The Formal Transformation Approach to Source Code Analysis and Manipulation

M. P. Ward
Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH, UK
`Martin.Ward@durham.ac.uk`

Abstract

In this paper we give a brief introduction to the foundations of WSL transformation theory and describe some applications to program slicing. We introduce some generalisations of traditional slicing, amorphous slicing and conditioned slicing which are possible in the framework of WSL transformations. One generalisation is "semantic slicing" which combines slicing and abstraction to a specification.

1 Introduction

In the development of methods for program analysis and manipulation it is important to start from a rigorous mathematical foundation. Without such a foundation, it is all too easy to assume that a particular transformation is valid, and come to rely upon it, only to discover that there are certain special cases where the transformation is not valid.

The following transformation was found in an article on program manipulation published in Communications of the ACM:

do $\mathbf{S}_1$ **od** is equivalent to **do** $\mathbf{S}_2$ **od**
if and only if
do $\mathbf{S}_1$ **od** is equivalent to **do** $\mathbf{S}_1$; $\mathbf{S}_2$ **od**

Here, $\mathbf{S}_1$ and $\mathbf{S}_2$ are any statements and the **do** ... **od** loops are unbounded or infinite loops which can only be terminated by executing an **exit** statement within the loop body. The statement **exit**(n) will terminate n enclosing **do** ... **od** loops.

The reverse implication is easily seen to be false: simply take $\mathbf{S}_2$ to be **skip**, then for any statement $\mathbf{S}_1$:

do $\mathbf{S}_1$ **od** is equivalent to **do** $\mathbf{S}_1$; **skip od**

while:

do $\mathbf{S}_1$ **od** is equivalent to **do skip od**

is not necessarily the case!

The forward implication looks more reasonable, and quite a useful transformation: it suggests that if we have two loops which implement the same program, then we can generate another program by combining the two loops. But consider these two programs:

```
do if $x \leqslant 0$ then exit fi;
   if $even(x)$ then $x := x - 2$ else $x := x + 1$ fi od
```

and

```
do if $x \leqslant 0$ then exit fi;
   $x := x - 1$ od
```

Both programs will terminate with $x = 0$ if they are started in a state with the integer $x \geqslant 0$, otherwise both loops will terminate immediately.

The interleaved program is:

```
do if $x \leqslant 0$ then exit fi;
   if $even(x)$ then $x := x - 2$ else $x := x + 1$ fi;
   if $x \leqslant 0$ then exit fi;
   $x := x - 1$ od
```

If $x = 1$ initially, then $x = 1$ at the end of the loop body. So the loop never terminates.

The point of this example is that it is very easy to invent a plausible new "transformation", and even base further research on it, before discovering that it is not valid. This underlines the importance of a sound mathematical foundation.

2 Refinement and Equivalence

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 are 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 (in which case, we put all the other states into $f(s)$).

If two programs have the same semantic function then they are *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 $f_2(s) \subseteq f_1(s)$. 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.

3 Transformation Theory

Our transformation theory 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;
6. Extend WSL with constructs for implementing program transformations (the result is called $\mathcal{M}\mathcal{E}\mathcal{T}\mathcal{A}$WSL);
7. Implement an industrial strength transformation engine in $\mathcal{M}\mathcal{E}\mathcal{T}\mathcal{A}$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: see [13,14,17]).

3.1 The Kernel Language

It turns out that for our kernel language we can do away with many familiar programming constructs: including assignments and **if** statements. We need just four primitive statements and three compound statements. Let $\mathbf{P}$ and $\mathbf{Q}$ be any logical formulae (technically, these are formulae of an infinitary first order logic) and $\mathbf{x}$ and $\mathbf{y}$ 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 without changing any variables, otherwise it aborts (we treat abnormal termination and nontermination 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:** $\mathsf{add}(\mathbf{x})$ adds the variables in $\mathbf{x}$ to the state space (if they are not already present) and assigns arbitrary values to them. The arbitrary values may be restricted to particular values by a subsequent guard;

4. **Remove variables:** $\mathsf{remove}(\mathbf{y})$ removes the variables in $\mathbf{y}$ from the state space (if they are present).

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

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

3.3 Semantic Refinement

A *state* is a collection of variables (the *state space*) with values assigned to them; thus a state is a function which maps from a finite set V of variables to a set $\mathcal{H}$ of values. There is a special extra state $\perp$ which is used to represent nontermination or error conditions. (It does not give values to any variables). States other than $\perp$ are called *proper states*. A state transformation f maps each initial state s in one state space, to the set of possible final states $f(s)$, which may be in a different state space. If $\perp$ is in $f(s)$ then, by definition, so is every other state, also $f(\perp)$ is the set of all states (including $\perp$).

Semantic refinement is defined in terms of these state transformations. A state transformation f is a refinement of a state transformation g if they have the same initial and final state spaces and $f(s) \subseteq g(s)$ for every initial state s. Note that if $\perp \in g(s)$ for some s, then by definition $g(s)$ includes every state, so $f(s)$ can be anything at all. In other words we can correctly refine an "undefined" program to do anything we please. If f is a refinement of g (equivalently, g is refined by f) we write $g \leq f$.

A *structure* for a logical language $\mathcal{L}$ consists of a set of values, plus a mapping between constant symbols, function symbols and relation symbols of $\mathcal{L}$ and elements, functions and relations on the set of values. If the interpretation of statement $\mathbf{S}_1$ under the structure M is refined by the interpretation of statement $\mathbf{S}_2$ under the same structure, then we write $\mathbf{S}_1 \leq_M \mathbf{S}_2$. A *model* for a set of sentences (formulae with no free variables) is a structure for the language such that each of the sentences is interpreted as true. If $\mathbf{S}_1 \leq_M \mathbf{S}_2$ for every model M of a countable set Δ of sentences of $\mathcal{L}$ then we write $\Delta \models \mathbf{S}_1 \ \leq \ \mathbf{S}_2$.

Here Δ is the set of assumptions about the logical symbols under which the refinement is valid.

3.4 Weakest Preconditions

Given any statement $\mathbf{S}$ and any formula $\mathbf{R}$ 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 $\mathbf{R}$:

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

where $(\mu X.\mathbf{S})^0 = \textsf{abort} = \{\textsf{false}\}$ and $(\mu X.\mathbf{S})^{n+1} = \mathbf{S}[(\mu X.\mathbf{S})^n/X]$ which is $\mathbf{S}$ with all occurrences of X replaced by $(\mu X.\mathbf{S})^n$. (In general, for statements $\mathbf{S}$, $\mathbf{T}$ and $\mathbf{T}'$, the notation $\mathbf{S}[\mathbf{T}'/\mathbf{T}]$ means $\mathbf{S}$ with $\mathbf{T}'$ instead of each $\mathbf{T}$).

The motivation for considering weakest preconditions is given in the next section.

3.5 Proof-Theoretic Refinement

Given two statements $\mathbf{S}_1$ and $\mathbf{S}_2$, and a formula $\mathbf{R}$, we can express the weakest preconditions $\mathrm{WP}(\mathbf{S}_1, \mathbf{R})$ and $\mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ as formulae in infinitary logic, as shown above. 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 [3] and Morgan [9,10] 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 [1,2] 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 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 3.1 Let $\mathbf{x}$ be a sequence of all variables assigned to in either $\mathbf{S}_1$ or $\mathbf{S}_2$ and let $\mathbf{x}'$ be a sequence of new variables the same length as $\mathbf{x}$. If the formulae $\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}')$ and $\mathrm{WP}(\mathbf{S}_1, \textsf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \textsf{true})$ are provable from the set Δ of sentences, then we say that $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ and write: $\Delta \vdash \mathbf{S}_1 \ \leq \ \mathbf{S}_2$.

It turns out that these two notions of refinement (semantic refinement and proof theoretic refinement) are the same. In other words:

Theorem 3.2 *For any statements $\mathbf{S}_1$ and $\mathbf{S}_2$, and any countable set Δ of sentences of $\mathcal{L}$:*

$$\Delta \models \mathbf{S}_1 \ \leq \ \mathbf{S}_2 \textit{ if and only if } \Delta \vdash \mathbf{S}_1 \ \leq \ \mathbf{S}_2$$

Proof: See [12].

These two equivalent definitions of refinement give rise to two very different methods for proving the correctness of refinements. Both methods are exploited in [12]—for example, weakest preconditions and infinitary logic are used to develop the induction rule for recursion and the recursive implementation theorem, while state transformations are used to prove the representation theorem.

Definition 3.3 *Two programs are* equivalent*, written* $\Delta \vdash \mathbf{S}_1 \approx \mathbf{S}_2$ *if and only if* $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$ *and* $\Delta \vdash \mathbf{S}_2 \leq \mathbf{S}_1$.

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:

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

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

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

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 $\mathsf{add}(\langle x' \rangle);\ [x' = e];\ \mathsf{add}(\langle x \rangle);\ [x = x']$ so the weakest precondition is:

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

The proof of this transformation is:

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

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 3.4 *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 $\Delta \vdash \mathbf{S} \leq \mathbf{S}'$.

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:

$$\begin{aligned}
&\Delta \vdash \textsf{\textbf{while B}}_1 \textsf{\textbf{ do S od}};\ \textsf{\textbf{while B}}_2 \textsf{\textbf{ do S od}}\\
&\qquad\qquad\qquad \approx \textsf{\textbf{while B}}_2 \textsf{\textbf{ do S od}}
\end{aligned}$$

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**)

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 [5] and simple local variables:

$$\begin{aligned}
&\textsf{\textbf{begin x}} := \mathbf{t}\colon \textsf{\textbf{S end}}\\
&\qquad =_{\mathrm{DF}} (\mathsf{add}(\mathbf{x});\ ([\mathbf{x} = \mathbf{t}];\ (\mathbf{S};\ \mathsf{remove}(\mathbf{x}))))
\end{aligned}$$

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

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 [9] calls the program [**false**] a "miracle", after Dijkstra's "Law of Excluded Miracles" [5]:

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

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). Fortunately, any WSL program with no explicit guard statements is non-null and obeys Dijkstra's law.

5 Operational Semantics

The correctness proofs of WSL transformations only look at the external behaviour of the programs. If you 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.

It turns out that this extra work is not necessary: 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. We then annotate the original program, adding assignments to seq at the appropriate places:

$\mathsf{annotate}(\mathbf{S}_1;\ \mathbf{S}_2)$
  $= \mathsf{annotate}(\mathbf{S}_1);\ \mathsf{annotate}(\mathbf{S}_2)$
$\mathsf{annotate}(\textbf{if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi})$
  $= \textbf{if B then } \mathsf{annotate}(\mathbf{S}_1) \textbf{ else } \mathsf{annotate}(\mathbf{S}_2) \textbf{ fi}$
$\mathsf{annotate}(v := e)$
  $= \mathsf{seq} := \mathsf{seq} +\!\!+ \langle\langle \text{"v"}, e\rangle\rangle;$

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 show that the transformation preserves operational semantics it is sufficient to show that it turns the annotated program $\mathsf{annotate}(\mathbf{S}_1)$ into a program 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

The notion of a program slice, originally introduced by Mark Weiser [18], is useful in program analysis and debugging. A slice of a program is taken with respect to a program point p and a variable x; the slice consists of all statements of the program that might affect the value of x at point p.

Initially we will consider the special case where p is the end point of the program, but we will generalise the variable x to a set X of variables. 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. Suppose program $\mathbf{S}$ maps state space V to W (we write this as $\mathbf{S} : V \to W$), then the effect of slicing $\mathbf{S}$ at its end point on the variables in X is to generate a program equivalent to $\mathbf{S};\ \mathsf{remove}(W \setminus X)$.

So one way to define a program slice is:

Definition 6.1 A *traditional program slice* of $\mathbf{S}$ on X is any program $\mathbf{S}'$ generated from $\mathbf{S}$ by deleting individual statements, such that:

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

Within this framework, a proof of correctness of an algorithm for program slicing (such as the algorithm for interprocedural slicing in [8]) is simply a proof of the validity of the transformation which deletes the statements in $\mathbf{S};\ \mathsf{remove}(W \setminus X)$ to create $\mathbf{S}'$.

The definition immediately suggests a possible generalisation: why restrict the transformations to deleting individual statements? Harman and Danicic [6] coined the term "amorphous program slicing" for a combination of slicing and transformation, but perhaps the term "semantic slice" is more expressive: we are slicing on the semantics of the program since we are allowing any operation which preserves the semantics. A traditional slice could then be called a "syntactic slice" since with traditional slicing we preserve the syntax of the program so far as is consistent with removing irrelevant statements.

Definition 6.2 A *semantic slice* of $\mathbf{S}$ on X is any program $\mathbf{S}'$ such that:

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

Note that there are many possible semantic slices for a program (but we would normally expect that the semantic slice should be no larger than the equivalent syntactic slice). This is because with any reverse engineering or program understanding process there are other constraints on the format of the abstract specification. See [15] and [16] for a discussion of the issues.

An intermediate operation between traditional syntactic slicing and semantic slicing restricts the transformations to preserve operational semantics using the technique in Section 5:

Definition 6.3 An *operational slice* of $\mathbf{S}$ on X is any program $\mathbf{S}'$ such that:

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

6.1 Slicing At Any Position

To slice at an arbitrary position in the program we need to preserve the sequence of values of 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. If $X = \{x_1, \dots, x_n\}$ is the set of variables we are interested in then we insert the statement:
$\mathsf{slice} := \mathsf{slice} +\!\!\!+ \langle\langle x_1, \dots, x_n\rangle\rangle$
at the point where we want to slice 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.

This process can be generalised to slice 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.

6.2 Conditioned Slicing

A *Conditioned Slice* of a program is a generalisation of a traditional slice where an extra condition is given for the initial state to satisfy. This additional condition can be used to simplify the program before applying a traditional slicing algorithm. Danicic et al [7] describe a tool called ConSIT, for slicing a program at a particular point, given that the initial state satisfies a given condition. Conditioned slicing is thus a generalisation of static slicing (where there are no conditions on the initial state) and dynamic slicing (slicing based on a particular initial state).

The slicing condition can be given in the form of `ASSERT` statements scattered through the program: [7] claim that these `ASSERT` statement are equivalent to a single condition on the initial state: but this seems to require that assertions can be formulae of *infinitary* logic. Fortunately, the assertion statements in WSL are already expressed in infinitary logic, so this is not a problem in our framework.

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 the traditional method.

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 [11] we provide a number of transformations for propagating assertions and eliminating dead code. Using weakest preconditions, for example, we can move an assertion (with the appropriate modification) backwards past any statement:

$$\Delta \vdash \mathbf{S};\ \{\mathbf{Q}\}\ \approx\ \{\mathrm{WP}(\mathbf{S}, \mathbf{Q})\};\ \mathbf{S}$$

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

Similarly, an assertion can be moved out of a loop:

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

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

Again, a generalisation is suggested: why restrict ourselves to the assertion moving and dead code removal transformations? A conditioned semantic slice can be defined as:

Definition 6.4 A *conditioned semantic slice* of $\mathbf{S}$ on X is any program $\mathbf{S}''$ such that:

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

where $\mathbf{S}'$ is constructed from $\mathbf{S}$ by inserting assertions and assignments to slice wherever needed.

7 Slicing Example

The following WSL program is a translation of the C program in [4]:

```
$i := 1$;
posprod := 1;
negprod := 1;
possum := 0;
negsum := 0;
while $i \leqslant n$ 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 $\geqslant$ negsum
             then possum := 0
             else negsum := 0 fi;
           if posprod $\geqslant$ negprod
             then posprod := 1
             else negprod := 1 fi fi;
  $i := i + 1$ od;
if possum $\geqslant$ negsum
  then sum := possum
  else sum := negsum fi;
if posprod $\geqslant$ 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. We also append the remove statement:

remove(i, posprod, negprod, possum, negsum, n, a, test0)

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

The resulting syntactic slice is:

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

With semantic slicing we can do much more. For a start, the test $a > 0$ is redundant because of the assertion. Also negsum is zero throughout and possum $\geqslant 0$ throughout (since it is initialised to zero and only modified by having positive numbers added). So a possible semantic slice is:

```
$i := 1$;
possum := 0;
while $i \leqslant n$ do
  $a :=$ input[$i$];
  $\{a > 0\}$;
  possum := possum + $a$;
  $i := i + 1$ od;
sum := possum;
remove($i$, posprod, negprod, possum, negsum, $n$, $a$, test0)
```

But we can do even more than this. If we first move the assertion out of the loop, then the loop itself can be collapsed to a reduce operation over the input array:

```
$\{\forall i. \, 1 \leqslant i \leqslant n \Rightarrow$ input[$i$] $> 0\}$;
sum := +/input[$1 \,..\, n$];
remove($i$, posprod, negprod, possum, negsum, $n$, $a$, test0)
```

The result is a concise specification of the final value of sum under the given slicing condition.

8 Conclusion

In this paper we have given a brief introduction to the foundations of program transformation theory in WSL and described some applications to program slicing which existing slicing algorithms. Traditional slicing, which is restricted to deleting irrelevant statements has the advantage of a unique solution 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.

References

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

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

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

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

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

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

[7] 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).

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

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

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

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

[12] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," Durham University, Technical Report, 1994, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/foundation2-t.ps.gz⟩.

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

[14] 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/~dcs0mpw/martin/papers/wcre2000.ps.gz⟩.

[15] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/prog-spec.ps.gz⟩.

[16] M. Ward, "A Definition of Abstraction," *J. Software Maintenance: Research and Practice* 7 (Nov., 1995), 443–450, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/abstraction-t.ps.gz⟩.

[17] 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/~dcs0mpw/martin/papers/evolution-t.ps.gz⟩.

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