Program Slicing

Martin Ward

Reader in Software Engineering

martin@gkc.org.uk

Software Technology Research Lab
De Montfort University

Program Slicing

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.

Program Slicing

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 as a debugging technique [Weiser 1984], and has since proved to have applications in testing, parallelisation, integration, software safety, program understanding and software maintenance.

Program Slicing

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 as a debugging technique [Weiser 1984], and has since proved to have applications in testing, parallelisation, integration, software safety, program understanding and software maintenance.

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 behaviour of **P**.

Slicing as a Program Transformation

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.

Slicing as a Program Transformation

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.

Slicing *can* be formalised as a program transformation on a modification of the original program.

Slicing as a Program Transformation

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.

Slicing *can* be formalised as a program transformation on a modification of the original program.

If we remove the variables we are not interested in from the state space (for both programs) then the final state space contains only the variables of interest, so the two programs are equivalent.

Slicing as a Program Transformation

Recall that the semantics of a WSL program is a function which maps the initial state to the set of possible final states.

A *state* is a function which maps each variable in the *state space* to a value.

The WSL kernel statement $\mathbf{remove}(\mathbf{y})$ removes variables from the state space.

Slicing Example

Slicing on the final value of x:

$$
\begin{array}{lcl}
x := y + 1; & \qquad\qquad & x := y + 1; \\
y := y + 4; & & \\
x := x + z & & x := x + z
\end{array}
$$

These two programs are *not* equivalent: because the final value of y is different.

Slicing Example

Slicing on the final value of x:

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

These two programs are *not* equivalent: because the final value of y is different. But if we remove y from the state space:

$x := y + 1;$	$x := y + 1;$
$y := y + 4;$	
$x := x + z;$	$x := x + z$
remove$(\langle y \rangle)$	**remove**$(\langle y \rangle)$

then the resulting programs are equivalent.

Slicing Example

sum $:= 0$;

prod $:= 1$;

$i := 1$;

while $i \leqslant n$ **do**

  sum $:=$ sum $+ A[i]$;

  prod $:=$ prod $* A[i]$;

  $i := i + 1$ **od**;

PRINT(“`sum = `”, sum);

PRINT(“`prod = `”, prod)

Slice with respect to the variable prod on the last line

Slicing Example

$\boxed{\textsf{sum} := 0;}$

prod $:= 1;$

$i := 1;$

while $i \leqslant n$ **do**

  $\boxed{\textsf{sum} := \textsf{sum} + A[i];}$

  prod $:=$ prod $* A[i];$

  $i := i + 1$ **od**;

$\boxed{\textsf{PRINT}(\text{“}\texttt{sum = }\text{”}, \textsf{sum});}$

PRINT(“`prod = `”, prod)

Slice with respect to the variable prod on the last line

These statements can be deleted

Slicing Example

prod $:= 1;$

$i := 1;$

while $i \leqslant n$ **do**

prod $:=$ prod $* A[i];$

$i := i + 1$ **od**;

PRINT("`prod = `", prod)

Slice with respect to the variable prod on the last line

The resultant slice

Slicing in the Middle

Suppose we want to slice on the value of i in 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 allow $i := 0$ as a valid slice: which is not what we wanted!

Slicing in the Middle

Add a new variable, slice, which records this sequence of values of the variable of interest at the point of interest:

$i := 0;\ s := 0;$

while $i < n$ **do**

$\quad s := s + i;$

$\quad i := i + 1$ **od**;

$i := 0;$

Slicing in the Middle

Add a new variable, slice, which records this sequence of values of the variable of interest at the point of interest:

$i := 0;\ s := 0;$

while $i < n$ **do**

  slice $:=$ slice $+\!\!+ \langle i \rangle;$

  $s := s + i;$

  $i := i + 1$ **od**;

$i := 0;$

Slicing on slice at the end of the program is equivalent to slicing on i at the top of the loop.

Slicing in the Middle

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:

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

So the sliced program is:

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

Slicing as a Program Transformation

A key insight of this formulation is that it defines the concept of slicing as a combination of two relations:

1. A syntactic relation (statement deletion) and
2. A semantic relation (which shows what subset of the semantics has been preserved).

Slicing Definition

A *slicing criterion* is a set of points in a program (the points of interest) with a set of variables associated with each point (the variables of interest).

A *syntactic slice* of a program **S** on a given slicing criterion is any program $\mathbf{S}'$ formed by deleting statements from **S** such that $\mathbf{S}'$ preserves the values of the variables of interest at each of the points of interest. The slice may terminate on initial states for which the original program does not terminate.

A *semantic slice* of a program **S** on a given slicing criterion is any program $\mathbf{S}'$ such that $\mathbf{S}'$ preserves the values of the variables of interest at each of the points of interest. The slice may terminate on initial states for which the original program does not terminate.

"Syntax" is everything pertaining to form rather than meaning.

"Semantic" refers to meaning, independent of form.

Slicing Definition

A syntactic slice must preserve two relations:

Slicing Definition

A syntactic slice must preserve two relations:

1. A syntactic relation. The slice is formed from the original program by deleting statements. This relation is called *reduction*

Slicing Definition

A syntactic slice must preserve two relations:

1. A syntactic relation. The slice is formed from the original program by deleting statements. This relation is called *reduction*
2. A semantic relation. The slice must preserve the variables of interest at the points of interest. The slice may terminate when the original program did not. This relation is called *semi-refinement*

Slicing Definition

A syntactic slice must preserve two relations:

1. A syntactic relation. The slice is formed from the original program by deleting statements. This relation is called *reduction*

2. A semantic relation. The slice must preserve the variables of interest at the points of interest. The slice may terminate when the original program did not. This relation is called *semi-refinement*

A semantic slice only has to preserve the semantic relation. So any syntactic slice is also a semantic slice.

Slicing Definition

Any syntactic slice is also a semantic slice, but the reverse is not necessarily the case.

Since a semantic slice is formed from the original program by deleting statements, there can be only a finite number of possible syntactic slices for a given program.

There can be infinitely many different semantic slices for a program. For example, adding a **skip** statement to a semantic slice gives a different semantic slice.

Syntactic vs Semantic Slicing

Example:

Original Program	Syntactic slice on y

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

Syntactic vs Semantic Slicing

Example:

Original Program

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

Syntactic slice on y

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

Semantic slice on y

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

Termination

Suppose we are slicing on the final value of x:

$$\begin{array}{l} \textbf{while } n > 1 \textbf{ do} \\ \quad \textbf{if } \text{odd?}(n) \textbf{ then } n := 3 * n + 1 \\ \qquad\qquad\qquad\quad \textbf{else } n = n/2 \textbf{ fi od}; \\ x := 3 \end{array}$$

The loop clearly cannot affect the value assigned to x so it should be deleted: even though it is not known whether the loop will always terminate.

Termination

Another example:

Original Program	Syntactic slice on x
while $n > 1$ **do** $y := f(y)$; $n := n - 1$ **od**; **if** $y > 0$ **then** $x := 3;\ z := 2$ **else** $z := 4;\ x := 3$ **fi**	**if** $y > 0$ **then** $x := 3$ **else** $x := 3$ **fi**

Termination

Another example:

Original Program

while $n > 1$ **do**
$y := f(y);$
$n := n - 1$ **od**;
if $y > 0$ **then** $x := 3;\ z := 2$
else $z := 4;\ x := 3$ **fi**

Syntactic slice on x

if $y > 0$ **then** $x := 3$
else $x := 3$ **fi**

Semantic slice on x

$x := 3$

Termination

Another example:

Original Program	Semantic slice on x
while $n > 1$ **do** $y := f(y);$ $n := n - 1$ **od**; **if** $y > 0$ **then** $x := 3$ **else abort fi**	$x := 3$

Termination

Another example:

Original Program	Semantic slice on x
while $n > 1$ **do** $\quad y := f(y);$ $\quad n := n - 1$ **od**; **if** $y > 0$ **then** $x := 3$ $\qquad$ **else abort fi**	$x := 3$

In this case, the smallest syntactic slice is the original program.

Reduction

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 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:

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

Reduction

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

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

$$\textbf{do S}' \textbf{ od} \;\sqsubseteq\; \textbf{do S od}$$

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

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

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

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

Terminal Values TVs(S)

A proper sequence is any program such that every **exit**(n) is contained within at least n enclosing loops.

If **S** contains an **exit**(n) surrounded by loops nested fewer than n deep, then this **exit** will cause termination of one or more loops enclosing **S**.

The set of terminal values of **S**, denoted TVs(**S**) is the set of integers $n - d > 0$ such that there is an **exit**(n) within d nested loops in **S** which could cause termination of **S**. TVs(**S**) also contains 0 if **S** could terminate normally (i.e. without terminating an enclosing loop).

Terminal Values $\mathrm{TVs}(\mathbf{S})$

For example, if **S** is the program:

do if $x = 0$ **then exit**(3)

elsif $x = 1$ **then exit**(2) **fi**;

$x := x - 2$ **od**

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

Reduction

The reduction relation does not allow actual deletion of statements: only replacing a statement by a **skip**.

This makes it easy to determine the relationship between components of the original and the reduced program.

Three important properties of the reduction relation are:

Lemma 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 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 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

Semi-Refinement

A slice does not have to be exactly equivalent to the original program. Consider the program:

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

where we are slicing on x and **S** has no assignments to x. Clearly we want to slice away **S**.

But **S**; $x := 0$ is only *equivalent* to $x := 0$ on x provided **S** terminates.

We want to be able to "slice away" potentially non-terminating code. The semantic relation we need is **semi-refinement**:

$$\Delta \vdash \mathbf{S} \ \preccurlyeq\ \mathbf{S}'$$

If **S** terminates, then **S** $\approx$ **S**$'$, if **S** does not terminate then **S**$'$ can be anything at all.

Weakest Preconditions

Dijkstra introduced the concept of weakest preconditions as a tool for reasoning about programs.

For a given program **P** and condition **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 **P** is started in a state satisfying $\mathrm{WP}(\mathbf{P}, \mathbf{R})$ then it is guaranteed to terminate in a state satisfying **R**.

By using an infinitary logic, it turns out that $\mathrm{WP}(\mathbf{P}, \mathbf{R})$ has a simple definition for all kernel language programs **S** and all (infinitary logic) formulae **R**.

Weakest Preconditions

For any kernel language statement $\mathbf{S}\colon V \to W$, and formula $\mathbf{R}$ whose free variables are all in W, we define $\mathsf{WP}(\mathbf{S}, \mathbf{R})$ as follows:

1. $\mathsf{WP}(\{\mathbf{P}\}, \mathbf{R}) \;=_{\text{DF}}\; \mathbf{P} \wedge \mathbf{R}$
2. $\mathsf{WP}([\mathbf{Q}], \mathbf{R}) \;=_{\text{DF}}\; \mathbf{Q} \Rightarrow \mathbf{R}$
3. $\mathsf{WP}(\mathsf{add}(\mathbf{x}), \mathbf{R}) \;=_{\text{DF}}\; \forall \mathbf{x}.\, \mathbf{R}$
4. $\mathsf{WP}(\mathsf{remove}(\mathbf{x}), \mathbf{R}) \;=_{\text{DF}}\; \mathbf{R}$
5. $\mathsf{WP}((\mathbf{S}_1;\ \mathbf{S}_2), \mathbf{R}) \;=_{\text{DF}}\; \mathsf{WP}(\mathbf{S}_1, \mathsf{WP}(\mathbf{S}_2, \mathbf{R}))$
6. $\mathsf{WP}((\mathbf{S}_1 \ \sqcap\ \mathbf{S}_2), \mathbf{R}) \;=_{\text{DF}}\; \mathsf{WP}(\mathbf{S}_1, \mathbf{R}) \wedge \mathsf{WP}(\mathbf{S}_2, \mathbf{R})$
7. $\mathsf{WP}((\mu X.\mathbf{S}), \mathbf{R}) \;=_{\text{DF}}\; \bigvee_{n<\omega} \mathsf{WP}((\mu X.\mathbf{S})^n, \mathbf{R})$

where $(\mu X.\mathbf{S})^0 = \mathbf{abort}$ 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 $(\mu X.\mathbf{S})^n$.

Weakest Preconditions

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.

For example, for any fomula **R**:

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

which proves that:

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

Proof Theoretic Refinement

Proof theoretic refinement is defined from the weakest precondition formula WP, applied to the special postcondition $\mathbf{x} \neq \mathbf{x}'$ where $\mathbf{x}$ is a list of all the variables assigned in either statement, and $\mathbf{x}'$ is a list of new variables.

If $\mathbf{S}, \mathbf{S}' : V \to W$ have no free statement variables and $\mathbf{x}$ is a sequence of all variables assigned to in either $\mathbf{S}$ or $\mathbf{S}'$, and the formulae

$$\mathsf{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathsf{WP}(\mathbf{S}', \mathbf{x} \neq \mathbf{x}')$$

and

$$\mathsf{WP}(\mathbf{S}, \mathbf{true}) \Rightarrow \mathsf{WP}(\mathbf{S}', \mathbf{true})$$

are provable from the set Δ of sentences, then we say that $\mathbf{S}$ is refined by $\mathbf{S}'$ and write:

$$\Delta \vdash \ \mathbf{S} \ \leq \ \mathbf{S}'$$

Semi-Refinement

Semi-refinement:

$$\Delta \vdash \mathbf{S} \ \preccurlyeq\ \mathbf{S}'$$

is defined as:

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

If $\mathbf{S}$ terminates, then $\mathrm{WP}(\mathbf{true}, \mathbf{S})$ is **true** and the assertion is a **skip**. In this case, we must have $\mathbf{S}' \ \approx\ \mathbf{S}$

If $\mathbf{S}$ may not terminate, then $\mathrm{WP}(\mathbf{true}, \mathbf{S})$ is **false** and the assertion is **abort**. In this case, $\mathbf{S}'$ can be anything at all.

Semi-refinement lies between semantic equivalence and semantic refinement.

Semi-refinement captures precisely what we need for the formal mathematical definition of slicing.

Equivalence or Semi-Refinement?

The following three programs show that there is *no* semantic equivalence relation which can be used to define program slicing, and which allows deletion of irrelevant code:

P_1	P_2	P_3
$x := 3;$ **while** $y \neq 0$ **do** $\quad z := z + y;$ $\quad y := y - 1$ **od**	$x := 3$	$x := 3;$ **while** $y \neq 0$ **do** $\quad z := z + y$ **od**

Equivalence or Semi-Refinement?

P_1	P_2	P_3

$x := 3;$
while $y \neq 0$ **do**
    $z := z + y;$
    $y := y - 1$ **od**
	$x := 3$	$x := 3;$
		while $y \neq 0$ **do**
		    $z := z + y$ **od**

- The **while** loop in P_1 does not affect x, so we should be able to delete it
- But the loop does not terminate when $y < 0$ initially
- So the equivalence relation must have **abort** equivalent to **skip**
- But this would allow P_3 as a valid slice of P_1
- But P_3 does not terminate in cases when P_1 does!

Syntactic Slice

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

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

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

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.

A more complex slicing criterion might consist of a set of program points, with a different set of variables of interest at each point.

Simple Slicing

One way to compute slices using program transformations:

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.

The Slicing Relation

If $\mathbf{S}, \mathbf{S}' : V \to W$ and $Y \subseteq V$ and $X \subseteq W$, then: $\Delta \vdash \mathbf{S}' \;{}_Y\!\!\lessdot_X \mathbf{S}$ iff:

$\mathbf{S}' \sqsubseteq \mathbf{S}$ and

$$\begin{aligned} \Delta \vdash \mathbf{S};\ \mathbf{remove}(W \setminus X) &\preccurlyeq \mathbf{add}(V \setminus Y);\ \mathbf{S}';\ \mathbf{remove}(W \setminus X) \\ &\approx \mathbf{S}';\ \mathbf{remove}(W \setminus X) \end{aligned}$$

X is the set of variables of interest in the final state space. Y includes all the variables whose initial values are needed to compute the final values of X.

Note that Y may be larger than is strictly necessary.

An example:

$$\mathbf{skip};\ x := y + 1 \;\;{}_{\{y\}}\!\!\lessdot_{\{x\}}\;\; z := 4;\ x := y + 1$$

Properties of the Slicing Relation

- **Weaken Requirements:**
  If $X_1 \subseteq X$ and $Y \subseteq Y_1$ and $\mathbf{S}' \ {}_{Y}\!\lessdot_{X} \ \mathbf{S}$ then $\mathbf{S}' \ {}_{Y_1}\!\lessdot_{X_1} \ \mathbf{S}$
  Example: **skip**; $x := y + 1 \ \ {}_{\{y\}}\!\lessdot_{\{x\}} \ \ z := 4;\ x := y + 1$, and $\{\} \subseteq \{x\}$ and $\{y\} \subseteq \{y, z\}$, so:

$$\textbf{skip};\ x := y + 1 \ \ {}_{\{y,z\}}\!\lessdot_{\{\}} \ \ z := 4;\ x := y + 1$$

Properties of the Slicing Relation

- **Weaken Requirements:**
  If $X_1 \subseteq X$ and $Y \subseteq Y_1$ and $\mathbf{S}'\ {}_{Y}\!\lessdot_{X}\ \mathbf{S}$ then $\mathbf{S}'\ {}_{Y_1}\!\lessdot_{X_1}\ \mathbf{S}$
  Example: **skip**; $x := y + 1\ \ {}_{\{y\}}\!\lessdot_{\{x\}}\ \ z := 4;\ x := y + 1$, and $\{\} \subseteq \{x\}$ and $\{y\} \subseteq \{y, z\}$, so:

$$\textbf{skip};\ x := y + 1\ \ {}_{\{y,z\}}\!\lessdot_{\{\}}\ \ z := 4;\ x := y + 1$$

- **Strengthen Requirements:**
  If $\mathbf{S}'\ {}_{Y}\!\lessdot_{X}\ \mathbf{S}$ and variable y does not appear in $\mathbf{S}$ or $\mathbf{S}'$, then $\mathbf{S}'\ {}_{Y\setminus\{y\}}\!\lessdot_{X\cup\{y\}}\ \mathbf{S}$
  Example: variable p does not appear in our example, so:

$$\textbf{skip};\ x := y + 1\ \ {}_{\{z\}}\!\lessdot_{\{x,p\}}\ \ z := 4;\ x := y + 1$$

Properties of the Slicing Relation

- **Identity Slice:**
  If **S** $: V \rightarrow W$ and $X \subseteq W$ then **S** ${}_V\!\leqslant_X$ **S**.
  Any slicing relation ought to allow any statement as a valid slice of itself.

Properties of the Slicing Relation

- **Identity Slice:**
  If **S** $: V \to W$ and $X \subseteq W$ then **S** ${}_V\!\lessdot_X$ **S**.
  Any slicing relation ought to allow any statement as a valid slice of itself.

- **Abort:**
  abort ${}_\varnothing\!\lessdot_X$ **abort** and **skip** ${}_\varnothing\!\lessdot_X$ **abort** for any X.
  Since the **abort** is guaranteed not to terminate, code before the **abort** has no effect, and therefore we don't need to preserve the values of any variables before the **abort**.

Properties of the Slicing Relation

- **Identity Slice:**
  If $\mathbf{S} : V \to W$ and $X \subseteq W$ then $\mathbf{S} \; {}_V\!\!\lessdot_X \; \mathbf{S}$.
  Any slicing relation ought to allow any statement as a valid slice of itself.

- **Abort:**
  abort ${}_\varnothing\!\!\lessdot_X$ **abort** and **skip** ${}_\varnothing\!\!\lessdot_X$ **abort** for any X.
  Since the **abort** is guaranteed not to terminate, code before the **abort** has no effect, and therefore we don't need to preserve the values of any variables before the **abort**.

- **Assertion:**
  For any formula **Q** and any set X: **skip** ${}_X\!\!\lessdot_X$ $\{\mathbf{Q}\}$ and $\{\mathbf{Q}\}$ ${}_Y\!\!\lessdot_X$ $\{\mathbf{Q}\}$ where $Y = X \cup \mathsf{vars}(\mathbf{Q})$.
  Any assertion can be deleted, since it is OK for the slice to terminate when the original program does not.

Properties of the Slicing Relation

- **Add Variables:**
  For any set X and list of variables $\mathbf{x}$: $\mathbf{add}(\mathbf{x}) \ {}_{Y}\!\leqslant_{X} \mathbf{add}(\mathbf{x})$ where $Y = X \setminus \mathsf{vars}(\mathbf{x})$

$$\mathbf{add}(\langle x, y\rangle) \ {}_{\{z\}}\!\leqslant_{\{x,y,z\}} \ \mathbf{add}(\langle x, y\rangle)$$

Properties of the Slicing Relation

- **Add Variables:**
  For any set X and list of variables $\mathbf{x}$: $\mathbf{add}(\mathbf{x})\ {}_{Y}\!\lessdot_{X}\ \mathbf{add}(\mathbf{x})$ where $Y = X \setminus \mathsf{vars}(\mathbf{x})$

$$\mathbf{add}(\langle x, y\rangle)\ \ {}_{\{z\}}\!\lessdot_{\{x,y,z\}}\ \ \mathbf{add}(\langle x, y\rangle)$$

- **Remove Variables:**
  For any set X and list of variables $\mathbf{x}$:
  $\mathbf{remove}(\mathbf{x})\ {}_{X}\!\lessdot_{X}\ \mathbf{remove}(\mathbf{x})$. Note that $\mathsf{vars}(\mathbf{x})$ and X are disjoint since X must be a subset of the final state space, and no variable in $\mathbf{x}$ is in the final state space.

Properties of the Slicing Relation

- **Specification Statement:**
  If $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$ is any specification statement then
  $\mathbf{x} := \mathbf{x}'.\mathbf{Q} \;\; {}_{Y}\!\preccurlyeq_{X} \;\; \mathbf{x} := \mathbf{x}'.\mathbf{Q}$ where
  $Y = (X \setminus \mathsf{vars}(\mathbf{x})) \cup (\mathsf{vars}(\mathbf{Q}) \setminus \mathsf{vars}(\mathbf{x}'))$

$$\langle x \rangle := \langle x' \rangle.(x' = y + 3) \;\; {}_{\{y,z\}}\!\preccurlyeq_{\{x,z\}} \;\; \langle x \rangle := \langle x' \rangle.(x' = y + 3)$$

$$\langle x \rangle := \langle x' \rangle.(x' = x + y) \;\; {}_{\{x,y,z\}}\!\preccurlyeq_{\{x,z\}} \;\; \langle x \rangle := \langle x' \rangle.(x' = x + y)$$

Properties of the Slicing Relation

- **Specification Statement:**
  If $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$ is any specification statement then
  $\mathbf{x} := \mathbf{x}'.\mathbf{Q} \;\; {}_Y\!\preccurlyeq_X \;\; \mathbf{x} := \mathbf{x}'.\mathbf{Q}$ where
  $Y = (X \setminus \mathsf{vars}(\mathbf{x})) \cup (\mathsf{vars}(\mathbf{Q}) \setminus \mathsf{vars}(\mathbf{x}'))$

$$\langle x \rangle := \langle x' \rangle.(x' = y + 3) \;\; {}_{\{y,z\}}\!\preccurlyeq_{\{x,z\}} \;\; \langle x \rangle := \langle x' \rangle.(x' = y + 3)$$

$$\langle x \rangle := \langle x' \rangle.(x' = x + y) \;\; {}_{\{x,y,z\}}\!\preccurlyeq_{\{x,z\}} \;\; \langle x \rangle := \langle x' \rangle.(x' = x + y)$$

- **Assignment:**
  If $x := e$ is any assignment, then: $x := e \;\; {}_Y\!\preccurlyeq_X \;\; x := e$ where
  $Y = (X \setminus \{x\}) \cup \mathsf{vars}(e)$

  This is a special case of the specification statement.

Properties of the Slicing Relation

- **Total Slice:**
  If **S** $: V \to V$ and $X \subseteq V$ and no variable in X is assigned in **S**, then: **skip** ${}_X\!\preccurlyeq_X$ **S**. In particular, **skip** ${}_X\!\preccurlyeq_X$ **skip** for any X.

  For example:

  $$\textbf{skip} \;\; {}_{\{z\}}\!\preccurlyeq_{\{z\}} \;\; \langle x \rangle := \langle x' \rangle.(x' = y + 3)$$

  $$\textbf{skip} \;\; {}_{\{q,r\}}\!\preccurlyeq_{\{q,r\}} \;\; \textbf{while } y \neq 0 \textbf{ do } x := x + y;\; y := y - 1 \textbf{ od}$$

  (Note: the Assertion property is actually a special case).

Properties of the Slicing Relation

- **Total Slice:**
  If **S** $: V \to V$ and $X \subseteq V$ and no variable in X is assigned in **S**, then: **skip** ${}_X\!\preccurlyeq\!\!\!\!/_X$ **S**. In particular, **skip** ${}_X\!\preccurlyeq\!\!\!\!/_X$ **skip** for any X.

  For example:

  $$\textbf{skip} \ {}_{\{z\}}\!\preccurlyeq\!\!\!\!/_{\{z\}} \ \langle x\rangle := \langle x'\rangle.(x' = y + 3)$$

  $$\textbf{skip} \ {}_{\{q,r\}}\!\preccurlyeq\!\!\!\!/_{\{q,r\}} \ \textbf{while } y \neq 0 \textbf{ do } x := x + y;\ y := y - 1 \textbf{ od}$$

  (Note: the Assertion property is actually a special case).

- **Sequence:**
  If $\mathbf{S}_1, \mathbf{S}'_1 : V \to V_1$, $\mathbf{S}_2, \mathbf{S}'_2 : V_1 \to W$, $Y \subseteq W$, $X_1 \subseteq V_1$ and $X \subseteq V$ are such that $\mathbf{S}'_1 \ {}_Y\!\preccurlyeq\!\!\!\!/_{X_1} \ \mathbf{S}_1$ and $\mathbf{S}'_2 \ {}_{X_1}\!\preccurlyeq\!\!\!\!/_X \ \mathbf{S}_2$ then:

  $$(\mathbf{S}'_1;\ \mathbf{S}'_2) \ {}_Y\!\preccurlyeq\!\!\!\!/_X \ (\mathbf{S}_1;\ \mathbf{S}_2)$$

Sequence Example

Slicing the sequence: $x := y + 3$; $z := z + x$; $x := x + y$ on $\{x\}$:

Sequence Example

Slicing the sequence: $x := y + 3;\ z := z + x;\ x := x + y$ on $\{x\}$:

By the Assignment property on $x := x + y$, we have:

$$x := x + y \ \ {}_{\{x,y\}}\!\not\leqslant_{\{x\}} \ \ x := x + y$$

Sequence Example

Slicing the sequence: $x := y + 3;\ z := z + x;\ x := x + y$ on $\{x\}$:

By the Assignment property on $x := x + y$, we have:

$$x := x + y \ \ {}_{\{x,y\}}\!\lessdot_{\{x\}} \ \ x := x + y$$

Now slice $z := z + x$ on $\{x, y\}$. By the Total Slice property:

$$\textbf{skip} \ \ {}_{\{x,y\}}\!\lessdot_{\{x,y\}} \ \ z := z + x$$

Sequence Example

Slicing the sequence: $x := y + 3$; $z := z + x$; $x := x + y$ on $\{x\}$:

By the Assignment property on $x := x + y$, we have:

$$x := x + y \ \ {}_{\{x,y\}}\!\!\not\leqslant_{\{x\}} \ \ x := x + y$$

Now slice $z := z + x$ on $\{x, y\}$. By the Total Slice property:

$$\textbf{skip} \ \ {}_{\{x,y\}}\!\!\not\leqslant_{\{x,y\}} \ \ z := z + x$$

Now slice $x := y + 3$ on $\{x, y\}$. By the Assignment property:

$$x := y + 3 \ \ {}_{\{y\}}\!\!\not\leqslant_{\{x,y\}} \ \ x := y + 3$$

Sequence Example

Slicing the sequence: $x := y + 3;\ z := z + x;\ x := x + y$ on $\{x\}$:

By the Assignment property on $x := x + y$, we have:

$$x := x + y \quad {}_{\{x,y\}}\!\lessdot_{\{x\}} \quad x := x + y$$

Now slice $z := z + x$ on $\{x, y\}$. By the Total Slice property:

$$\textbf{skip} \quad {}_{\{x,y\}}\!\lessdot_{\{x,y\}} \quad z := z + x$$

Now slice $x := y + 3$ on $\{x, y\}$. By the Assignment property:

$$x := y + 3 \quad {}_{\{y\}}\!\lessdot_{\{x,y\}} \quad x := y + 3$$

Putting these results together, by the Sequence property:

$$x := y + 3;\ \textbf{skip};\ x := x + y \quad {}_{\{y\}}\!\lessdot_{\{x\}} \quad x := y + 3;\ z := z + x;\ x := x + y$$

Properties of the Slicing Relation

- **Deterministic Choice:**
  If $\mathsf{S}_1, \mathsf{S}_2, \mathsf{S}'_1, \mathsf{S}'_2 : V \to W$, and $X \subseteq W$, $Y_i \subseteq V$ are such that $\mathsf{S}'_i \;{}_{Y_i}\!\preccurlyeq_X \mathsf{S}_i$ and **B** is any formula, then:

  $$\textbf{if B then } \mathsf{S}'_1 \textbf{ else } \mathsf{S}'_2 \textbf{ fi } \;{}_{Y}\!\preccurlyeq_X\; \textbf{if B then } \mathsf{S}_1 \textbf{ else } \mathsf{S}_2 \textbf{ fi}$$

  where $Y = Y_1 \cup Y_2 \cup \mathsf{vars}(\mathbf{B})$. This can be extended to a multi-way **if** statement.

  $$\textbf{if } z = 0 \textbf{ then } x := x + y \textbf{ else skip fi}$$

  $$\;{}_{\{x,y,z\}}\!\preccurlyeq_{\{x\}}\; \textbf{if } z = 0 \textbf{ then } x := x + y \textbf{ else } p := q + 1 \textbf{ fi}$$

Properties of the Slicing Relation

- **Nondeterministic Choice:**
  If $\mathsf{S}_1, \mathsf{S}_2, \mathsf{S}'_1, \mathsf{S}'_2 : V \to W$, and $X \subseteq W$, $Y_i \subseteq V$ are such that $\mathsf{S}'_i \;{}_{Y_i}\!\preccurlyeq_X \mathsf{S}_i$ and $\mathbf{B}_1$ and $\mathbf{B}_2$ are any formulae, then:

$$\textbf{if } \mathbf{B}_1 \;\to\; \mathsf{S}'_1 \;\square\; \mathbf{B}_2 \;\to\; \mathsf{S}'_2 \textbf{ fi} \;\;{}_{Y}\!\preccurlyeq_X\;\; \textbf{if } \mathbf{B}_1 \;\to\; \mathsf{S}_1 \;\square\; \mathbf{B}_2 \;\to\; \mathsf{S}_2 \textbf{ fi}$$

  where $Y = Y_1 \cup Y_2 \cup \mathsf{vars}(\mathbf{B})$. Again, this can be extended to a multi-way statement.

$$\textbf{if } z = 0 \;\to\; x := x + y \;\square\; z > 0 \;\to\; \textbf{skip fi}$$

$$_{\{x,y,z\}}\!\preccurlyeq_{\{x\}} \;\; \textbf{if } z = 0 \;\to\; x := x + y \;\square\; z > 0 \;\to\; p := q + 1 \textbf{ fi}$$

Properties of the Slicing Relation

- **Local Variable:**
  If $\mathbf{S}, \mathbf{S}' : V \to W$, $X \subseteq W$ and $\mathbf{S}' \;{}_{Y}\!\leqslant_{X \setminus \{x\}} \mathbf{S}$, then let $Y_1 = (Y \setminus \{x\}) \cup (\{x\} \cap X)$ and $Y_2 = Y_1 \cup \mathsf{vars}(e)$. Then:

$$\begin{array}{rcll} \textbf{var } \langle x := \perp \rangle : \mathbf{S}' \textbf{ end} & {}_{Y_1}\!\leqslant_X & \textbf{var } \langle x := e \rangle : \mathbf{S} \textbf{ end} & \text{if } x \notin Y \\ \textbf{var } \langle x := e \rangle : \mathbf{S}' \textbf{ end} & {}_{Y_2}\!\leqslant_X & \textbf{var } \langle x := e \rangle : \mathbf{S} \textbf{ end} & \text{otherwise} \end{array}$$

  The component $(\{x\} \cap X)$ ensures that the *global* variable x is added to the required initial set if and only if it is in the required final set. Note that the second relation above is also true when $x \notin Y$, but we usually want to minimise the initial set of variables, so the first relation is preferred for computing a slice.

Local Variable Examples

In this example, the initial value of x is not needed:

$$\textbf{var}\ \langle x := \perp\rangle : x := y + 3;\ \textbf{skip};\ z := y\ \textbf{end}$$

$$_{\{y\}}\not\leqslant_{\{z\}}\ \textbf{var}\ \langle x := y\rangle : x := y + 3;\ z := z + y;\ z := y\ \textbf{end}$$

In this example, the initial value of local variable x *is* needed, and it is, in fact, the value of the *global* variable x:

$$\textbf{var}\ \langle x := x\rangle : x := x + 3;\ \textbf{skip};\ z := x\ \textbf{end}$$

$$_{\{x\}}\not\leqslant_{\{z\}}\ \textbf{var}\ \langle x := x\rangle : x := x + 3;\ z := z + y;\ z := x\ \textbf{end}$$

Properties of the Slicing Relation

- **While Loop:**
  If $\mathbf{S}, \mathbf{S}' : V \to V$ and $Y \subseteq V$ are such that $\mathbf{S}' \ {}_{Y}\!\lessdot_{Y}\ \mathbf{S}$, and $\mathsf{vars}(\mathbf{B}) \subseteq Y$, then:

  $$\textbf{while B do S}' \textbf{ od} \ \ {}_{Y}\!\lessdot_{Y}\ \ \textbf{while B do S od}$$

  Unlike all the other properties, this property gives no indication as to how to compute the set Y from a given set X of variables of interest.

  A simple method is to start with the $X \cup \mathsf{vars}(\mathbf{B})$ and repeatedly process **S**, adding variables as necessary, until the result converges.

While Loop Example

while $i \neq 0$ **do**

$y := x_1;\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1$ **od**

We want to slice this loop on $\{x_1\}$.

While Loop Example

while $i \neq 0$ **do**

$y := x_1;\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1$ **od**

We want to slice this loop on $\{x_1\}$.

Let **S** be the loop body, and slice it on $\{x_1, i\}$, using the properties:

$$\textbf{skip};\ x_1 := x_2;\ \textbf{skip};\ i := i - 1 \ \ {}_{\{x_2,i\}}\!\not\leqslant_{\{x_1,i\}} \ \ \textbf{S}$$

While Loop Example

while $i \neq 0$ **do**

$y := x_1;\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1$ **od**

We want to slice this loop on $\{x_1\}$.

Let **S** be the loop body, and slice it on $\{x_1, i\}$, using the properties:

$$\textbf{skip};\ x_1 := x_2;\ \textbf{skip};\ i := i - 1 \ \ {}_{\{x_2,i\}}\!\leqslant_{\{x_1,i\}} \ \ \textbf{S}$$

So we need to add x_2 to the set of variables of interest.

$$\textbf{skip};\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1 \ \ {}_{\{x_2,x_3,i\}}\!\leqslant_{\{x_1,x_2,i\}} \ \ \textbf{S}$$

While Loop Example

while $i \neq 0$ **do**

$y := x_1;\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1$ **od**

We want to slice this loop on $\{x_1\}$.

Let **S** be the loop body, and slice it on $\{x_1, i\}$, using the properties:

$$\textbf{skip};\ x_1 := x_2;\ \textbf{skip};\ i := i - 1 \ \ {}_{\{x_2,i\}}\!\preccurlyeq_{\{x_1,i\}} \ \mathbf{S}$$

So we need to add x_2 to the set of variables of interest.

$$\textbf{skip};\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1 \ \ {}_{\{x_2,x_3,i\}}\!\preccurlyeq_{\{x_1,x_2,i\}} \ \mathbf{S}$$

So we also need to add x_3 to the set:

$$\textbf{skip};\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1 \ \ {}_{\{x_2,x_3,i\}}\!\preccurlyeq_{\{x_1,x_2,x_3,i\}} \ \mathbf{S}$$

While Loop Example

while $i \neq 0$ **do**

$y := x_1;\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1$ **od**

We want to slice this loop on $\{x_1\}$.

Let **S** be the loop body, and slice it on $\{x_1, i\}$, using the properties:

$$\textbf{skip};\ x_1 := x_2;\ \textbf{skip};\ i := i - 1 \ \ {}_{\{x_2,i\}}\!\preccurlyeq_{\{x_1,i\}} \ \ \textbf{S}$$

So we need to add x_2 to the set of variables of interest.

$$\textbf{skip};\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1 \ \ {}_{\{x_2,x_3,i\}}\!\preccurlyeq_{\{x_1,x_2,i\}} \ \ \textbf{S}$$

So we also need to add x_3 to the set:

$$\textbf{skip};\ x_1 := x_2;\ x_2 := x_3;\ i := i - 1 \ \ {}_{\{x_2,x_3,i\}}\!\preccurlyeq_{\{x_1,x_2,x_3,i\}} \ \ \textbf{S}$$

The iteration converges, so we set $Y = \{x_1, x_2, x_3, i\}$

While Loop Example

We have proved that:

$$\begin{array}{lcl} \textbf{while } i \neq 0 \textbf{ do} & {}_{\{x_1,x_2,x_3,i\}}\not\leqslant_{\{x_1,x_2,x_3,i\}} & \textbf{while } i \neq 0 \textbf{ do} \\ \quad \textbf{skip}; & & \quad y := x_1; \\ \quad x_1 := x_2; & & \quad x_1 := x_2; \\ \quad x_2 := x_3; & & \quad x_2 := x_3; \\ \quad i := i - 1 \textbf{ od} & & \quad i := i - 1 \textbf{ od} \end{array}$$

While Loop Example

We have proved that:

$$\begin{array}{lcl}
\textbf{while } i \neq 0 \textbf{ do} & {}_{\{x_1,x_2,x_3,i\}}\not\leqslant_{\{x_1,x_2,x_3,i\}} & \textbf{while } i \neq 0 \textbf{ do} \\
\quad \textbf{skip}; & & \quad y := x_1; \\
\quad x_1 := x_2; & & \quad x_1 := x_2; \\
\quad x_2 := x_3; & & \quad x_2 := x_3; \\
\quad i := i - 1 \textbf{ od} & & \quad i := i - 1 \textbf{ od}
\end{array}$$

So, by the Weakening Requirements property:

$$\begin{array}{lcl}
\textbf{while } i \neq 0 \textbf{ do} & {}_{\{x_1,x_2,x_3,i\}}\not\leqslant_{\{x_1\}} & \textbf{while } i \neq 0 \textbf{ do} \\
\quad \textbf{skip}; & & \quad y := x_1; \\
\quad x_1 := x_2; & & \quad x_1 := x_2; \\
\quad x_2 := x_3; & & \quad x_2 := x_3; \\
\quad i := i - 1 \textbf{ od} & & \quad i := i - 1 \textbf{ od}
\end{array}$$

Simple Slicing

This collection of properties gives enough information to compute a slice for any WSL program which uses these constructs.

This slicing algorithm has been implemented in FermaT as the Simple_Slice transformation.

The paper "Deriving a Slicing Algorithm via FermaT Transformations" Martin Ward and Hussein Zedan, (to appear in IEEE Transactions on Software Engineering), develops a formal specification for slicing, proves the various properties of the slicing relation and uses these properties to derive the simple slicing algorithm via transformational programming.

Simple Slicing Algorithm

```
proc slice() ≡
  if @ST(I) = Statements
    then var ⟨L := ⟨⟩⟩ :
           for I ∈ REVERSE(@Cs(I)) do
              slice; L := ⟨I⟩ ++ L od;
           I := @Make(Statements, ⟨⟩, L) end
  elsif @ST(I) = Abort
      then x := ⟨⟩
  elsif @Assigned(I) ∩ x = ∅
      then I := @Make(Skip, ⟨⟩, ⟨⟩)
  elsif @ST(I) = Assignment
      then x := (x \ @Assigned(I)) ∪ @Used(I)
  elsif @ST(I) = Var
      then var ⟨assign := I^1⟩ :
             var ⟨v := @V(assign^1),
             e := @Used(assign^2, x0 := x⟩ :
             I := I^2;
             slice;
             if v ∉ x
               then assign := @Make(Assign, ⟨⟩,
                          ⟨assign^1, BOTTOM⟩) fi;
             x := (x \ {v}) ∪ ({v} ∩ x0) ∪ e)
             I := @Make(Var, ⟨⟩,
               ⟨assign, I⟩) end end
  elsif @ST(I) = Cond
    then var ⟨x1 := ∅, x0 := x, G := ⟨⟩⟩ :
           for guard ∈ @Cs(I) do
              I := guard^2; x := x0; slice;
              G := ⟨@Make(Guarded, ⟨⟩,
                 ⟨guard^1, I⟩ + +G;
              x1 := x1 ∪ @Used(guard^1) ∪ x od;
           x := x1;
           I := @Make(Cond, ⟨⟩, REVERSE(G)) end
  elsif @ST(I) = While
    then var ⟨B := I^1, I0 := I^2,
           x1 := x ∪ @Used(I^1)⟩ :
           do I := I0;
              x := x1;
              slice;
              if x ⊆ x1 then exit fi;
              x1 := x1 ∪ x od;
           I := @Make(While, ⟨⟩, ⟨B, I⟩);
           x := x1 end
  else ERROR("Unexpected type: ",
           @Type_Name(@ST(I))) fi.
```

Minimal Syntactic Slice

For program understanding and debugging, small slices are more useful than large slices;

Definition: A *minimal slice* of **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 is not necessarily unique and is not necessarily a slice with the smallest number of statements.

Consider the program **S**: $x := 2;\ x := x + 1;\ x := 3$

A syntactic slice can be obtained from **S** by deleting the last statement to give $\mathbf{S}'$: $x := 2;\ x := x + 1$

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

Dynamic Syntactic Slice

A dynamic slice of a program **P** is a reduced executable program **S** which replicates part of the behaviour of **P** on a particular initial state. We can define this initial state by means of an assertion.

A **Dynamic Syntactic Slice** of **S** with respect to a formula **A** of the form

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

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

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

Conditioned Syntactic Slice

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

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

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

If we remove the requirement that $\mathbf{S}' \sqsubseteq \mathbf{S}$, then we have a **Conditioned Semantic Slice**

Conditioned Syntactic Slice

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).

So, the condition in a conditioned slice may be provided by inserting one or more assertions in the program.

Semantic Slicing

By dropping the syntactic requirement (that the slice is formed from the original program by deleting statements), we get a generalised slicing concept called *semantic slicing*.

Harman and Dancic coined the term “amorphous program slicing” for a combination of slicing and transformation of executable programs. Amorphous slicing is restricted to finite, executable programs. It is a combination of a syntactic relation (a partial order) and a semantic *equivalence* relation. As we saw earlier, no semantic equivalence relation is suitable for defining a useful program slice!

Semantic slicing applies to any WSL programs including non-executable specification statements, non-executable guard statements, and programs containing infinitary formulae;

Semantic Slicing

We define a "semantic slice" to be any semi-refinement in WSL, so the concepts of semantic slicing and amorphous slicing are distinct.

The relation between a WSL program and its semantic slice is a purely semantic one.

A *semantic slice* of **S** on X is any program **S**$'$ such that:

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

There are only a finite number of different syntactic slices, but there are infinitely many possible semantic slices for a program: including slices which are actually larger than the original program.

A conditioned slice is a slice of a program to which extra conditions have been added in the form of assertions. These conditions can allow further statements to be deleted.

Semantic Slicing

Example:

Original Program

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

Syntactic slice on y

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

Semantic Slicing

Example:

Original Program

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

Syntactic slice on y

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

Semantic slice on y is:

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

Semantic Slicing Implementation

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.

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.

Operational Slicing

An intermediate option between syntactic slicing and full semantic slicing is to restrict the transformations to preserve operational semantics;

Definition Program $\mathbf{S}'$ is an operational slice of $\mathbf{S}$ on X if there exists a sequence of statements $\mathbf{S}_1, \ldots, \mathbf{S}_n$ such that $\mathbf{S}_1 = \mathbf{S}$, $\mathbf{S}_n = \mathbf{S}'$ and for each $1 \leqslant i < n$, either $\mathbf{S}_{i+1}$ is a syntactic slice of $\mathbf{S}_i$ on X or $\Delta \vdash \mathsf{annotate}(\mathbf{S}_i) \approx \mathsf{annotate}(\mathbf{S}_{i+1})$.

An operational slice is therefore a combination of syntactic slicing and operational transformations. The implementation of operational slicing can iterate the slicing and transformation steps until the result converges.

Conditioned Semantic Slice

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

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

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.

FermaT

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

http://www.cse.dmu.ac.uk/∼mward/fermat.html
http://www.gkc.org.uk/fermat.html

FermaT is an industrial strength program transformation system, the result of two decades of research and development, released under the GNU GPL (General Public License).

FermaT's transformations include three slicers:

- Simple_Slice
- Syntactic_Slice and
- Semantic_Slice

FermaT's Syntactic Slicer

As well as the Simple_Slice, which is defined for a restricted subset of WSL, FermaT also has a Syntactic_Slice transformation. This is a more general syntactic slicer which works for unstructured code, as well as structured code, and is also an interprocedural slicer.

Syntactic_Slice is implemented by tracking data flows and control dependencies in the control flow graph (CFG):

1. Compute the control flow graph as a "basic blocks" file
2. Convert the CFG to Static Single Assignment form
3. Compute control dependencies in the SSA
4. Track control and data dependencies in the CFG to determine which blocks are in the slice
5. Delete any WSL statements whose blocks are not in the slice

FermaT Syntactic Slice

For syntactic slicing we allow deletion of unused parameters in procedures: again, this is to prevent the creation of extra dependencies.

$$
\begin{array}{ll}
\textbf{begin} & \textbf{proc}\ \mathsf{A}(\ \textbf{var}\ x, y)\ \equiv \\
\quad \mathsf{sum} := \mathsf{sum\_0}; & \quad \mathsf{Add}(y\ \textbf{var}\ x); \\
\quad i := 1; & \quad \mathsf{Inc}(\ \textbf{var}\ y)\ \textbf{end} \\
\quad \textbf{while}\ i \leqslant 10\ \textbf{do} & \textbf{proc}\ \mathsf{Add}(b\ \textbf{var}\ a)\ \equiv \\
\qquad \mathsf{A}(\ \textbf{var}\ \mathsf{sum}, i)\ \textbf{od}; & \quad a := a + b\ \textbf{end} \\
\quad \mathsf{PRINT}(\text{``}\texttt{sum = }\text{''}, \mathsf{sum}) & \textbf{proc}\ \mathsf{Inc}(\ \textbf{var}\ z)\ \equiv \\
\textbf{where} & \quad \mathsf{Add}(1\ \textbf{var}\boxed{z})\ \textbf{end}\ \textbf{end}
\end{array}
$$

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

FermaT Syntactic Slice

Original Program

begin
  sum := sum_0;
  $i := 1$;
  while $i \leqslant 10$ **do**
    A(**var** sum, i) **od**;
  PRINT("sum = ", sum)
where
proc A(**var** x, y) $\equiv$
  Add(y **var** x);
  Inc(**var** y) **end**
proc Add(b **var** a) $\equiv$
  $a := a + b$ **end**
proc Inc(**var** z) $\equiv$
  Add(1 **var** $\boxed{z}$) **end end**

Syntactic slice on z

begin
  $i := 1$;
  while $i \leqslant 10$ **do**
    A(**var** i) **od**
where
proc A(**var** y) $\equiv$
  Inc(**var** y) **end**
proc Add(b **var** a) $\equiv$
  $a := a + b$ **end**
proc Inc(**var** z) $\equiv$
  Add(1 **var** $\boxed{z}$) **end end**

FermaT Syntactic Slice

Slice on the final value of sum:

actions A1 :

A1 $\equiv$ sum $:= 0$; **call** A2 **end**

A2 $\equiv$ prod $:= 0$; **call** A3 **end**

A3 $\equiv$ $i := 1$; **call** A4 **end**

A4 $\equiv$ **if** $i \leqslant n$ **then** **call** A5 **else** **call** B1 **fi** **end**

A5 $\equiv$ sum $:=$ sum $+ A[i]$; **call** A6 **end**

A6 $\equiv$ prod $:=$ prod $* A[1]$; **call** A7 **end**

A7 $\equiv$ $i := i + 1$; **call** A4 **end**

B1 $\equiv$ PRINT(“`sum = `”, sum); **call** B2 **end**

B2 $\equiv$ PRINT(“`prod = `”, prod); **call** Z **end** **endactions**

FermaT Syntactic Slice

Slice on the final value of sum:

actions A1 :

A1 $\equiv$ sum $:=0$; $\boxed{\textbf{call } \text{A2}}$ **end**

$\boxed{\text{A2} \equiv \text{prod} := 0;\ \textbf{call}\ \text{A3}\ \textbf{end}}$

A3 $\equiv i := 1$; **call** A4 **end**

A4 $\equiv$ **if** $i \leqslant n$ **then** **call** A5 **else** $\boxed{\textbf{call}\ \text{B1}}$ **fi** **end**

A5 $\equiv$ sum $:=$ sum $+ A[i]$; $\boxed{\textbf{call}\ \text{A6}}$ **end**

$\boxed{\text{A6} \equiv \text{prod} := \text{prod} * A[1];\ \textbf{call}\ \text{A7}\ \textbf{end}}$

A7 $\equiv i := i + 1$; **call** A4 **end**

$\boxed{\text{B1} \equiv \text{PRINT}(\text{``}\texttt{sum = }\text{''}, \text{sum});\ \textbf{call}\ \text{B2}\ \textbf{end}}$

$\boxed{\text{B2} \equiv \text{PRINT}(\text{``}\texttt{prod = }\text{''}, \text{prod});\ \textbf{call}\ Z\ \textbf{end}}$ **endactions**

FermaT Syntactic Slice

Slice on the final value of sum:

actions A1 :

A1 $\equiv$ sum $:=0;$ $\boxed{\textbf{call } \text{A3}}$ **end**

A3 $\equiv$ $i := 1;$ **call** A4 **end**

A4 $\equiv$ **if** $i \leqslant n$ **then** **call** A5 **else** $\boxed{\textbf{call } Z}$ **fi** **end**

A5 $\equiv$ sum $:=$ sum $+ A[i];$ $\boxed{\textbf{call } \text{A7}}$ **end**

A7 $\equiv$ $i := i + 1;$ **call** A4 **end**

endactions

The SCAM Mug

A ceramic mug given to attendees of the First Source Code Analysis and Manipulation Workshop (SCAM) contained the program:

```
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 SCAM Mug

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

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

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

The SCAM Mug

Some of the control and data dependencies in MUG_0

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

Any algorithm which computes slices by tracking control and data dependencies will assume that every statement in the program contributes to the final value of x.

The SCAM Mug: Semantic Slice

Our aim 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.

First, 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**

The SCAM Mug: Semantic Slice

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**

The SCAM Mug: Semantic Slice

In the second **while** loop, $\neg q?(c)$ is invariant over the loop. 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**

The SCAM Mug: Semantic Slice

Now 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 SCAM Mug: Semantic Slice

Constant_Propagation shows 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**

The SCAM Mug: Semantic Slice

Constant_Propagation shows 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**

The SCAM Mug: Semantic Slice

Constant_Propagation shows 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 SCAM Mug: Syntactic Slice

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**

The SCAM Mug: Syntactic Slice

Within the second **while** loop, $\neg q?(c)$ is invariant as before, so we can make any changes we like to the body of **if** $q?(c)$ **then** $\ldots$ **fi**:

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

The SCAM Mug: Syntactic Slice

Within the second **while** loop, $\neg q?(c)$ is invariant as before, so we can make any changes we like to the body of **if** $q?(c)$ **then** ... **fi**:

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

Lets delete the assignment $c := g$.

The SCAM Mug: Syntactic Slice

Constant_Propagation then removes all references to c, so the first assignment can also be deleted:

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 SCAM Mug: Syntactic Slice

The first marked **if** statement is redundant, so we can change it to match the second one:

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

The SCAM Mug: Syntactic Slice

The first marked **if** statement is redundant, so we can change it to match the second one:

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

The loops, and $i := h(i)$ can be taken out of the enclosing **if**

The SCAM Mug: Syntactic Slice

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

The SCAM Mug: Syntactic Slice

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

Roll up the loop:

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

This is a valid syntactic slice of MUG_0 on x.

The SCAM Mug: Syntactic Slice

The slice:

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

The SCAM Mug: Syntactic Slice

The slice:

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

The SCAM Mug: Syntactic Slice

The slice:

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

The SCAM Mug: Syntactic Slice

The slice:

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

Deleting any more statements from this program will produce an incorrect result.

So this is a *minimal* slice.

It is easy to prove that it is the *only* minimal slice, in this case.

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 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 **B**$'$:

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

The Generalised Mug Problem

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 unroll this iteration:

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

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 unroll this iteration:

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

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 unroll this iteration:

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

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 then Syntactic_Slice on x:

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