component forms a SCC, as does a collection of mutually recursive procedures. Forgacs and Gyimothy's approach [15] is to reduce the call graph by replacing each SCC by a single node. The processing for this "supernode" uses the algorithm of Weiser [43] and therefore does not track call site information. If the call graph is reduced by replacing each SCC by a single node, then the graph becomes a DAG (Directed Acyclic Graph) and there is no recursion. The nodes in the DAG can be topologically sorted to give a general invocation order which gives the order in which to compute summaries. This algorithm is efficient for large programs, but has the drawback that *within* each

SCC it is not possible to preserve information across call sites. In the worst case, where every procedure is strongly connected to every other procedure, the algorithm reduces to Weiser's simple algorithm. This worst case could occur for example, if every procedure directly or indirectly calls a particular "utility" procedure which in turn calls a procedure at the top of the calling hierarchy.

The example at the end of Section 6.5 illustrates the effect of a loss of call site information: with Weiser's algorithm, slicing on z in the body of Inc creates a dependency on the second parameter of Add. This dependency is tracked into the body of A, so the slicer incorrectly concludes that the first parameter of A is needed. FermaT's procedure summaries ensure that the slice is computed correctly in this case. Each summary is computed from accurate call site information, so each summary provides accurate information. Even for recursive and mutually recursive procedures, the iteration to convergence prevents information from "leaking" from one call site to another.

8. SLICING EXAMPLES

The following WSL program is a translation of the C program in [11]:

```
i := 1;
posprod := 1;
negprod := 1;
possum := 0;
negsum := 0;
while i ⩽ n do
  a := input[i];
  if a > 0
    then possum := possum + a;
         posprod := posprod ∗ a
  elsif a < 0
      then negsum := negsum − a;
           negprod := negprod ∗ (−a)
  elsif test0 = 1
      then if possum ⩾ negsum
             then possum := 0
              else negsum := 0 fi;
           if posprod ⩾ negprod
             then posprod := 1
              else negprod := 1 fi fi;
  i := i + 1 od;
if possum ⩾ negsum
  then sum := possum
   else sum := negsum fi;
if posprod ⩾ negprod
  then prod := posprod
   else prod := negprod fi
```

Suppose we want to slice this program with respect to the sum variable at the end of the program and with the additional constraint that all the input values are positive. We can either add the assertion $\{\forall i.\, 1 \leqslant i \leqslant n \Rightarrow \mathsf{input}[i] > 0\}$ to the top of the program, or equivalently add the assertion $\{a > 0\}$ just after the assignment to a at the top of the loop (i.e. after line 7). We also append the **remove** statement:

$$\textbf{remove}(i, \mathsf{posprod}, \mathsf{negprod}, \mathsf{possum}, \mathsf{negsum}, n, a, \mathsf{test0})$$

to the program. This removes all the variables we are not interested in.

We use the assertion to delete unreachable code and then apply the FermaT syntactic slicing transformation. The resulting conditioned syntactic slice is:

```
i := 1;
possum := 0;
negsum := 0;
while i ⩽ n do
  a := input[i];
  {a > 0};
  if a > 0
    then possum := possum + a fi;
  i := i + 1 od;
if possum ⩾ negsum
  then sum := possum fi;
remove(i, posprod, negprod, possum, negsum, n, a, test0)
```

ConSIT [17], a conditioned syntactic slicer, took about 20 minutes on a Pentium II running at 233MHz to compute the same slice which FermaT computed in a fraction of a second. Note that ConSIT was unable to remove the variable negsum since it is a purely syntactic conditioned slicer: both programs have computed a minimal syntactic slice since no further statements can be deleted. But with semantic slicing we can do much more. FermaT's semantic slicer took just over half a second (0.63 seconds) on a 2.6GHz PC to produce the slice:

```
{input[1..n] > 0};
sum := REDUCE("+", input[1..n])
```

Here, the initial slice over the program deleted the assignments to negsum in the loop body. A subsequent Constant_Propagation replaced references to negsum by zero, after which a second slicing operation removed negsum from the program. At this point, the While_To_Reduce transformation could reduce the simplified **while** loop to the assignment:

```
possum := possum + REDUCE("+", input[1..n])
```

Then a subsequent abstraction and refinement step could simplify the whole program: making use of the fact that if $\mathsf{input}[1..n] > 0$ then $\mathsf{REDUCE}(\text{“+”}, \mathsf{input}[1..n]) \geqslant 0$, so FermaT can deduce the result of the test in the final **if** statement.

All these transformations were carried out automatically by Semantic_Slice to give the result above. This is a concise specification of the final value of sum under the given slicing condition.

9. CONCLUSION

In this paper we have provided a complete mathematical foundation for program slicing in terms of the WSL transformation theory. The mathematical foundation is independent of the various representations of programs upon which our slicing algorithms act. Slicing is no longer so closely intertwined with the presence of absence of various types of edges in the program dependency graphs. Instead, new ideas about slicing or algorithms for slicing can be represented and analysed in WSL and then implemented. The implementations can be checked against the formal definition of slicing: in fact it is even possible to *prove* the correctness of a slicing algorithm by transforming the specification into the implementation.

Traditional (or syntactic) slicing, which is restricted to deleting irrelevant statements has the advantage of a finite set of solutions and may be useful in debugging situations where programmers are already familiar with the layout of the code. But in more general program comprehension, reverse engineering, reengineering and migration tasks, it is much more useful to use transformations to simplify the slices and even present the sliced program at a higher level of abstraction.

A particularly useful application of conditioned semantic slicing is to remove the error handling code during program comprehension or reverse engineering. Often much of the code in a program is there to handle errors: this code can obscure the structure and function of the "main line" code. By adding assertions in appropriate places and slicing on the outputs of interest a much more concise specification of the main function can be generated.

We finish with a summary of the various forms of slicing discussed in the paper. In each case except the last, **S**' is a slice of **S** under the slicing criterion X. In the last case, the slicing criterion is defined by inserting assignments to the slice variable at the points of interest.

(1) Syntactic Slice: $\mathbf{S}' \sqsubseteq \mathbf{S}$ and

$$\Delta \vdash \mathbf{S};\ \textsf{\textbf{remove}}(W \setminus X) \ \preccurlyeq\ \mathbf{S}';\ \textsf{\textbf{remove}}(W \setminus X)$$

(2) Minimal Slice: **S**' is a syntactic slice and if $\mathbf{S}'' \sqsubseteq \mathbf{S}'$ is also a syntactic slice, then $\mathbf{S}'' = \mathbf{S}'$.

(3) Semantic Slice.

$$\Delta \vdash \mathbf{S};\ \textsf{\textbf{remove}}(W \setminus X) \ \preccurlyeq\ \mathbf{S}';\ \textsf{\textbf{remove}}(W \setminus X)$$

(4) Operational Slice: there exists a sequence of statements $\mathbf{S}_1, \dots, \mathbf{S}_n$ such that $\mathbf{S}_1 = \mathbf{S}$, $\mathbf{S}_n = \mathbf{S}'$ and for each $1 \leq i < n$: either $\mathbf{S}_{i+1}$ is a syntactic slice of $\mathbf{S}_i$ on X, or:

$$\Delta \vdash \textsf{annotate}(\mathbf{S}_{i+1}) \ \preccurlyeq\ \textsf{annotate}(\mathbf{S}_i)$$

(5) Dynamic Syntactic Slice: $\mathbf{S}' \sqsubseteq \mathbf{S}$ and

$$\Delta \vdash \{\mathbf{A}\};\ \mathbf{S};\ \textsf{\textbf{remove}}(W \setminus X) \ \preccurlyeq\ \{\mathbf{A}\};\ \mathbf{S}';\ \textsf{\textbf{remove}}(W \setminus X)$$

where **A** is of the form: $v_1 = V_1 \wedge v_2 = V_2 \wedge \cdots \wedge v_n = V_n$.

(6) Conditioned Syntactic Slice: $\mathbf{S}' \sqsubseteq \mathbf{S}$ and

$$\Delta \vdash \{\mathbf{A}\};\ \mathbf{S};\ \textsf{\textbf{remove}}(W \setminus X) \ \preccurlyeq\ \{\mathbf{A}\};\ \mathbf{S}';\ \textsf{\textbf{remove}}(W \setminus X)$$

where **A** is unrestricted.

(7) Conditioned Semantic Slice:

$$\Delta \vdash \mathbf{S}''; \ \textsf{\textbf{remove}}(W) \ \preccurlyeq \ \mathbf{S}'; \ \textsf{\textbf{remove}}(W)$$

where **S**" is constructed from **S** by adding assertions and assignments to a new variable, slice, at the points of interest.

Acknowledgements

The research described in this paper was funded by De Montfort University and Software Migrations Ltd. The idea of initialising v to $\perp$ in a **var** clause was suggested by Mark Harman. A very interesting discussion with David Binkley led to the Section 6.13. I would like to thank the referees for their careful reading of the first draft, their suggestions for improvements and their encouraging support.

References

[1] H. Agrawal & J. R. Horgan, "Dynamic Program Slicing," *SIGPLAN Notices* 25 (June, 1990), 246–256.

[2] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, Amsterdam, 1980.

[3] R. J. R. Back, "A Calculus of Refinements for Program Derivations," *Acta Informatica* 25 (1988), 593–624.

[4] R. J. R. Back & J. von Wright, "Refinement Concepts Formalised in Higher-Order Logic," *Formal Aspects of Computing* 2 (1990), 247–272.

[5] R. Balzer, "EXDAMS – EXtendable Debugging And Monitoring System," Proceedings of the AFIPS SJCC, 1969.

[6] Árpád Beszédas, Csaba Faragó, Zsolt Mihály Szabó, János Csirik & Tibor Gyimothy, "Union slices for program maintenance.," *18th International Conference on Software Maintenance (ICSM), 3rd–6th October 2002, Montreal Quebec* (2002).

[7] Gianfranco Bilardi & Keshav Pingali, "The Static Single Assignment Form and its Computation," Cornell University Technical Report, July, 1999, ⟨http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/ssa.ps⟩.

[8] Dave Binkley, Mark Harman & Sebastian Danicic, "Amorphous Program Slicing," *Journal of Systems and Software* 68 (Oct., 2003), 45–64.

[9] David W. Binkley & Keith Gallagher, "A survey of program slicing," in *Advances in Computers*, Marvin Zelkowitz, ed., Academic Press, San Diego, CA, 1996.

[10] C. Bohm & G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *Comm. ACM* 9 (May, 1966), 366–371.

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

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman & F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependance Graph," *Trans. Programming Lang. and Syst.* 13 (July, 1991), 451–490.

[13] Sebastian Danicic, "Dataflow Minimal Slicing," London University, PhD Thesis, 1999.

[14] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[15] István Forgács & Tibor Gyimóthy, "An Efficient Interprocedural Slicing Method for Large Programs," *Proceedings of SEKE'97, the 9th International Conference on Software Engineering & Knowledge Engineering, Madrid, Spain* (1997).

[16] Mark Harman & Sebastian Danicic, "Amorphous program slicing," *5th IEEE International Workshop on Program Comprehesion (IWPC'97), Dearborn, Michigan, USA* (May 1997).

[17] Mark Harman, Sebastian Danicic & R. M. Hierons, "ConSIT: A conditioned program slicer," *9th IEEE International Conference on Software Maintenance (ICSM'00), San Jose, California, USA*, Los Alamitos, California, USA (Oct., 2000).

[18] Mark Harman, Lin Hu, Malcolm Munro & Xingyuan Zhang, "GUSTT: An Amorphous Slicing System Which Combines Slicing and Transformation," *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, Los Alamitos, California, USA (2001).

[19] Susan Horwitz, Thomas Reps & David Binkley, "Interprocedural slicing using dependence graphs," *Trans. Programming Lang. and Syst.* 12 (Jan., 1990), 26–60.

[20] C. R. Karp, *Languages with Expressions of Infinite Length*, North-Holland, Amsterdam, 1964.

[21] Bogdan Korel & Janusz Laski, "Dynamic program slicing," *Information Processing Letters,* 29 (Oct., 1988), 155–163.

[22] Andrea De Lucia, Mark Harman, Robert Hierons & Jens Krinke, "Unions of Slices are not Slices," *7th European Conference on Software Maintenance and Reengineering Benevento, Italy March 26-2* (2003).

[23] C. C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1994, Second Edition.

[24] C. C. Morgan & K. Robinson, "Specification Statements and Refinements," *IBM J. Res. Develop.* 31 (1987).

[25] Keshav Pingali & Gianfranco Bilardi, "Optimal Control Dependence Computation and the Roman Chariots Problem," *Trans. Programming Lang. and Syst.* (May, 1997), ⟨http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/toplas97.ps⟩.

[26] J. E. Stoy, *Denotational Semantics: the Scott-Strachy Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.

[27] R. D. Tennet, "The Denotational Semantics of Programming Languages," *Comm. ACM* 19 (Aug., 1976), 437–453.

[28] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages,* 3 (Sept., 1995), 121–189.

[29] G. A. Venkatesh, "The semantic approach to program slicing.," *SIGPLAN Notices* 26 (1991), 107–119, Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, June 26-28.

[30] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

[31] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/sorting-t.ps.gz⟩.

[32] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," Durham University, Technical Report, 1994, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/foundation2-t.ps.gz⟩.

[33] M. Ward, "Reverse Engineering through Formal Transformation Knuths "Polynomial Addition" Algorithm," *Comput. J.* 37 (1994), 795–813, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/poly-t.ps.gz⟩.

[34] M. Ward, "Assembler to C Migration using the FermaT Transformation System," *International Conference on Software Maintenance, 30th Aug–3rd Sept 1999, Oxford, England* (1999).

[35] M. Ward, "Reverse Engineering from Assembler to Formal Specifications via Program Transformations," *7th Working Conference on Reverse Engineering, 23-25th November*, Brisbane, Queensland, Australia (2000), ⟨http://www.dur.ac.uk/martin.ward/martin/papers/wcre2000.ps.gz⟩.

[36] M. Ward, "The Formal Transformation Approach to Source Code Analysis and Manipulation," *IEEE International Workshop on Source Code Analysis and Manipulation Florence, Italy, 10th November*, Los Alamitos, California, USA (2001).

[37] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/prog-spec.ps.gz⟩.

[38] M. Ward, "A Definition of Abstraction," *J. Software Maintenance: Research and Practice* 7 (Nov., 1995), 443–450, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/abstraction-t.ps.gz⟩.

[39] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," *IEEE Trans. Software Eng.* 22 (Sept., 1996), 665–686, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/sw-alg.ps.gz⟩.

[40] M. Ward & K. H. Bennett, "Formal Methods to Aid the Evolution of Software," *International Journal of Software Engineering and Knowledge Engineering* 5 (1995), 25–47, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/evolution-t.ps.gz⟩.

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

[42] Martin Ward, "Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations," *Science of Computer Programming, Special Issue on Program Transformation* 52 (2004), 213–255.

[43] M. Weiser, "Program slicing," *IEEE Trans. Software Eng.* 10 (July, 1984), 352–357.

[44] M. Weiser, "Programmers use slices when debugging," *Comm. ACM* 25 (July, 1984), 352–357.

[45] H. Yang & M. Ward, *Successful Evolution of Software Systems*, Artech House, 2003, ISBN 1-58053-349-3.

[46] E. J. Younger & M. Ward, "Inverse Engineering a simple Real Time program," *J. Software Maintenance: Research and Practice* 6 (1993), 197–234, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/eddy-t.ps.gz⟩.

[47] Xingyuan Zhang, Malcolm Munro, Mark Harman & Lin Hu, "Mechanized Operational Semantics of WSL," *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, Los Alamitos, California, USA (2002).

Appendix

In this Appendix we briefly describe the $\mathcal{META}$WSL notation used in the source code for the Simple_Slice transformation, and then give the source itself.

The @Simple_Slice_Test procedure tests the applicability conditions for the transformation and calls either @Pass (if the conditions are met), or @Fail (passing a string which describes the reason for the failure). The @Simple_Slice_Code function implements the transformation by calling the @Slice function on the currently selected statement.

@I	The currently selected item (statement, expression, condition etc.)
@GT	Returns the "generic type" of the given item (`Statement`, `Expression` etc.)
@ST	Returns the "specific type" (a `Statement` can be a `While`, `Assignment` etc.)
@V	Returns the value of the given item (eg the name of the variable)
@Type_Name	Returns a string giving the name of the type
@Make_Name	Convert a string to the internal representation of a name
@N_String	Convert a name to a string (the opposite of @Make_Name)
@Stat_Types	Returns the set of specific types of statements in the given item
@Cs	Returns a list of the components of the given item
@Size	Returns the number of components in the given item
@Make	Returns a new item with the given specific type, value and components
@Assigned	Returns the list of variables assigned in the given item
@Used	Returns the list of all variables appearing in the given item
I^n	Returns the nth component of item I

Table I. $\mathcal{META}$WSL functions used in the simple slicer

Table I describes the $\mathcal{META}$WSL functions used in the program. The first component of a **var** is the list of assignments, the first component of that is the first assignment, and the first component of an assignment is the assigned variable. So if I is the statement **var** $\langle v := e \rangle$: **S** **end** then I^1^1^1 is v, I^1^1^2 is e and I^2 is **S**.

```
proc @Simple_Slice_Test() ≡
  if @GT(@I) ≠ Statements ∧ @GT(@I) ≠ Statement
    then @Fail("Can only slice statements.")
  elsif @Stat_Types(@I) ⊆
        {Cond, D_If, While, Assignment, Assert, Skip, Abort}
        then foreach Lvalue do
                 if @ST(@I) ≠ Var_Lvalue
                   then @Fail("All assignments must be to simple variables.") fi od;
               foreach Statement do
                 if @ST(@I) = Assignment ∧ @Size(@I) > 1
                   then @Fail("Statement contains a parallel assignment.") fi od;
               if ¬@Failed? then @Pass fi
       else @Fail("Current item contains an item which cannot be sliced.") fi.;

proc @Simple_Slice_Code(Data) ≡
  var ⟨X := MAP("@Make_Name", @Split(Data)), R := ⟨⟩⟩ :
    print("Simple Slice, input variables are:  ",
          @Join(" ", MAP("@N_String", X)));
    R := @Slice(@I, @Make_Set(X));
```

```
    @Paste_Over(R[1]);
    print("Simple Slice, output variables are:  ",
        @Join(" ", MAP("@N_String", R[2]))) end.;

funct @Slice(I, X) ≡
  var ⟨R := ⟨⟩, new := ⟨⟩, newX := ⟨⟩⟩ :
    if @ST(I) = Statements
      then for S ∈ REVERSE(@Cs(I)) do
               R := @Slice(S, X);
               new := ⟨R[1]⟩ ++ new;
               X := R[2] od;
           R := ⟨@Make(Statements, ⟨⟩, new), X⟩
    elsif @ST(I) = Abort
        then R := ⟨I, ⟨⟩⟩
    elsif @GT(I) = Statement ∧ X ∩ @Assigned(I)) = ⟨⟩
        then R := ⟨@Skip, X⟩
    elsif @ST(I) = Assignment
        then R := ⟨I, (X \ @Assigned(I)) ∪ @Used(I)⟩
    elsif @ST(I) = Cond ∨ @ST(I) = D_If
        then for guard ∈ @Cs(I) do
                 R := @Slice(guard^2, X);
                 new := ⟨@Make(Guarded, ⟨⟩, ⟨guard^1, R[1]⟩)⟩ ++ new;
                 newX := newX ∪ @Variables(guard^1) ∪ R[2] od;
             R := ⟨@Make(@ST(I), ⟨⟩, REVERSE(new)), newX⟩
    elsif @ST(I) = While
        then var ⟨B := @Variables(I^1), S := I^2, newX := X⟩ :
               do R := @Slice(S, newX);
                  R[2] := R[2] ∪ newX ∪ B;
                  if R[2] = newX then exit(1) fi;
                  newX := R[2] od end;
             R := ⟨@Make(While, ⟨⟩, ⟨I^1, R[1]⟩), R[2]⟩
    elsif @ST(I) = Var
        then var ⟨v := I^1^1^1, e := I^1^1^2, S := I^2, newX := ⟨⟩⟩ :
               R := @Slice(I^2, X \ {@V(v)});
               S := R[1];
               newX := (R[2] \ {@V(v)}) ∪ ({@V(v)} ∩ X);
               if @V(v) ∈ R[2]
                 then R := ⟨fill Statement  var ⟨~?v := ~?e⟩ : ~?S end endfill,
                               newX ∪ @Used(e)⟩
                  else R := ⟨fill Statement  var ⟨~?v := BOTTOM⟩ : ~?S end endfill,
                        else ERROR("Unexpected type:  ", @Type_Name(@ST(I))) fi;
                 (R).
```

Received February 2005