Slicing the SCAM Mug: A Case Study in Semantic Slicing

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 describe an improved formalisation of slicing in WSL transformation theory and apply the result to a particularly challenging slicing problem: the SCAM mug [1]. We present both syntactic and semantic slices of the mug program and give semantic slices for various generalisations of the program. Although there is no algorithm for constructing a minimal syntactic *slice, we show that it is possible, in the WSL language, to derive a minimal* semantic *slice for any program and any slicing criteria.*

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 [17] as a debugging technique [18], and has since proved to have applications in testing, parallelization, integration, software safety, program understanding and software maintenance. Survey articles by Binkley and Gallagher [2] and Tip [10] include extensive bibliographies.

In [12] a formalisation of slicing in terms of program transformations was proposed. In this paper we present an improved formalisation and apply it to a particularly challenging slicing problem: the SCAM mug [1]. The WSL formulation of slicing immediately lends itself to several extensions: simply by relaxing some of the constraints in the definition and removing restrictions on the allowed transformations.

Weiser defined a program slice **S** as a *reduced, executable program* obtained from a program **P** by removing statements, such that **S** replicates part of the behavior of **P**. A *program transformation* is any operation on a program which generates a semantically equivalent program. A slice is not generally a transformation of the original program since 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: then 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. Consider this simple example:

$x := y + 1;$
$y := y + 4;$
$x := x + z$

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

$x := y + 1;$
$x := x + z$

These two programs are not equivalent, but if we modify both programs by appending a **remove**(y) statement then the resulting programs *are* equivalent.

To be precise, let $\mathbf{S}_1$ be the program:

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

and let $\mathbf{S}_2$ be the program:

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

The initial state space for $\mathbf{S}_1$ and $\mathbf{S}_2$ is $\{x, y, z\}$ while the final state space is $\{x\}$ (the **remove** statement ensures that y cannot appear in the final state space). Both programs set x to the value $y+1+z$, so the two programs are equivalent.

So much for slicing at the end of a program. Suppose we want to slice on the value of i at the top of the **while** loop body in this program:

$i := 0;\ s := 0;$
while $i < n$ **do**
  $s := s + i;$
  $i := i + 1$ **od**;
$i := 0$

Slicing on i at the end of the program would give $i := 0$ as a valid slice: which is not what we wanted! So we need some way to get the sequence of values taken on by i at the top of the loop to be "carried" to the end of the program. A simple way to do this is to add a new variable, slice, which records this sequence of values:

$i := 0;\ s := 0;$
while $i < n$ **do**
  $\mathsf{slice} := \mathsf{slice} +\!\!+ \langle i \rangle;$
  $s := s + i;$
  $i := i + 1$ **od**;
$i := 0;$

where the statement $\mathsf{slice} := \mathsf{slice} +\!\!+ \langle i \rangle$ appends the value of i to the end of the sequence stored in slice. By the end of the program, slice contains a list of all the values taken on by i at each iteration of the loop.

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

$i := 0;$
while $i < n$ **do**
  $\mathsf{slice} := \mathsf{slice} +\!\!+ \langle i \rangle;$
  $i := i + 1$ **od**;
remove(i, s, n)

which yields the sliced program:

$i := 0;$
while $i < n$ **do**
  $i := i + 1$ **od**

This approach easily generalises to slicing on a set of locations with the same or a different set of variables of interest at each location.

2. Slicing as a Program Transformation

The WSL language has been described elsewhere [11, 13,16], so will not be described in detail here. Key points to note are that WSL is based on *infinitary* logic: which means that formulae in WSL programs can be infinitely long. The weakest precondition of a WSL program $\mathbf{S}$, for a given postcondition, $\mathbf{R}$, denoted $\mathrm{WP}(\mathbf{S}, \mathbf{R})$, is the weakest condition on the initial state such that if $\mathbf{S}$ is started in a state which satisfies $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ then it is guaranteed to terminate and every possible final state will satisfy $\mathbf{R}$. If the postcondition $\mathbf{R}$ is defined as an infinitary logic formula, then $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ can be defined as an infinitary logic formula. Hence, the WP for one program can be used as an assertion or condition in another program.

WSL includes loops of the form **do** ... **od** which can only be terminated via an **exit**(n) statement. The statement **exit**(n) (in which n is an integer, not a variable or expression) causes the immediate termination of n enclosing nested **do** ... **od** loops. A statement in which each **exit**(n) is enclosed in at least n nested loops is called a *proper sequence*: such a statement cannot terminate an enclosing loop.

2.1. Reduction

To give a formal definition of slicing in WSL we need to define a *reduction* of a program. Informally, this relates a program with the result of replacing certain statements 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 such that there is an **exit**$(n+k)$ within $k \geqslant 0$ nested **do** ... **od** loops in $\mathbf{S}$, then:

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

(In other words, if execution of $\mathbf{S}$ could result in the termination of at most n enclosing loops, then $\mathbf{S}$ can be reduced to **exit**(n)).

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:

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

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

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

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:

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

The reduction relation does not allow deletion of a **skip** statement from a sequence of statements. This is so that the position of each subcomponent of the reduced program is the same as the corresponding subcomponent in the original program. This relationship makes it easer to prove the correctness of slicing algorithms: deleting the extraneous **skip** statements is a trivial additional step. In what follows we will omit the extra **skip** statements when the relationship between the original and sliced programs is clear.

Three important properties of the reduction relation are:

Lemma 2.1 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 2.2 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 2.3 The *Replacement Property*: If any component of a program is replaced by a reduction, then the result is a reduction of the whole program.

2.2. Syntactic Slicing

In [12] a syntactic slice was defined as any reduction of the program which is also a *refinement* of the program. This definition allows a program slicer to delete loops which do not affect the variables in the slicing criteria without having to prove termination of the loop (most slicing researchers allow deletion of nonterminating code as a valid slice). 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.

Consider the following nondeterministic program which we want to slice on the final value of x:

$x := 1$;
while $n > 1$ **do**
  if even?(n) **then** $n := n/2$
      else $n := 3 * n + 1$ **fi od**;
if true $\rightarrow x := 1$
$\square$ **true** $\rightarrow x := 2$ **fi**

The **while** loop clearly does not affect x, so we would like to delete it from the slice. But if we are insisting that the slice be *equivalent* to the original program (on x), then we have to prove that the loop terminates for all n before we can delete it. The loop generates the Collatz sequence and it is an open question as to whether the sequence always reaches 1. (The problem was first posed by L. Collatz in 1937 [7,8]).

Allowing any refinement as a valid slice (as in [12]) would allow us to delete the **while** loop, but would also allow us to delete the **if** statement, giving $x := 1$ as a valid slice. If the slice is being determines as part of a program analysis or comprehension task, then the programmer might (incorrectly) conclude that the original program assigns the value 1 to x whenever it terminates.

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

Definition 2.4 A *semi-refinement* of $\mathbf{S}$ is any program $\mathbf{S}'$ such that

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

The semi-refinement relationship is denoted $\Delta \vdash \mathbf{S} \preccurlyeq \mathbf{S}'$.

For any statement $\mathbf{S}$ and formula $\mathbf{R}$ the *weakest precondition* $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ is a formula which is true on those initial states for which $\mathbf{S}$ is guaranteed to terminate in a state which satisfies $\mathbf{R}$ (See [12] for the definition of WP on WSL). So $\mathrm{WP}(\mathbf{S}, \textbf{true})$ is true on precisely those initial states for which $\mathbf{S}$ is guaranteed to terminate. Hence, the assertion $\{\mathrm{WP}(\mathbf{S}, \textbf{true})\}$ is a **skip** when $\mathbf{S}$ terminates and **abort** when $\mathbf{S}$ may not terminate.

If $\Delta \vdash \mathbf{S} \preccurlyeq \mathbf{S}'$ then $\mathbf{S}'$ must be equivalent to $\mathbf{S}$ when $\mathbf{S}$ terminates, but $\mathbf{S}'$ can do anything at all when $\mathbf{S}$ does not terminate. In particular $\mathbf{S}'$ can be equivalent to **skip** when $\mathbf{S}$ does not terminate.

We define a syntactic slice of $\mathbf{S}$ on X to be any reduction of $\mathbf{S}$ which is also a semi-refinement:

Definition 2.5 A *Syntactic Slice* of $\mathbf{S}$ on X is any program $\mathbf{S}'$ such that $\mathbf{S}' \sqsubseteq \mathbf{S}$ and

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

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

We can extend this definition to slicing at arbitrary points in the program by adding assignments to a new slice variable as discussed in Section 1.

2.3. Semantic Slicing

The definition of a syntactic slice immediately suggests a generalisation: why restrict the semi-refinements to deleting statements? Or, to put it another way, why not drop the requirement that $\mathbf{S}' \sqsubseteq \mathbf{S}$?

Harman and Danicic [4,5] 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 2.6 A *semantic slice* of $\mathbf{S}$ on X is any program $\mathbf{S}'$ such that:

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

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, [14, 15] 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 [14] and [15] for a discussion of the issues.

3. The SCAM Mug

The following program was "published" on the side of a mug distributed to delegates at the first SCAM (Source Code Analysis and Manipulation) workshop in 2001 [1]. It is based on an example in [3]:

```
while (p(i))
{        if (q(c))
                   { x := f();
                     c := g(); }
         i := h(i)
}
```

The problem is to determine which lines do not affect the value of `x`.

The mug "solves" the problem when filled with hot java (or other hot liquid) and gives the answer that the assignment `c := g()` does not affect `x`. This is because if at any point the assignment `x := f()` is executed, then from that point on it does not matter whether the `if` statement is taken or not. In other words, the value of `c` *from that point on* is irrelevant to the final value of `x` (further assignments to `x` will give it the same value). Note that the functions `f()` and `g()` are assumed not to have side-effects or reference any of the variables `i`, `c` or `x`, so they return the same value on each call. The other functions (`p`, `q` and `h`) are also assumed to be "pure functions" (with no side effects and a returned value which only depends on the parameters).

A WSL translation of the program (called MUG_0) is:

$$
\begin{array}{l}
\textsf{while}\ p?(i)\ \textsf{do} \\
\quad \textsf{if}\ q?(c) \\
\qquad \textsf{then}\ x := f; \\
\qquad\qquad c := g\ \textsf{fi}; \\
\quad i := h(i)\ \textsf{od}
\end{array}
$$

where we have used the constants f and g for the values returned by `f()` and `g()`.

The traditional dataflow-based slicing algorithms (such as [6]) will observe that there is a data dependency between the final value of x and the assignment $x := f$. There is a control dependency between the test $q?(c)$ and the assignment $x := f$ and there is a data dependency between $c := g$ and $q?(c)$. Similarly, there is a control dependency between $x := f$ and $p?(i)$ and a data dependency between $i := h(i)$ and $p?(i)$. Table 1 summarises the dependencies.

$$
\begin{aligned}
x := f &\xrightarrow{\text{ctrl}} q?(c)\\
q?(c) &\xrightarrow{\text{data}} c := g\\
x := f &\xrightarrow{\text{ctrl}} p?(i)\\
p?(i) &\xrightarrow{\text{data}} i := h(i)
\end{aligned}
$$

Table 1. Control and data dependencies in MUG_0

If the algorithm simply follows all dependencies in order to determine what statements to include in the slice, then it will conclude that *all* the statements appear to affect x. For example, the FermaT syntactic slicer will return the whole program when asked to slice on the final value of x since it uses just such a dataflow-based algorithm.

It should not be surprising that the dataflow algorithm sometimes produces a less than minimal slice, since the task of determining a minimal slice is noncomputable in the general case: so there can be *no* algorithm which *always* returns a minimal slice.

Our aim in this section is to illustrate the power of FermaT transformations by showing how a few simple transformations can firstly give a very simple *semantic* slice, and then using this result to derive a minimal *syntactic* slice. We then discuss various generalisations of the mug program.

3.1. Transformation and Slicing

First, we note that if $q?(c)$ is initially true, then the first iteration of the loop will set x to f, and this will be the final value of x (the only thing that is assigned to x is the constant value f). So the first thing we want to do is to unroll the first iteration of the loop:

if $p?(i)$
  then if $q?(c)$
        then $x := f;\ c := g$ **fi**;
      $i := h(i)$;
      while $p?(i)$ **do**
        if $q?(c)$
          then $x := f;\ c := g$ **fi**;
        $i := h(i)$ **od fi**

Expand the **if** $q?(c)\ldots$ statement forwards over the next two statements:

if $p?(i)$
  then if $q?(c)$
        then $x := f;\ c := g$;
          $i := h(i)$;
          while $p?(i)$ **do**
            if $q?(c)$
              then $x := f;\ c := g$ **fi**;
            $i := h(i)$ **od**
        else $i := h(i)$;
          while $p?(i)$ **do**
            if $q?(c)$
              then $x := f;\ c := g$ **fi**;
            $i := h(i)$ **od fi fi**

In the second **while** loop, $\neg q?(c)$ is invariant over the loop ($\neg q?(c)$ is true initially, so the assignment to c will never be executed). So we can simplify the loop body:

if $p?(i)$
  then if $q?(c)$
        then $x := f;\ c := g$;
          $i := h(i)$;
          while $p?(i)$ **do**
            if $q?(c)$
              then $x := f;\ c := g$ **fi**;
            $i := h(i)$ **od**
        else $i := h(i)$;
          while $p?(i)$ **do**
            $i := h(i)$ **od fi fi**

Now we can apply syntactic slicing to the final value of x:

if $p?(i)$
  then if $q?(c)$
        then $x := f$;
          $c := g$;
          $i := h(i)$;
          while $p?(i)$ **do**
            if $q?(c)$
              then $x := f;\ c := g$ **fi**;
            $i := h(i)$ **od fi fi**

The Constant_Propagation transformation will determine that the second assignment to x is redundant:

if $p?(i)$
  then if $q?(c)$
        then $x := f$;
          $c := g$;
          $i := h(i)$;
          while $p?(i)$ **do** $i := h(i)$ **od fi fi**

Another syntactic slice will delete all the code after the first assignment to x:

if $p?(i)$ **then if** $q?(c)$ **then** $x := f$ **fi fi**

Align_Nested_Statements will simplify this to the program MUG_1:

if $p?(i)\ \wedge\ q?(c)$ **then** $x := f$ **fi**

The result is a nice, compact semantic slice. An equivalent semantic slice (in fact, a *minimal* semantic slice, if we are counting statements) can be achieved by transforming to a single specification statement:

$x := x'.((p?(i) \wedge q?(c) \Rightarrow x' = f)$
$\qquad \wedge (\neg(p?(i) \wedge q?(c)) \Rightarrow x' = x))$

A different minimal semantic slice uses a conditional expression:

$x :=$ **if** $p?(i) \wedge q?(c)$ **then** f **else** x **fi**

3.2. A Minimal Syntactic Slice

To get a minimal syntactic slice of the mug problem we start as before by unfolding the **while** loop and expanding the **if** statement in MUG_0 to give:

```
if p?(i)
  then if q?(c)
         then x := f; c := g;
              i := h(i);
              while p?(i) do
                if q?(c)
                  then x := f; c := g fi;
                i := h(i) od
         else i := h(i);
              while p?(i) do
                if q?(c)
                  then x := f; c := g fi;
                i := h(i) od fi fi
```

Within the second **while** loop, $\neg q?(c)$ is invariant as before, so we can make any changes we like to the body of the statement **if** $q?(c)$ **then** ... **fi**. We choose to delete the assignment $c := g$. Apply constant propagation to the first assignment $c := g$ (outside the first loop) to remove all references to c. The first assignment can then be deleted (since we are not interested in the final value of c):

```
if p?(i)
  then if q?(c)
         then x := f;
              i := h(i);
              while p?(i) do
                if q?(g)
                  then x := f fi;
                i := h(i) od
         else i := h(i);
              while p?(i) do
                if q?(c)
                  then x := f fi;
                i := h(i) od fi fi
```

The statement **if** $q?(g)$ **then** $x := f$ **fi** is redundant since at this point, x already has the value f. So we can replace it by the equally redundant statement **if** $q?(c)$ **then** $x := f$ **fi**. The two **while** loops are identical and can be taken out of the enclosing **if** statement, along with the statement $i := h(i)$:

```
if p?(i)
  then if q?(c)
         then x := f fi;
       i := h(i);
       while p?(i) do
         if q?(c)
           then x := f fi;
         i := h(i) od fi
```

Finally, we can roll up the loop to get MUG_2:

```
while p?(i) do
  if q?(c)
    then x := f fi;
  i := h(i) od
```

This is a valid syntactic slice of MUG_0 on x.

One way to prove that a syntactic slice is minimal is to prove that every smaller reduction of the program is *not* a semi-refinement of the program. To prove that a program is not a semi-refinement it is sufficient to give an instantiation of the function and predicate symbols, and an initial state, such that the original program terminates in one state but the proposed slice either does not terminate or terminates in a different state.

For MUG_0 there are two cases to consider: deleting the assignment $x := f$ and/or deleting the assignment $i := h(i)$ (deleting any other statement will also delete one or both of these).

Let $p?(i)$ be the condition $i > 0$, let $q?(c)$ be the condition **true**, let $h(i)$ be $i - 1$ and let $f = 1$. Under this instantiation, MUG_0 is:

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

Suppose $i = 1$ and $x = 0$ initially, then MUG_0 will terminate and set x to 1.

For the first case, any proposed slice which omits the assignment $x := f$ cannot change x at all, so cannot be valid.

For the second case (omitting $i := h(i)$), the proposed slice:

```
while p?(i) do
  if q?(c)
    then x := f fi od
```

does not terminate on the given initial state, so also cannot be valid. These two cases cover all the possible reductions of the original program, so our syntactic slice is indeed the minimal slice.

Note that there are several plausible definitions of a minimal syntactic slice:

1. Deleting any statement from the slice does not result in a valid slice;
2. Deleting any *set of statements* from the slice does not result in a valid slice;
3. Any other valid slice has at least as many statements as this one.

A minimal slice is not necessarily unique (for any of the definitions). Consider the program:

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

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

$$x := 1;\ x := x + 2 \qquad \text{and} \qquad x := 2;\ x := x + 1$$

(These slices are minimal according to all three suggested definitions of minimality).

Although of theoretical interest, demanding that the slices be minimal is too restrictive a requirement to place on a slicing algorithm. This is because an algorithm for finding minimal slices can be converted into an algorithm for solving the halting problem (see [17]). The halting problem is non-computable, hence the minimal slicing problem is also non-computable.

Weiser's proof, with slight modifications, can be applied to syntactic slicing in WSL to show that there is no algorithm for computing minimal syntactic slices in WSL. This proof only applies to *syntactic* slices: as we will show in Section 5 for WSL it is always possible to construct a minimal *semantic* slice (the construction is not an "algorithm" in the usual sense because the output may be infinitely large: even though the output is a single statement, the statement may contain infinitary formulae).

4. The Generalised Mug Problem

A generalisation of the mug problem is the following:

while $p(i)$ **do**
  if $q?(c, i)$
    then $x := f;\ x := g(i)$ **fi**;
  $i := h(i)$ **od**

If at some point in the course of execution $q?(c, i)$ becomes true, then the assignment $x := f$ will occur. All subsequent iterations are redundant since the only way the can affect x is by assigning the value it already has. So in this case, our first step is to split the **while** loop on the condition $\neg q?(c, i)$.

The Loop_Merging transformation states that for any condition $\mathbf{B}'$:

$$\Delta \vdash \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od}$$
$$\approx \textbf{ while } \mathbf{B} \wedge \mathbf{B}' \textbf{ do } \mathbf{S} \textbf{ od};\ \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od}$$

If we split the loop on the condition $\neg q?(c, i)$, then the first iteration of the second loop will assign to x and c. So split the loop and then unroll the first step of the second loop:

while $p(i) \wedge \neg q(c, i)$ **do**
  if $q?(c, i)$
    then $x := f;\ x := g(i)$ **fi**;
  $i := h(i)$ **od**;
$\{p(i) \Rightarrow q(c, i)\}$;
if $p(i)$
  then $\{q(c, i)\}$;
      if $q(c, i)$
        then $x := f;\ c := g(i)$ **fi**;
      $i := h(i)$;
      while $p(i)$ **do**
        if $q(c, i)$
          then $x := f;\ c := g(i)$ **fi**;
        $i := h(i)$ **od fi**

The assertions come from the fact that on termination of the first loop we must have $\neg(p(i) \wedge \neg q(c, i))$ which is equivalent to $p(i) \Rightarrow q(c, i)$. We can use the assertions to simplify the program:

while $p(i) \wedge \neg q(c, i)$ **do**
  $i := h(i)$ **od**;
if $p(i)$
  then $x := f;\ c := g(i)$;
      $i := h(i)$;
      while $p(i)$ **do**
        if $q(c, i)$
          then $x := f;\ c := g(i)$ **fi**;
        $i := h(i)$ **od fi**

Now apply Constant_Propagation to delete the second assignment to x and then Syntactic_Slice on x will give:

while $p(i) \wedge \neg q(c, i)$ **do**
  $i := h(i)$ **od**;
if $p(i)$ **then** $x := f$ **fi**

As before, this can be transformed into a single specification statement. First we need to know the value of i after the first loop.

Define the formula $H(i, j, c)$ as:

$$\exists n. (j = h^n(i) \wedge \forall m < n. (p(h^m(i)) \wedge \neg q(h^m(i))) \wedge (\neg p(h^n(i)) \vee q(h^n(i))))$$

Here $H(i, j, c)$ is true precisely when j is the final value given to i when the loop terminates. So the **while** loop is equivalent to the specification statement $i := i'.H(i, i', c)$. This gives the following semantic slice:

$i := i'.H(i, i', c)$;
if $p(i)$ **then** $x := f$ **fi**

which simplifies to the single specification statement:

$$x := x'.((\forall i'.\,(H(i, i', c) \Rightarrow p(i')) \Rightarrow x' = f) \\ \wedge\ (\neg\forall i'.\,(H(i, i', c) \Rightarrow p(i')) \Rightarrow x' = x))$$

Using a conditional expression, we have this equivalent statement:

$x :=$ **if** $\forall i'.\,(H(i, i', c) \Rightarrow p(i'))$ **then** f **else** x **fi**

5. Further Generalisations

The generalised mug problem has the following features:

- The program is deterministic;
- The variable x is assigned a constant value in one or more places.

By transforming the program we were able to determine the condition under which x is assigned, and therefore transform the whole program into a conditional assignment.

Suppose **S** is a deterministic program which contains one or more assignments of constant values to x, of the form $x := e_i$, where $e_1, \ldots, e_n$ are constants.. The precondition such that x is assigned the value e_i is simply $\mathrm{WP}(\mathbf{S}, x = e_i)$. So a valid semantic slice for the whole program is:

$x :=$ **if** $\mathrm{WP}(\mathbf{S}, x = e_1)$ **then** e_1
elsif ...
elsif $\mathrm{WP}(\mathbf{S}, x = e_n)$ **then** e_n
else x **fi**

Note that if all the weakest preconditions are false for a given initial state, then either none of the assignments to x is executed when **S** is started in that state (in which case x has its initial value) or **S** does not terminate when started in that state (in which case, $x := x$ is a valid semantic slice).

Finally, for an arbitrary **S** (deterministic or nondeterministic) we can use the *Representation Theorem* to derive a general semantic slice.

Theorem 5.1 *The Representation Theorem*
Let $\mathbf{S}\colon V \to V$, be any statement and let $\mathbf{x}$ be a list of all the variables in V. Then **S** is equivalent to:

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

If **S** is null-free (which is guaranteed for all WSL statements in language levels above the kernel level) then $\mathrm{WP}(\mathbf{S}, \textsf{false})$ is false, and the initial guard is redundant. For such statements we can transform the specification statement to show that **S** is equivalent to:

$\{\mathrm{WP}(\mathbf{S}, \textsf{true})\};\ \mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$

Then, by the definition of semi-refinement:

$$\mathbf{S} \preccurlyeq \mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$$

This is clearly a *minimal* semantic slice (counting statements) since it only contains a single statement, and by definition no WSL program can be smaller than a single statement. (It is not necessarily minimal if we are counting the total number of symbols: if statement **S** contains loops or recursion then the formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ is infinitely long!) So we have:

Theorem 5.2 *The Minimal Semantic Slice Theorem*
Let $\mathbf{S}\colon V \to V$, be any null-free statement and let $\mathbf{x}$ be a list of any subset of the variables in V. Then the statement $\mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$ is a valid semantic slice of **S** on the set $\textsf{vars}(\mathbf{x})$.

This may appear to contradict Weiser's theorem on the non-computability of minimal slices, but Weiser's theorem only applies to algorithms for computing minimal *syntactic* slices. The construction of $\mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'))$ from **S**, while being well defined, is not an algorithm in the usual sense because the formula $\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}')$ may be infinitely long. (In fact, it *will* be infinite whenever **S** contains any loops or recursion). An infinite specification statement is not directly executable, so this result is only practical for statements which contain no loops or recursion, but it does show that *no semantic slice need be larger than a single statement*.

For WSL programs with no loops or recursion (and where all the formulae are finite). Theorem 5.2 *does* give an algorithm for computing a minimal semantic slice on any given slicing criterion.

5.1. Minimal Semantic Slicing Example

Tip [9] suggested the computation of slices using a mixture of slicing and transformation in which a program is translated to an intermediate representation (IR), the IR is transformed and optimised (while maintaining a mapping back to the source text), and slices are extracted from the source text. Tip gave the following example program (which we have translated into WSL):

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

where we are interested in slicing on the final value of y. Tip suggests that a slicing algorithm which is capable of merging the two conditionals to produce the following optimised program:

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

ought to be capable of producing the following slice:

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

With semantic slicing we can, of course, produce a smaller slice. Theorem 5.2 gives the following slice:

$$y := y'.(\neg \mathrm{WP}(\textbf{if } p = q \textbf{ then } x := 18 \textbf{ else } x := 17 \textbf{ fi}; \textbf{if } p \neq q \textbf{ then } y := x \textbf{ else } y := 2 \textbf{ fi}, y \neq y'))$$

This simplifies to:

$$y := y'.(\neg \mathrm{WP}(\textbf{if } p = q \textbf{ then } x := 18 \textbf{ else } x := 17 \textbf{ fi}, (p \neq q \Rightarrow x \neq x') \wedge (p = q \Rightarrow 2 \neq y')))$$

which in turn simplifies to:

$$y := y'.(\neg((p = q \Rightarrow 2 \neq y') \wedge (p \neq q \Rightarrow 17 \neq y')))$$

pwhich is equivalent to:

$$y := y'.((p = q \Rightarrow 2 = y') \wedge (p \neq q \Rightarrow 17 = y'))$$

This can be expressed as a simple assignment on a conditional expression:

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

6. Slicing in FermaT

The FermaT transformation system implements most of the transformations described in this paper, including syntactic slicing and constant propagation. Semantic slices can be constructed by applying a sequence of syntactic slicing and transformation steps: research is currently underway to develop a general set of heuristics (in the form of a $\mathcal{M}\varepsilon\mathcal{T}\mathcal{A}$WSL program) which will automatically apply an appropriate sequence of transformation and slicing steps to generate a semantic slice which is suitable for program analysis and reverse engineering.

FermaT is available under the GNU GPL (General Public Licence) from the following web sites:

http://www.dur.ac.uk/~dcs6mpw/fermat.html
http://www.cse.dmu.ac.uk/~mward/fermat.html

FermaT, together with an extended transformation catalogue, forms the core of the commercial FERMAT Migration and Comprehension Workbench produced by Software Migrations Ltd. The Workbench includes a number of tools for assembler analysis, comprehension and migration including dataflow analysers and program slicing. The tools have been used in several migration projects from assembler to C and COBOL, contact sales@smltd.com for information.

7. Conclusion

In this paper we have described an improved formalisation of program slicing in terms of WSL program transformation theory. We have applied the result to a particularly challenging slicing problem and found that by applying FermaT transformations and syntactic slicing we can produce minimal semantic and syntactic slices for the mug program. We describe various generalisations for the mug program, culminating in a construction for a minimal semantic slice for *any* WSL program which consists of exactly one statement. In the case of WSL programs with finite formulae and no iteration or recursion, this construct is finite, so the theorem provides an *algorithm* for a minimal semantic slice.

8. References

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

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

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

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

[5] 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).

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

[7] J. C. Lagarias, "The $3x + 1$ Problem and Its Generalizations," *American Mathematical Monthly* 92 (1985), 3–23, ⟨http://www.cecm.sfu.ca/organics/papers/lagarias/⟩.

[8] B. Thwaites, "Two Conjectures, or How to Win £1100," *Mathematical Gazette* 80 (1996), 35–36.

[9] F. Tip, "Generation of Program Analysis Tools," Cantrum voor Wiskunde en Informatica, PhD Thesis, Amsterdam, 1995.

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

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

[12] 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).

[13] M. Ward, "Program Slicing via FermaT Transformations," *COMPSAC 2002, 26th Annual International Computer Software and Applications Conference, Oxford, England, 26th-29th August*, Los Alamitos, California, USA (2002).

[14] 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⟩.

[15] 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⟩.

[16] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," *IEEE Trans. Software Eng.* 22 (Sept., 1996), 665–686, ⟨http://www.dur.ac.uk/∼dcs0mpw/martin/papers/sw-alg.ps.gz⟩.

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

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