Deriving a Slicing Algorithm via FermaT Transformations

M. P. Ward and H. Zedan Software Technology Research Lab
De Montfort University
The Gateway,
Leicester LE1 9BH, UK
martin@gkc.org.uk and zedan@dmu.ac.uk

November 27, 2009

Abstract

In this paper we present a case study in deriving an algorithm from a formal specification via FermaT transformations. The general method (which is presented in a separate paper) is extended to a method for deriving an implementation of a program transformation from a specification of the program transformation. We use program slicing as an example transformation, since this is of interest outside the program transformation community. We develop a formal specification for program slicing, in the form of a WSL specification statement, which is refined into a simple slicing algorithm by applying a sequence of general purpose program transformations and refinements. Finally, we show how the same methods can be used to derive an algorithm for semantic slicing. The main novel contributions of this paper are: (1) Developing a formal specification for slicing. (2) Expressing the definition of slicing in terms of a WSL specification statement. (3) By applying correctness preserving transformations to the specification we can derive a simple slicing algorithm.

1 Introduction

The wide spectrum language WSL includes high-level, abstract specifications and low-level programming constructs within a single language. The FermaT transformation theory, [49,63] which is based on WSL, is therefore capable of deriving complex algorithms from specifications [38,59], reverse engineering from programs to specifications [58,63,74], and migrating from assembler to high-level languages [53,60].

In this paper we present a case study in the derivation of an algorithm from a formal specification. The general derivation method is presented in a separate paper: in this paper we show how the method can be applied to deriving an implementation of a program transformation from a specification of the program transformation. This process is called *metatransformation* [65] since the program which is being transformed is the source code for another transformation. Previous work by the authors [66] has shown that *program slicing*, can be formalised as a program transformation. Slicing is of interest outside the field of program transformations so in this paper we will use slicing as a case study in the derivation of a transformation from a specification. We will develop a formal specification for program slicing and use transformations to refine this specification into an implementation. The WSL language and transformation theory is based on *infinitary logic*: an extension of normal first order logic which allows infinitely long formulae. The application of infinitary logic to specifications and weakest preconditions is central to the method.

The main novel contribution of the paper is the derivation of a slicing algorithm from a formal specification. This requires:

1. Developing a formal specification for slicing. The mathematical framework on which the definition of slicing is based differs from the "Amorphous slicing" framework of Binkley, Harman, Danicic et al. The two frameworks are compared and contrasted, particularly with

the aid of two novel theorems: one proves that any equivalence relation which partially defines a truncating slicing relation on a potentially divergent language and semantics, is the universal equivalence relation. The other proves that any semantic relation which partially defines a truncating slicing relation on a potentially divergent language and semantics must allow any statement as a valid slice of an **abort**. These results justify the use of semi-refinement in the formal definition of slicing.

2. Expressing the definition of slicing in terms of a WSL specification statement. The fact that WSL is based on *infinitary* logic makes this task much easier than it would be otherwise.
3. Applying various correctness-preserving transformation rules to turn the formal specification for slicing into an implementation of a slicing algorithm. The implementation is therefore guaranteed to be correct by construction. In previous work, transformations have been used to derive programs: in this paper we show that the same methods can be extended to derive metaprograms (programs which take programs as data and produce programs as output). Again, the foundation in infinitary logic is key to this extension.

2 Formal Program Development Methods

2.1 Program Verification

The *program verification* approach to ensuring the correctness of a program starts with a formal specification and a proposed implementation and uses formal methods to attempt to prove that the program is a correct implementation of the specification [16]. One obvious difficulty with this approach is that if the implementation happens to be incorrect (i.e. the program has a bug), then the attempt at a proof is doomed to failure. One way to avoid this problem is to attempt to develop the program and the correctness proof in parallel, in such a way that, provided the development succeeds, the correctness proof will also succeed. A more serious problem with verification is that the correctness proof has to consider the total behaviour of the program as a whole: for a large and complex program, this can involve a great deal of tedious work in developing and proving verification conditions. For example, the introduction of a loop requires the following steps (not necessarily in the given order):

1. Determine the loop termination condition;
2. Determine the loop body;
3. Determine a suitable loop invariant;
4. Prove that the loop invariant is preserved by the loop body;
5. Determine a variant function for the loop;
6. Prove that the variant function is reduced by the loop body (thereby proving termination of the loop);
7. Prove that the combination of the invariant plus the termination condition satisfies the specification for the loop.

Even with the aid of an automated proof assistant, there may still be several hundred remaining "proof obligations" to discharge (these are theorems which need to be proved in order to verify the correctness of the development) [7,24,34]

2.2 Algorithm Derivation

Algorithm derivation is the process of deriving an efficient executable program from a high-level abstract specification by means of a series of program refinement and transformation steps, each of which has been proved to refine or preserve the semantics of the program [15,22,31,33,49]. The resulting program is therefore guaranteed to be a correct implementation of the specification.

The *Refinement Calculus* approach [22,31,33] which appears at first sight to be a program derivation, rather than verification method, is based on Morgan's specification statement [30] and Dijkstra's guarded commands [17]. This language has very limited programming constructs: lacking loops with multiple exits, action systems with a "terminating" action, and side-effects. The most serious limitation in the method is that the transformations for introducing and manipulating loops require that any loops introduced must be accompanied by suitable invariant conditions and variant functions. Morgan says: "The refinement law for iteration *relies on* capturing the potentially unbounded repetition in a single formula, the invariant", ([31] p. 60, our emphasis). So, in order to refine a statement to a loop, the developer still has to carry out all the steps 1–7 listed above for verifying the correctness of a loop.

Calculational programming [14] is a methodology for constructing programs, based on category theory. Starting with a program which may be highly inefficient but is assumed to be correct, the developer improves its efficiency by applying calculation rules, such as fusion and tupling. Applications include optimisation problems and dynamic programming.

In contrast, our starting point is a specification statement expressed in infinitary logic. This does not have to be a deterministic specification, or even an executable specification: it is simply a precise description of the relationship between the initial state and the desired final state, or allowed final states. The basic transformation rules given in this and other papers enable the developer to *derive* a program from this formal specification by means of a sequence of general purpose correctness preserving refinements and transformations, without needing to introduce loop invariants. Variant functions (which may be based on *any* well-founded partial order relation) may be needed for the introduction of recursion, but these are much easier to discover than invariants.

3 The WSL Language and Transformation Theory

The WSL approach to program derivation and reverse engineering starts with a formally defined wide spectrum language, called WSL, which covers a wide spectrum of constructs from high-level, abstract, formal specifications to low-level programming constructs within a single language. This means that the process of deriving a program from a specification, or the reverse engineering process of deriving the specification of a given program, can be carried out as a transformation process within a single language. The FermaT transformation theory, and its associated industrial strength supporting tools, provides two different methods of proving the correctness of a refinement:

1. *Semantic Refinement* is defined in terms of the denotational semantics of WSL programs: Program $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$ if and only if the semantic function for $\mathbf{S}_2$ is *more defined* and *more deterministic* (giving set of final states for each initial state which is no larger) than the semantic function for $\mathbf{S}_1$. "*More defined*" means that the semantic function for $\mathbf{S}_2$ is defined over an initial set of states which is at least as large as that for $\mathbf{S}_1$. "*More deterministic*" means that for each initial state, the semantic function for $\mathbf{S}_2$ gives a set of possible final states which is no larger than that for $\mathbf{S}_1$.
2. *Proof Theoretic Refinement* is defined in terms of weakest preconditions [17], expressed in infinitary logic [25]. The weakest precondition for a given program and postcondition is the weakest condition on the initial state such that the program is guaranteed to terminate in a state satisfying the postcondition. In the WSL transformation theory a weakest precondition is a simple formula of infinitary logic, and it is sufficient to analyse the weakest preconditions for two special postconditions in order to prove the correctness of a refinement [49,63,73].

Using both proof methods we have developed a large catalogue of general purpose transformations and refinements, many of which are implemented in the FermaT transformation system.

WSL is constructed by starting with a small, tractable kernel language which is then built up into a powerful programming language by defining new constructs in terms of existing language constructs via *definitional transformations*. For example, the **while** loop is defined in terms of an equivalent recursive construct.

The kernel language is based on infinitary first order logic, originally developed by Carol Karp [25]. Infinitary logic is an extension of ordinary first order logic which allows conjunction and disjunction over (countably) infinite lists of formulae, but quantification over finite lists of variables, see [66] for a description of the syntax and semantics of the kernel language. The kernel language includes, among others, the following three primitive statements (where $\mathbf{P}$ is any infinitary logical formula and $\mathbf{x}$ and $\mathbf{y}$ are finite lists of variables):

1. **Assertion:** $\{\mathbf{P}\}$ is an assertion statement which acts as a partial **skip** statement. If the formula $\mathbf{P}$ is true then the statement terminates immediately without changing any variables, otherwise it aborts (we treat abnormal termination and non-termination as equivalent, so a program which aborts is equivalent to one which never terminates);
2. **Add variables: add**$(\mathbf{x})$ ensures that the variables in $\mathbf{x}$ are in the state space (by adding them if necessary) and assigns arbitrary values to the variables in $\mathbf{x}$;
3. **Remove variables: remove**$(\mathbf{y})$ ensures that the variables in $\mathbf{y}$ are *not* present in the state space (by removing them if necessary).

See [63,66] for a description of the syntax and semantics of the full kernel language.

The first level of WSL (defined directly from the kernel) contains assertions, sequencing, recursion, and nondeterministic choice, specification statements (see Section 3.1), assignments and local variables. In addition **if** statements, **while** loops, Dijkstra's guarded commands [17] and **for** loops are included.

We define a trinary relation on $\mathbf{S}$, V and W, denoted $\mathbf{S} : V \to W$, which is true if and only if $\mathbf{S}$ is a WSL statement, V and W are finite sets of variables, and W is a valid final state space for $\mathbf{S}$ with initial state space V. For two or more statements, we write $\mathbf{S}_1, \mathbf{S}_2 : V \to W$ as shorthand for: $(\mathbf{S}_1 : V \to W) \wedge (\mathbf{S}_2 : V \to W)$.

The details of WSL syntax and semantics and the transformation theory have been presented in earlier papers [51,52,55,57] and the book "Successful Evolution of Software Systems" [73] so will not be given here. Instead we will give an informal description of specification statements and the main transformations used in deriving algorithms from specifications. The main novel contributions of this paper are:

1. To demonstrate that by using specifications with infinitary logic we can give an abstract specification of a program transformation and a slicing algorithm; and
2. By applying correctness preserving transformations to the specification we can derive a simple slicing algorithm.

In the final section, we outline the derivation of a more powerful semantic slicing algorithm which is capable of computing high level abstractions of low-level programs.

3.1 The Specification Statement

For our transformation theory to be useful for both forward and reverse engineering it is important to be able to represent abstract specifications as part of the language, and this motivates the definition of the *Specification statement*. Then the refinement of a specification into an executable program, or the reverse process of abstracting a specification from executable code, can both be carried out within a single language. Specification statements are also used in semantic slicing [61, 66].

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

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

An ordinary assignment statement, such as $x := x+1$ is equivalent to the specification statement: $x := x'.(x' = x + 1)$. A specification of an "integer square root" program can be written as:

$$r := r'.(r' \in \mathbb{N} \wedge r'^2 \leqslant x < (r' + 1)^2)$$

If x is negative, then this specification will **abort**. A slight modification produces a non-deterministic specification statement which will compute either the positive or negative root:

$$r := r'.(r' \in \mathbb{Z} \wedge r'^2 \leqslant x < (|r'| + 1)^2)$$

3.2 Weakest Preconditions

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

For the specification statement $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$ we have:

$$\mathrm{WP}(\mathbf{x} := \mathbf{x}'.\mathbf{Q}, \mathbf{R}) \iff \exists \mathbf{x}'.\, \mathbf{Q} \wedge \forall \mathbf{x}'.\, (\mathbf{Q} \Rightarrow \mathbf{R}[\mathbf{x}'/\mathbf{x}])$$

The weakest precondition of an **if** statement is calculated as:

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

The weakest precondition of a **while** loop is defined a the infinite disjunction of the weakest preconditions of all the "truncations" of the loop. The truncations of a **while** loop are defined:

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

The weakest precondition for the whole loop is simply:

$$\mathrm{WP}(\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od}, \mathbf{R}) =_{\mathrm{DF}} \bigvee_{n<\omega} \mathrm{WP}(\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od}^n, \mathbf{R})$$

3.3 Proof-Theoretic Refinement

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

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

can be proved for *every* formula $\mathbf{R}$. Back and von Wright [5] and Morgan [31,32] use a second order logic to carry out this proof. In a second order logic we can quantify over boolean predicates, so the formula to be proved is:

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

This approach has a serious drawback: second order logic is *incomplete* which means that there is not necessarily a proof for every valid transformation. Back [3,4] gets round this difficulty by extending the logic with a new predicate symbol G_W to represent the postcondition and carrying out the proof of the formula

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

in the extended first order logic.

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

Definition 3.1 Let $\mathbf{S}_1, \mathbf{S}_2 : V \to W$ and let $\mathbf{x}$ be a list of all the variables in W. Let $\mathbf{x}''$ be a list of new variables (not appearing in $\mathbf{S}_1$, $\mathbf{S}_2$, V or W) which is the same length as $\mathbf{x}$. Let Δ be a set of sentences (formulae with no free variables). If the formulae $\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}'') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}'')$ and $\mathrm{WP}(\mathbf{S}_1, \textsf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \textsf{true})$ are provable from Δ, then we say that $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ and write: $\Delta \vdash \mathbf{S}_1 \ \leq\ \mathbf{S}_2$.

(Note: the formula $\mathbf{x} \neq \mathbf{x}''$ above comparing two lists of values, so it is true whenever they differ in any position, i.e. whenever the value of any variable in $\mathbf{x}$ differs from the value of the corresponding variable in $\mathbf{x}''$).

As a simple example, let $\mathbf{S}_1$ be the nondeterministic choice ($x := 1 \sqcap x := 2$) and let $\mathbf{S}_2$ be the simple assignment $x := 1$. We claim that $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$. Clearly:

$$\mathrm{WP}(\mathbf{S}_1, \textsf{true}) \iff \textsf{true} \iff \mathrm{WP}(\mathbf{S}_2, \textsf{true})$$

Now:

$$\mathrm{WP}(\mathbf{S}_1, x \neq x') \iff \mathrm{WP}(x := 1, x \neq x') \wedge \mathrm{WP}(x := 2, x \neq x') \iff 1 \neq x' \wedge 2 \neq x'$$

while:

$$\mathrm{WP}(\mathbf{S}_2, x \neq x') \iff 1 \neq x'$$

So $\mathrm{WP}(\mathbf{S}_1, x \neq x') \Rightarrow \mathrm{WP}(\mathbf{S}_2, x \neq x')$ and the refinement is proved.

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

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

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

For the remainder of this paper we will fix on a countable list of variables $\mathcal{V} = \langle v_1, v_2, \dots \rangle$ from which all initial and final state spaces and WSL program variables will be taken. The corresponding list of *primed variables* $\mathcal{V}' = \langle v'_1, v'_2, \dots \rangle$ will be reserved for use in specification statements and the list of *double primed variables* $\mathcal{V}'' = \langle v''_1, v''_2, \dots \rangle$ will be reserved for generating the special postcondition needed for Definition 3.1. If W is a finite set of variables, then define $\overrightarrow{W}$ to be the list of variables in W in the order in which they appear in $\mathcal{V}$. Define $\overrightarrow{W}'$ to be the corresponding list of primed variables and $\overrightarrow{W}''$ the list of double primed variables. For example, if W is the set $\{v_3, v_7, v_2\}$ then $\overrightarrow{W} = \langle v_2, v_3, v_7 \rangle$, $\overrightarrow{W}' = \langle v'_2, v'_3, v'_7 \rangle$ and $\overrightarrow{W}'' = \langle v''_2, v''_3, v''_7 \rangle$. The formula $\overrightarrow{W} \neq \overrightarrow{W}''$ is:

$$v_2 \neq v''_2 \wedge v_3 \neq v''_3 \wedge v_7 \neq v''_7$$

Then we can write Definition 3.1 as follows:

If $\mathbf{S}_1 : V \to W$ and $\mathbf{S}_2 : V \to W$ then for any set Δ of sentences, $\Delta \vdash \mathbf{S}_1 \ \leqslant \ \mathbf{S}_2$ if and only if:

$$\Delta \vdash \mathrm{WP}(\mathbf{S}_1, \textsf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \textsf{true}) \quad \text{and} \quad \Delta \vdash \mathrm{WP}(\mathbf{S}_1, \overrightarrow{W} \neq \overrightarrow{W}'') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \overrightarrow{W} \neq \overrightarrow{W}'')$$

For program equivalence, we have:

Theorem 3.2 *For any* $\mathbf{S}_1, \mathbf{S}_2 : V \to W$, $\Delta \vdash \mathbf{S}_1 \ \approx \ \mathbf{S}_2$ *if and only if:*

$$\Delta \vdash \mathrm{WP}(\mathbf{S}_1, \textsf{true}) \iff \mathrm{WP}(\mathbf{S}_2, \textsf{true}) \quad \textit{and} \quad \Delta \vdash \mathrm{WP}(\mathbf{S}_1, \overrightarrow{W} \neq \overrightarrow{W}'') \iff \mathrm{WP}(\mathbf{S}_2, \overrightarrow{W} \neq \overrightarrow{W}'')$$

3.4 Some Basic Transformations

In this section we prove some fundamental transformations which are useful for deriving programs from specifications.

An important property for any notion of refinement is the replacement property: if any component of a statement is replaced by any refinement then the resulting statement is a refinement of the original one.

Transformation 1 *General Replacement:* If $\mathbf{S}'$ is formed from $\mathbf{S}$ by replacing any selection of components by refinements of those components, then $\Delta \vdash \mathbf{S} \ \leqslant \ \mathbf{S}'$.

The next theorem shows that *any* program can be translated into an equivalent specification statement:

Transformation 2 *The Representation Theorem:* Let $\mathbf{S}\colon V \to W$, be any WSL program. Let $\mathbf{x} = \overrightarrow{W}$ and $\mathbf{x}' = \overrightarrow{W}'$. Then for any Δ we have:

$$\Delta \vdash \mathbf{S} \ \approx \ \mathbf{x} := \mathbf{x}'.(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \ \wedge \ \mathrm{WP}(\mathbf{S}, \textsf{true}));\ \textsf{remove}(\overrightarrow{V \setminus W})$$

Proof: See [63] for the proof. ∎

This theorem shows that the specification statement is sufficiently powerful to specify *any* computer program we may choose to develop. It would also appear to solve all reverse engineering problems at a stroke, and therefore be a great aid to software maintenance and reverse engineering. However, for programs which contain loops or recursion, the specification given by this theorem will include infinite formulae: and this limits its practical application. The theorem is still very useful when combined with the Replacement Property and applied to fragments of programs which do not contain loops or recursion. See [66] for applications to semantic slicing. It also has practical applications in analysing commercial assembler systems. One study [67] found that over 40% of all commercial assembler modules contained no loops or recursion.

The FermaT transformation system [62,63] includes two transformations which are forwards and backwards implementations of Transformation 2 for non-iterative, non-recursive statements. The transformation Prog_To_Spec converts any statement which does not use loops, **goto**s or procedures into an equivalent specification statement. For example, the program:

$y := 2 * y;$
if $y > 0$ **then** $x := x + z$ **fi**

is transformed to the equivalent specification:

$\langle x, y \rangle := \langle x', y' \rangle.(((x' = x \ \vee \ y > 0) \ \wedge \ x' = x + z \ \vee \ x' = x \ \wedge \ y \leqslant 0) \ \wedge \ y' = 2 * y)$

while the program:

$r := q + 1;$
if $p = q$
  then $z := r - 1;\ x := z + 3;\ y := 10$

elsif $p \neq r - 1$
  then $y := 10;\ x := r + y - 8$
  else $x := q$ **fi**

is transformed to the equivalent specification:
$\langle r, x, y, z\rangle := \langle r', x', y', z'\rangle.((z' = q \ \wedge\ p = q \ \vee\ z' = z \ \wedge\ p \neq q)$
$\wedge\ r' = q + 1 \ \wedge\ x' = q + 3 \ \wedge\ y' = 10)$

The other transformation, Refine_Spec, takes a specification statement and refines it into a program containing assertions, **if** statements, simple assignments and simpler specification statements as necessary. For example, the specification above is refined into the equivalent program:
if $p = q$ **then** $z := q$ **fi**;
$\langle r := q + 1, x := q + 3, y := 10\rangle$

This process of abstraction followed by refinement is particularly useful for implementing conditioned semantic slicing, see [61] for examples.

Our next transformation is a key step in the derivation of an algorithm from a specification. This transformation, together with the others in this section, can be used to implement recursive specifications as recursive procedures, to introduce recursion into an abstract program to get a "more concrete" program (i.e. closer to a programming language implementation), and to transform a given recursive procedure into a different form.

Suppose we have a statement $\mathbf{S}$ which we wish to transform into the recursive procedure $(\mu X.\mathbf{S}')$. We claim that this is possible whenever:

1. The statement $\mathbf{S}$ is refined by the (non-recursive) statement $\mathbf{S}'[\mathbf{S}/X]$. This denotes $\mathbf{S}'$ with all occurrences of X replaced by $\mathbf{S}$. In other words, if we replace each recursive call in the body of the procedure by a copy of the specification then we get a refinement of the specification;
2. We can find an expression $\mathbf{t}$ (called the *variant function*) whose value is reduced before each occurrence of $\mathbf{S}$ in $\mathbf{S}'[\mathbf{S}/X]$.

The expression $\mathbf{t}$ need not be integer valued: any set Γ which has a well-founded order $\trianglelefteq$ is suitable. To prove that the value of $\mathbf{t}$ is reduced it is sufficient to prove that if $\mathbf{t} \trianglelefteq t_0$ initially, then the assertion $\{\mathbf{t} \lhd t_0\}$ can be inserted before each occurrence of $\mathbf{S}'$ in $\mathbf{S}[\mathbf{S}'/X]$. The theorem combines these two requirements into a single condition:

Transformation 3 *Recursive Implementation:* If $\trianglelefteq$ is a well-founded partial order on some set Γ and $\mathbf{t}$ is a term giving values in Γ and t_0 is a variable which does not occur in $\mathbf{S}$ or $\mathbf{S}'$ then if

$$\Delta \vdash \{\mathbf{t} \trianglelefteq t_0\};\ \mathbf{S} \ \leq\ \mathbf{S}'[\{\mathbf{t} \lhd t_0\};\ \mathbf{S}/X]$$

then $\Delta \vdash \mathbf{S} \ \leq\ (\mu X.\mathbf{S}')$

Proof: See [49]. ■

The proposed body of the recursive procedure ($\mathbf{S}$ above) does not have to be "pulled out of thin air", followed by a tedious verification of the refinement $\Delta \vdash \mathbf{S} \ \leq\ \mathbf{S}'[\mathbf{S}/X]$ Instead, we can derive $\mathbf{S}'[\mathbf{S}/X]$ from $\mathbf{S}$ by applying a sequence of transformations, with the aim of ensuring that ultimately we will be able to insert the assertion $\{\mathbf{t} \lhd t_0\}$ before each copy of $\mathbf{S}$. The next few transformations are particularly useful for this derivation process:

Transformation 4 *Splitting A Tautology:* We can convert any statement to a conditional statement with identical branches. For any statement $\mathbf{S} : V \to W$ and any condition $\mathbf{B}$ whose free variables are in V:

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

Transformation 5 *Move Assertion:* If $\mathbf{B} \ \wedge\ \mathrm{WP}(\mathbf{S}, \textsf{\textbf{true}}) \iff \mathrm{WP}(\mathbf{S}, \mathbf{B})$ then $\Delta \vdash \mathbf{S};\ \{\mathbf{B}\} \ \approx\ \{\mathbf{B}\};\ \mathbf{S}$

Transformation 6 *Swap Statements*
If no variable modified in $\mathbf{S}_1$ is used in $\mathbf{S}_2$ and no variable modified in $\mathbf{S}_2$ is used in $\mathbf{S}_1$ then:

$$\Delta \vdash \mathbf{S}_1;\ \mathbf{S}_2\ \approx\ \mathbf{S}_2;\ \mathbf{S}_1$$

The proof is by an induction on the recursion nesting and the structure of $\mathbf{S}_2$ using Transformation 5 as one of the base cases.

4 Slicing

Weiser [68] 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**. Weiser's algorithm for slicing is based on propagating control and data dependencies. His algorithm does not take into account the "calling context" of a procedure call, in particular: (a) if a slice includes one call-site on a procedure then the slice includes all call-sites on the procedure, and (b) if a slice includes one parameter, it must include all parameters. Horwitz et. al. [23] present an algorithm for computing slices based on tracking control and data dependencies in the system dependence graph. Binkley [9] presents an improved algorithm which accounts for the calling context of each procedure call.

Reps and Yang [41] and Reps [40] *define* a program slice in terms of transitive control and data dependencies in the program dependence graph.

Venkatesh [48] defines a slice in terms of the denotational semantics, but modified the semantics to propagate "contamination" through the program in a way which is analogous to data dependency.

Rodrigues and Barbosa [42] proposed a slicing approach for functional programs based on a projection or a hiding function which, once composed with the original program, leads to the identification of the intended slice. It is unclear whether this approach will scale to real, complex examples.

Survey articles by Binkley and Gallagher [13], Tip [46] and Binkley and Harman [11] include extensive bibliographies.

4.1 Slicing Relations defined by Semantic Equivalence

The concept of an *Amorphous Slice* was first introduced in 1997 by Harman and Danicic [20] as a combination of a computable partial order and an (otherwise unspecified) semantic equivalence relation. Given that the partial order is required to be a computable function, it cannot be a function on the program semantics: for example, it cannot be refinement since semantic refinement is not a computable relation. (A program which computes the refinement relation could be used to solve the Halting Problem, which is known to be impossible [47]). So, the partial order must be a purely syntactic relation: although this fact is not made explicit in the paper.

In 2000 Hatcliff [21] pointed out that: "In many cases in the slicing literature, the desired correspondence between the source program and the slice is not formalized because the emphasis is often on applications rather than foundations, and this also leads to subtle differences between presentations." They give a formal definition of a slice as any program whose projected execution trace is identical to the projected execution trace of the original program.

In 2001 one of the present authors [56] defined a slice in WSL as a combination of statement deletion plus WSL semantic equivalence on the variables of interest. A "semantic slice" was defined purely in terms of semantic equivalence, by dropping the syntactic requirement. This flawed definition was later modified, using the concept of *semi-refinement* (see Section 4.2).

In 2003 Binkley, Harman et al [12] redefined an "Amorphous slice" to be as a combination of a syntactic ordering and a semantic equivalence relation using a "lazy semantics": where the slice of a terminating program may be a non-terminating program, and vice-versa.

In 2006 the same authors [10] defined *all* forms of slicing as a combination of a pre-order on the syntax of the programs and an equivalence relation on the semantics. While this might appear to provide a huge number of potential candidates for useful slicing relations, it turns out that there is *no* semantic equivalence relation which is suitable for defining a useful program slice! See [64] for the proof.

In [64] we define a set of properties of slicing definitions and examine various published definitions against this set of properties. We prove that two of the most basic and fundamental properties of slicing, together with a basic property of the programming language semantics, are sufficient to completely characterise the semantics of the slicing relation: in the sense that there is only one possible slicing relation which meets all the requirements. The two properties are:

1. Behaviour Preservation: If the original program terminates on some initial state, then the slice must also terminate on that state and give the same values to the variables of interest;
2. Truncation: A statement at the very end of the program which does not assign to any variable of interest can be deleted, and the result is a valid slice.

This semantic relation for slicing is called *semi-refinement*.

4.2 Semi-Refinement

Definition 4.1 Let $\mathbf{S}_1, \mathbf{S}_2 : V \to W$. Statement $\mathbf{S}_2$ is a *semi-refinement* of $\mathbf{S}_1$, denoted $\Delta \vdash \mathbf{S}_1 \preccurlyeq \mathbf{S}_2$, if and only if:

$$\Delta \vdash \mathbf{S}_1 \ \approx\ \{\mathrm{WP}(\mathbf{S}_1, \textsf{true})\};\ \mathbf{S}_2$$

For initial states on which $\mathbf{S}_1$ terminates, then $\mathbf{S}_2$ must be equivalent to $\mathbf{S}_1$. Otherwise (i.e. when $\mathbf{S}_1$ does not terminate), $\mathbf{S}_2$ can do anything at all.

A semi-refinement can be characterised in terms of the weakest preconditions using Theorem 3.2:

$$\begin{array}{rl} \Delta \vdash \mathbf{S}_1 \preccurlyeq \mathbf{S}_2 \quad \text{iff} & \Delta \vdash \mathrm{WP}(\mathbf{S}_1, \textsf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \textsf{true}) \\ \text{and} & \Delta \vdash \mathrm{WP}(\mathbf{S}_1, \overrightarrow{W} \neq \overrightarrow{W}'') \Leftrightarrow (\mathrm{WP}(\mathbf{S}_1, \textsf{true}) \wedge \mathrm{WP}(\mathbf{S}_2, \overrightarrow{W} \neq \overrightarrow{W}'')) \end{array}$$

If $\Delta \vdash \mathbf{S}_1 \ \preccurlyeq\ \mathbf{S}_2$ then $\Delta \vdash \mathbf{S}_1 \ \leq\ \mathbf{S}_2$ (since deleting an assertion is a valid refinement): so any semi-refinement is also a refinement. The the converse does not hold: for example $x := 1$ is a valid refinement of the nondeterministic choice $(x := 1 \ \sqcap\ x := 2)$, but not a valid semi-refinement. Clearly, any equivalence is also a semi-refinement, so the semi-refinement relation lies strictly between a refinement and an equivalence.

An example of a semi-refinement is deleting an assertion:

Lemma 4.2 *For any formula* $\mathbf{Q}$*:* $\Delta \vdash \{\mathbf{Q}\} \ \preccurlyeq\ \textsf{skip}$

Proof: $\mathrm{WP}(\{\mathbf{Q}\}, \textsf{true}) = \mathbf{Q}$ so: $\{\mathbf{Q}\}$ is equivalent to $\{\mathrm{WP}(\{\mathbf{Q}\}, \textsf{true})\};\ \textsf{skip}$ ■

In particular, **abort** which is defined as $\{\textsf{false}\}$ can be semi-refined to **skip**.

As with refinement, semi-refinement satisfies the *replacement property*: if any component of a program is replaced by a semi-refinement then the result is a semi-refinement of the whole program. Putting these two results together, we see that deleting an assertion anywhere in a program gives a valid semi-refinement.

Ranganath et. al. [39] carefully distinguishes between non-terminating preserving slicing definitions and those which are not required to preserve non-termination. They introduce new notions of control dependence which are non-termination sensitive and others which are non-termination insensitive.

Amtoft [2] also address the issue of slicing away non-terminating code by defining a slice in terms of *weak simulation* where the slice is required to perform all observable actions of the original program, but not necessarily vice-versa.

4.3 Reduction

In [66] we present a unified mathematical framework for program slicing which places all slicing work, for sequential programs, in a sound theoretical framework. This mathematical approach has many advantages: not least of which is that it is not tied to a particular representation. In fact the mathematics provides a sound basis for *any* sequential program representation and any program slicing technique.

The framework uses of semi-refinement as the semantic relation, combined with a syntactic relation (called *reduction*) and WSL's **add** and **remove** statements. Within this framework we can give mathematical definitions for backwards slicing, conditioned slicing, static and dynamic slicing and semantic slicing as program transformations in the WSL transformation theory.

The reduction relation defines the result of replacing certain statements in a program by **skip** 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:

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

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

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

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

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

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

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

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

This last case will be used when the variable v is used in $\mathbf{S}$, but the initial value e of v is not used (because v is overwritten before it is read).

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

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

Note that the reduction relation does not allow actual deletion of statements: only replacing a statement by a **skip**. This makes it easier to match up the original program with the reduced version: the position of each statement in the reduced program is the same as that of the corresponding statement in the original program. Deleting the extra **skip**s is a trivial step. Ranganath et. al. [39] and Amtoft [2] both a similar technique (replacing deleted statements by **skip** statements) to remove statements from a control flow graph while preserving the structure of the graph, in order to facilitate correctness proofs.

4.4 Slicing Relation

In this section we will define a relation on WSL programs and state spaces which captures the concept of syntactic slicing.

We define a slice of $\mathbf{S}$ on X to be any reduction of $\mathbf{S}$ which is also a semi-refinement. The following definition is an extension of the definition in [66] in that it also defines a subset of the initial state space, for which the values of all variables outside this subset are not required.

Definition 4.3 A *Syntactic Slice* of $\mathbf{S} : V \to W$ on a set $X \subseteq W$ is any program $\mathbf{S}' : V \to W$ and set $X' \subseteq V$ such that $\mathbf{S}' \sqsubseteq \mathbf{S}$ and

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

and

$$\Delta \vdash \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(W \setminus X) \ \approx\ \mathbf{S}';\ \textsf{remove}(W \setminus X)$$

The syntactic slice defined in [66] is the special case of this relation where $Y = V$.

We define the *slicing relation* $\nleqslant$ as follows:

Definition 4.4 *Slicing Relation:* If $\mathbf{S}, \mathbf{S}' : V \to W$ and $Y \subseteq V$ and $X \subseteq W$, then:

$$\begin{aligned}\Delta \vdash \mathbf{S}' \ {}_{Y}\!\nleqslant_{X} \mathbf{S} \quad \text{iff} \quad & \mathbf{S}' \sqsubseteq \mathbf{S} \quad \text{and} \\ & \Delta \vdash \mathbf{S};\ \textsf{remove}(W \setminus X) \ \preccurlyeq\ \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(W \setminus X) \ \approx\ \mathbf{S}';\ \textsf{remove}(W \setminus X)\end{aligned}$$

(The vertical bar in the slicing relation is intended to connote "slicing" rather than "negation".)

As a simple example, let $\mathbf{S}$ be $z := 4;\ x := y + 1$ and let $\mathbf{S}'$ be $\textsf{skip};\ x := y + 1$. Let $V = W = \{x, y, z\}$. We claim that:

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

It is trivial that $\mathbf{S}' \sqsubseteq \mathbf{S}$. For the semantic part of the definition:

$$\begin{aligned} z := 4;\ x := y + 1;\ \textsf{remove}(W \setminus X) \ &=\ z := 4;\ x := y + 1;\ \textsf{remove}(\{y, z\}) \\ &\approx\ z := 4;\ \textsf{remove}(\{z\});\ x := y + 1;\ \textsf{remove}(\{y, z\}) \\ &\approx\ \textsf{skip};\ \textsf{remove}(\{z\});\ x := y + 1;\ \textsf{remove}(\{y, z\}) \\ &\approx\ \textsf{skip};\ x := y + 1;\ \textsf{remove}(\{y, z\}) \end{aligned}$$

so the slice is valid.

The relation $\Delta \vdash \mathbf{S};\ \textsf{remove}(W \setminus X) \ \preccurlyeq\ \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(W \setminus X)$ holds when:

1. $\Delta \vdash \mathrm{WP}(\mathbf{S}, \textsf{true}) \Rightarrow \forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathbf{S}', \textsf{true})$ and
2. $\Delta \vdash \mathrm{WP}(\mathbf{S}, \overrightarrow{X} \neq \overrightarrow{X}'') \Leftrightarrow (\mathrm{WP}(\mathbf{S}, \textsf{true}) \ \wedge\ \forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathbf{S}', \overrightarrow{X} \neq \overrightarrow{X}''))$

The relation $\Delta \vdash \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(W \setminus X) \ \approx\ \mathbf{S}';\ \textsf{remove}(W \setminus X)$ holds when:

1. $\Delta \vdash \forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathbf{S}', \textsf{true}) \Leftrightarrow \mathrm{WP}(\mathbf{S}', \textsf{true})$ and
2. $\Delta \vdash \forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathbf{S}', \overrightarrow{X} \neq \overrightarrow{X}'') \Leftrightarrow \mathrm{WP}(\mathbf{S}', \overrightarrow{X} \neq \overrightarrow{X}'')$

Putting these results together, we can characterise the slicing relation in terms of weakest preconditions.

Theorem 4.5 If $\mathbf{S}, \mathbf{S}' : V \to W$ and $X \subseteq W$ and $Y \subseteq V$ then $\mathbf{S}' \ {}_{Y}\!\nleqslant_{X} \mathbf{S}$ if and only if:

1. $\mathbf{S}' \sqsubseteq \mathbf{S}$ and
2. $\Delta \vdash \mathrm{WP}(\mathbf{S}, \textsf{true}) \Rightarrow \forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathbf{S}', \textsf{true})$ and
3. $\Delta \vdash \mathrm{WP}(\mathbf{S}, \overrightarrow{X} \neq \overrightarrow{X}'') \Leftrightarrow (\mathrm{WP}(\mathbf{S}, \textsf{true}) \ \wedge\ \forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathbf{S}', \overrightarrow{X} \neq \overrightarrow{X}''))$ and
4. $\Delta \vdash \forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathbf{S}', \textsf{true}) \Leftrightarrow \mathrm{WP}(\mathbf{S}', \textsf{true})$ and
5. $\Delta \vdash \forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathbf{S}', \overrightarrow{X} \neq \overrightarrow{X}'') \Leftrightarrow \mathrm{WP}(\mathbf{S}', \overrightarrow{X} \neq \overrightarrow{X}'')$

This theorem will prove to be very useful for the development of a formal specification of a slicing program.

For our simple example where $\mathbf{S}$ is $z := 4;\ x := y + 1$ and $\mathbf{S}'$ is $\textsf{skip};\ x := y + 1$: Equation (1) in Theorem 4.5 is trivial, equations (2) and (4) are also trivial since both statements always terminate. For (3) we have:

$$\mathrm{WP}(z := 4;\ x := y + 1, x \neq x'') \iff \mathrm{WP}(z := 4, y + 1 \neq x'') \iff y + 1 \neq x''$$

and:

$$\begin{aligned}\forall x, z.\, \mathrm{WP}(\textsf{skip};\ x := y+1, x \neq x'') &\iff \forall x, z.\, \mathrm{WP}(\textsf{skip};\ y+1 \neq x'')\\ &\iff \forall x, z.\, y+1 \neq x''\\ &\iff y+1 \neq x''\end{aligned}$$

Similarly, for (5) we have:

$$\forall x, z.\, \mathrm{WP}(\textsf{skip};\ x := y+1, x \neq x'') \iff y+1 \neq x''$$

and:

$$\mathrm{WP}(\textsf{skip}, y+1 \neq x'') \iff y+1 \neq x''$$

which proves that the slice is valid.

Lemma 4.6 *For any statements* $\mathbf{S}_1, \mathbf{S}_2 : V \to W$ *and any* $X \subseteq W$*:*

$\Delta \vdash \mathbf{S}_1;\ \textsf{add}(W \setminus X) \ \preccurlyeq\ \mathbf{S}_2;\ \textsf{add}(W \setminus X)$ *iff* $\Delta \vdash \mathbf{S}_1;\ \textsf{remove}(W \setminus X) \ \preccurlyeq\ \mathbf{S}_2;\ \textsf{remove}(W \setminus X)$

Proof: First, note that:

$$\mathrm{WP}(\textsf{add}(W \setminus X);\ \textsf{remove}(W \setminus X), \mathbf{R}) \iff \forall \overrightarrow{W \setminus X}.\, \mathbf{R} \iff \mathbf{R}$$

since $\mathbf{R}$ has no free variables in $W \setminus X$, so:

$$\mathrm{WP}(\textsf{add}(W \setminus X);\ \textsf{remove}(W \setminus X), \mathbf{R}) \iff \mathrm{WP}(\textsf{remove}(W \setminus X), \mathbf{R})$$

Similarly:

$$\mathrm{WP}(\textsf{remove}(W \setminus X);\ \textsf{add}(W \setminus X), \mathbf{R}) \iff \forall \overrightarrow{W \setminus X}.\, \mathbf{R} \iff \mathrm{WP}(\textsf{add}(W \setminus X), \mathbf{R})$$

The proof now follows from the replacement property of semi-refinement. If:

$$\Delta \vdash \mathbf{S}_1;\ \textsf{add}(W \setminus X) \ \preccurlyeq\ \mathbf{S}_2;\ \textsf{add}(W \setminus X)$$

then, by the replacement property

$$\Delta \vdash \mathbf{S}_1;\ \textsf{add}(W \setminus X);\ \textsf{remove}(W \setminus X) \ \preccurlyeq\ \mathbf{S}_2;\ \textsf{add}(W \setminus X);\ \textsf{remove}(W \setminus X)$$

so, from the above:

$$\Delta \vdash \mathbf{S}_1;\ \textsf{remove}(W \setminus X) \ \preccurlyeq\ \mathbf{S}_2;\ \textsf{remove}(W \setminus X)$$

The converse follows similarly. ■

The next lemma presents some important properties of the slicing relation. These will be used later in the derivation of an algorithm for slicing.

Lemma 4.7 *Properties of the slicing relation.*

1. **Weaken Requirements:** If $X_1 \subseteq X$ and $Y \subseteq Y_1$ and $\mathbf{S}'\ {}_Y\!\!\trianglelefteq_X \mathbf{S}$ then $\mathbf{S}'\ {}_{Y_1}\!\!\trianglelefteq_{X_1} \mathbf{S}$

   Proof: We have: $\textsf{remove}(W \setminus X_1) \approx \textsf{remove}(W \setminus X);\ \textsf{remove}(W \setminus X_1)$, and $\textsf{add}(V \setminus Y) \approx \textsf{add}(V \setminus Y_1);\ \textsf{add}(V \setminus Y)$. So:

$$\begin{aligned}\Delta \vdash \mathbf{S};\ \textsf{remove}(W \setminus X_1) &\approx \mathbf{S};\ \textsf{remove}(W \setminus X);\ \textsf{remove}(W \setminus X_1)\\ &\preccurlyeq \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(W \setminus X);\ \textsf{remove}(W \setminus X_1)\\ &\approx \textsf{add}(V \setminus Y_1);\ \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(W \setminus X_1)\\ &\approx \textsf{add}(V \setminus Y_1);\ \mathbf{S}';\ \textsf{remove}(W \setminus X_1)\end{aligned}$$

   and

$$\begin{aligned}\Delta \vdash \textsf{add}(V \setminus Y_1);\ \mathbf{S}';\ \textsf{remove}(W \setminus X_1) &\approx \textsf{add}(V \setminus Y_1);\ \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(W \setminus X_1)\\ &\approx \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(W \setminus X_1)\\ &\approx \mathbf{S}';\ \textsf{remove}(W \setminus X_1)\end{aligned}$$

   which completes the proof. ■

2. **Strengthen Requirements:** If $\mathbf{S}' \ {}_{Y}\!\lessdot_{X}\ \mathbf{S}$ and variable y does not appear in $\mathbf{S}$ or $\mathbf{S}'$, then $\mathbf{S}' \ {}_{Y\cup\{y\}}\!\lessdot_{X\cup\{y\}}\ \mathbf{S}$.

   Proof: $\mathsf{add}(V \setminus Y) \approx \mathsf{add}(\langle y\rangle);\ \mathsf{add}(V \setminus Y \cup \{y\})$ and $\mathsf{remove}(W \setminus X) \approx \mathsf{remove}(W \setminus X \cup \{y\});\ \mathsf{remove}(\langle y\rangle)$. So by the premise, we have:

$$\begin{aligned} &\mathsf{add}(\langle y\rangle);\ \mathsf{add}(V \setminus (Y \cup \{y\}));\ \mathbf{S};\ \mathsf{remove}(W \setminus X \cup \{y\});\ \mathsf{remove}(\langle y\rangle) \\ &\quad \preccurlyeq\ \mathsf{add}(\langle y\rangle);\ \mathsf{add}(V \setminus Y \cup \{y\});\ \mathbf{S}';\ \mathsf{remove}(W \setminus X \cup \{y\});\ \mathsf{remove}(\langle y\rangle) \\ &\quad \approx\ \mathbf{S}';\ \mathsf{remove}(W \setminus (X \cup \{y\}));\ \mathsf{remove}(\langle y\rangle) \end{aligned}$$

   Move the $\mathsf{add}(\langle y\rangle)$ statements forwards using Transformation 6, and then delete the sequence $\mathsf{add}(\langle y\rangle);\ \mathsf{remove}(\langle y\rangle)$. The result follows. ∎

3. **Identity Slice:** If $\mathbf{S} : V \to W$ and $X \subseteq W$ then $\mathbf{S}\ {}_{V}\!\lessdot_{X}\ \mathbf{S}$

   Proof: Equations (1) to (5) in Theorem 4.5 are trivial when $\mathbf{S} = \mathbf{S}'$, $X \subseteq W$ and $Y = V$. ∎

4. **Abort:** $\mathsf{abort}\ {}_{\varnothing}\!\lessdot_{X}\ \mathsf{abort}$ and $\mathsf{skip}\ {}_{\varnothing}\!\lessdot_{X}\ \mathsf{abort}$ for any X.

   Proof: Follows from the fact that $\mathsf{abort} \preccurlyeq \mathsf{skip}$ and $\mathsf{abort} \approx \mathsf{add}(\mathbf{x});\ \mathsf{abort};\ \mathsf{remove}(\mathbf{y})$ for any $\mathbf{x}$ and $\mathbf{y}$. ∎

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

   Proof: $\mathrm{WP}(\mathsf{add}(\mathbf{x}), \mathsf{true}) \iff \mathsf{true}$ and:

$$WP(\mathsf{add}(\mathbf{x}), \overrightarrow{X} \neq \overrightarrow{X}'') = \forall \mathbf{x}.\, (\overrightarrow{X} \neq \overrightarrow{X}'') \iff \begin{cases} \overrightarrow{X} \neq \overrightarrow{X}'' & \text{if } X \cap \mathsf{vars}(\mathbf{x}) = \varnothing, \\ \mathsf{false} & \text{otherwise} \end{cases}$$

   since, if there is a variable, say y, which appears in both X and $\mathbf{x}$, then $\forall y.\,(y \neq y'')$ is false. So, equations (1) to (5) in Theorem 4.5 are trivial when X and $\mathsf{vars}(\mathbf{x})$ are not disjoint. So, assume $X \cap \mathsf{vars}(\mathbf{x}) = \varnothing$. Then $Y = X$ and:

$$\forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathsf{add}(\mathbf{x}), \overrightarrow{X} \neq \overrightarrow{X}'') \iff \forall \overrightarrow{V \setminus Y}.\, \forall \mathbf{x}.\, (\overrightarrow{X} \neq \overrightarrow{X}'') \iff \forall \mathbf{x}.\, (\overrightarrow{X} \neq \overrightarrow{X}'')$$

   since no variable in $V \setminus Y$ occurs free in $\overrightarrow{X} \neq \overrightarrow{X}''$. ∎

6. **Remove Variables:** For any set X and list of variables $\mathbf{x}$: $\mathsf{remove}(\mathbf{x})\ {}_{X}\!\lessdot_{X}\ \mathsf{remove}(\mathbf{x})$. 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.

   Proof: Follows from Theorem 4.5 since all the weakest preconditions are equivalent to true. ∎

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

   Proof: Let $\mathbf{S} = \mathbf{x} := \mathbf{x}'.\mathbf{Q}$ Recall that $\mathrm{WP}(\mathbf{S}, \mathbf{R}) \iff \exists \mathbf{x}'.\, \mathbf{Q} \wedge \forall \mathbf{x}'.\, (\mathbf{Q} \Rightarrow \mathbf{R}[\mathbf{x}'/\mathbf{x}])$, so the free variables in $\mathrm{WP}(\mathbf{S}, \overrightarrow{X} \neq \overrightarrow{X}'')$ are all in $Y \cup X''$. By definition, V and X'' are disjoint, so $\forall \overrightarrow{V \setminus Y}.\, \mathrm{WP}(\mathbf{S}, \overrightarrow{X} \neq \overrightarrow{X}'') \iff \mathrm{WP}(\mathbf{S}, \overrightarrow{X} \neq \overrightarrow{X}'')$. The result follows from Theorem 4.5. ∎

8. **Assignment:** If $x := e$ is any assignment, then: $x := e\ {}_{Y}\!\lessdot_{X}\ x := e$ where $Y = (X \setminus \{x\}) \cup \mathsf{vars}(e)$

   Proof: $x := e \approx \langle x\rangle := \langle x'\rangle.(x' = e)$, so the result follows from the last case. ∎

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

   Proof: If no variable in X is assigned in $\mathbf{S}$, then the final value of each variable in X is the same as the initial value. (This rather obvious fact can be formally proved by induction on the structure of $\mathbf{S}$). Then:

$$\mathrm{WP}(\mathbf{S}, \overrightarrow{X} \neq \overrightarrow{X}'') \iff \overrightarrow{X} \neq \overrightarrow{X}'' \iff \mathrm{WP}(\textsf{skip}, \overrightarrow{X} \neq \overrightarrow{X}'')$$

   and:

$$\forall \overrightarrow{V \setminus Y}. \mathrm{WP}(\mathbf{S}, \overrightarrow{X} \neq \overrightarrow{X}'') \iff \overrightarrow{X} \neq \overrightarrow{X}'' \iff \mathrm{WP}(\textsf{skip}, \overrightarrow{X} \neq \overrightarrow{X}'')$$

   The result follows from Theorem 4.5. ■

10. **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 $\mathbf{S}'_1\ {}_Y\!\lessdot_{X_1} \mathbf{S}_1$ and $\mathbf{S}'_2\ {}_{X_1}\!\lessdot_X \mathbf{S}_2$ then:

$$(\mathbf{S}'_1;\ \mathbf{S}'_2)\ \ {}_Y\!\lessdot_X\ \ (\mathbf{S}_1;\ \mathbf{S}_2)$$

   Proof:

$$\begin{aligned}
\mathbf{S}_1;\ \mathbf{S}_2;\ \textsf{remove}(W \setminus X) &\preccurlyeq \mathbf{S}_1;\ \textsf{add}(V_1 \setminus Y_1);\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X)
\end{aligned}$$

   by premise

$$\preccurlyeq \mathbf{S}_1;\ \textsf{remove}(V_1 \setminus Y_1);\ \textsf{add}(V_1 \setminus Y_1);\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X)$$

   by Lemma 4.6

$$\begin{aligned}
\preccurlyeq\ & \textsf{add}(V \setminus Y);\ \mathbf{S}'_1;\ \textsf{remove}(V_1 \setminus Y_1); \\
& \textsf{add}(V_1 \setminus Y_1);\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X)
\end{aligned}$$

   by premise

$$\preccurlyeq \textsf{add}(V \setminus Y);\ \mathbf{S}'_1;\ \textsf{add}(V_1 \setminus Y_1);\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X)$$

   by Lemma 4.6

$$\begin{aligned}
\preccurlyeq\ & \textsf{add}(V \setminus Y);\ \mathbf{S}'_1;\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X) \\
\approx\ & \textsf{add}(V \setminus Y);\ \mathbf{S}'_1;\ \textsf{add}(V_1 \setminus Y_1);\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X) \\
\approx\ & \textsf{add}(V \setminus Y);\ \mathbf{S}'_1;\ \textsf{remove}(V_1 \setminus Y_1); \\
& \textsf{add}(V_1 \setminus Y_1);\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X) \\
\approx\ & \mathbf{S}'_1;\ \textsf{remove}(V_1 \setminus Y_1);\ \textsf{add}(V_1 \setminus Y_1);\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X) \\
\approx\ & \mathbf{S}'_1;\ \textsf{add}(V_1 \setminus Y_1);\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X) \\
\approx\ & \mathbf{S}'_1;\ \mathbf{S}'_2;\ \textsf{remove}(W \setminus X)
\end{aligned}$$

   and the result is proved. ■

11. **Deterministic Choice:** If $\mathbf{S}_i, \mathbf{S}'_i : V \to W$, for $1 \leqslant i \leqslant n$ and $X \subseteq W$, $Y_i \subseteq V$ are such that $\mathbf{S}'_i\ {}_{Y_i}\!\lessdot_X \mathbf{S}_i$ and $\mathbf{B}_1, \dots, \mathbf{B}_{n-1}$ are any formulae, then:

$$\textsf{if } \mathbf{B}_1 \textsf{ then } \mathbf{S}'_1 \textsf{ elsif } \dots \textsf{ else } \mathbf{S}'_n \textsf{ fi}\ \ {}_Y\!\lessdot_X\ \ \textsf{if } \mathbf{B}_1 \textsf{ then } \mathbf{S}_1 \textsf{ elsif } \dots \textsf{ else } \mathbf{S}_n \textsf{ fi}$$

   where $Y = Y_1 \cup \dots \cup Y_n \cup \textsf{vars}(\mathbf{B}_1) \cup \dots \cup \textsf{vars}(\mathbf{B}_{n-1})$.

   Proof: We will prove the case for $n = 2$, the proof of the general case can then be proved by induction on n.

$$\begin{aligned}
&\textsf{if } \mathbf{B}_1 \textsf{ then } \mathbf{S}_1 \textsf{ else } \mathbf{S}_2 \textsf{ fi};\ \textsf{remove}(W \setminus X) \\
&\qquad \approx \textsf{if } \mathbf{B}_1 \textsf{ then } \mathbf{S}'_1;\ \textsf{remove}(W \setminus X) \textsf{ else } \mathbf{S}'_2;\ \textsf{remove}(W \setminus X) \textsf{ fi}
\end{aligned}$$

   by expanding the **if** statement forwards

$$\preccurlyeq \textbf{ if } \mathbf{B}_1 \textbf{ then add}(V \setminus Y_1);\ \mathbf{S}_1';\ \textbf{remove}(W \setminus X)$$
$$\textbf{else add}(V \setminus Y_2);\ \mathbf{S}_2';\ \textbf{remove}(W \setminus X) \textbf{ fi}$$

by premise

$$\preccurlyeq \textbf{ if } \mathbf{B}_1 \textbf{ then add}(V \setminus Y);\ \mathbf{S}_1';\ \textbf{remove}(W \setminus X)$$
$$\textbf{else add}(V \setminus Y);\ \mathbf{S}_2';\ \textbf{remove}(W \setminus X) \textbf{ fi}$$

by Weaken Requirements, since $Y_1 \subseteq Y$ and $Y_2 \subseteq Y$

$$\approx \textbf{ add}(V \setminus Y);\ \textbf{if } \mathbf{B}_1 \textbf{ then } \mathbf{S}_1' \textbf{ else } \mathbf{S}_2' \textbf{ fi};\ \textbf{remove}(W \setminus X) \textbf{ fi}$$

by separating code out of the **if** statement, using the fact that no variable in $\mathbf{B}_1$ is assigned in $\textbf{add}(V \setminus Y)$ since $\mathsf{vars}(\mathbf{B}_1) \subseteq Y$

$$\approx \textbf{ if } \mathbf{B}_1 \textbf{ then } \mathbf{S}_1';\ \textbf{remove}(W \setminus X) \textbf{ else } \mathbf{S}_2';\ \textbf{remove}(W \setminus X) \textbf{ fi}$$
$$\approx \textbf{ if } \mathbf{B}_1 \textbf{ then } \mathbf{S}_1' \textbf{ else } \mathbf{S}_2' \textbf{ fi};\ \textbf{remove}(W \setminus X)$$

and the result is proved. ■

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

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

This 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 chosen for preference.

Proof: Let V' and W' be the initial and final state spaces for the **var** statement. Note that, by definition, we have either $x \in V'$ and $x \in W'$ or $x \notin V'$ and $x \notin W'$.

Let y be a new variable which does not occur in $\mathbf{S}$ or $\mathbf{S}'$. Then:

$$\Delta \vdash \textbf{var } \langle x := e \rangle : \mathbf{S} \textbf{ end} \approx \begin{cases} y := x;\ x := e;\ \mathbf{S};\ x := y;\ \textbf{remove}(\langle y \rangle) & \text{if } x \in V \\ x := e;\ \mathbf{S};\ \textbf{remove}(\langle x \rangle) & \text{otherwise} \end{cases}$$

and

$$\Delta \vdash \textbf{var } \langle x := \bot \rangle : \mathbf{S} \textbf{ end} \approx \begin{cases} y := x;\ \textbf{add}(\langle x \rangle);\ \mathbf{S};\ x := y;\ \textbf{remove}(\langle y \rangle) & \text{if } x \in V \\ \textbf{add}(\langle x \rangle);\ \mathbf{S};\ \textbf{remove}(\langle x \rangle) & \text{otherwise} \end{cases}$$

Suppose $x \in Y$ and $x \notin X$. Then $X = X \setminus \{x\}$. By applying cases 6, 9 and 8 of this Lemma, working from right to left through the sequence of statements, we can deduce the following slicing relations:

$$\begin{array}{c} \textbf{skip } {}_{X}\!\lessdot_{X} \ x := y \\ \mathbf{S}' \ {}_{Y}\!\lessdot_{X \setminus \{x\}} \ \mathbf{S} \\ x := e \ {}_{(Y \setminus \{x\}) \cup \mathsf{vars}(e)}\!\lessdot_{Y} \ x := e \\ \textbf{skip } {}_{(Y \setminus \{x\}) \cup \mathsf{vars}(e)}\!\lessdot_{(Y \setminus \{x\}) \cup \mathsf{vars}(e)} \ y := x \end{array}$$

Then case 10 of this Lemma proves the result.

Now suppose $x \in Y$ and $x \in X$. Then by applying cases 2, 6 and 8 of this Lemma, we have:

$$\begin{gathered}
\textsf{\textbf{remove}}(\langle y \rangle) \ {}_X{\lessdot}_X \ \textsf{\textbf{remove}}(\langle y \rangle) \\
x := y \ {}_{(X\setminus\{x\})\cup\{y\}}{\lessdot}_X \ x := y \\
\mathbf{S}' \ {}_{Y\cup\{y\}}{\lessdot}_{(X\setminus\{x\})\cup\{y\}} \ \mathbf{S} \\
x := e \ {}_{(Y\setminus\{x\})\cup\mathsf{vars}(e)\cup\{y\}}{\lessdot}_{Y\cup\{y\}} \ x := e \\
y := x \ {}_{(Y\setminus\{x\})\cup\mathsf{vars}(e)\cup\{x\}}{\lessdot}_{(Y\setminus\{x\})\cup\mathsf{vars}(e)\cup\{y\}} \ y := x
\end{gathered}$$

Note that these two initial sets can both be represented as: $(Y \setminus \{x\}) \cup \mathsf{vars}(e) \cup (\{x\} \cap X)$.
Now suppose $x \notin Y$ and $x \notin X$:

$$\begin{gathered}
\textsf{\textbf{skip}} \ {}_X{\lessdot}_X \ x := y \\
\mathbf{S}' \ {}_Y{\lessdot}_{X\setminus\{x\}} \ \mathbf{S} \\
\textsf{\textbf{skip}} \ {}_Y{\lessdot}_Y \ x := e
\end{gathered}$$

Finally, if $x \notin Y$ and $x \in X$:

$$\begin{gathered}
x := y \ {}_{(X\setminus\{x\})\cup\{y\}}{\lessdot}_X \ x := y \\
\mathbf{S}' \ {}_{Y\cup\{y\}}{\lessdot}_{(X\setminus\{x\})\cup\{y\}} \ \mathbf{S} \\
\textsf{\textbf{skip}} \ {}_{Y\cup\{y\}}{\lessdot}_{Y\cup\{y\}} \ x := e \\
y := x \ {}_{Y\cup\{x\}}{\lessdot}_{(Y\cup\{y\}} \ y := x
\end{gathered}$$

This proves the result for the case where $x \in V$. The result is proved similarly for the case where $x \notin V$ using case 5 as appropriate. ■

13. **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:

$$\textsf{\textbf{while B do S}}' \textsf{\textbf{ od}} \ {}_Y{\lessdot}_Y \ \textsf{\textbf{while B do S od}}$$

Proof: We will prove by induction on n that:

$$\textsf{\textbf{while B do S}}' \textsf{\textbf{ od}}^n \ {}_Y{\lessdot}_Y \ \textsf{\textbf{while B do S od}}^n$$

for every $n < \omega$. Suppose the result holds for n. Let $\mathsf{DO} =$ **while B do S od** and $\mathsf{DO}' =$ **while B do S′ od**.

$$\begin{aligned}
\mathsf{DO}^{n+1}; \textsf{\textbf{remove}}(V \setminus Y) &= \textsf{\textbf{if B then S}};\ \mathsf{DO}^n \textsf{\textbf{ fi}};\ \textsf{\textbf{remove}}(V \setminus Y) \\
&\approx \textsf{\textbf{if B then S}};\ \mathsf{DO}^n;\ \textsf{\textbf{remove}}(V \setminus Y) \textsf{\textbf{ else remove}}(V \setminus Y) \textsf{\textbf{ fi}}
\end{aligned}$$

by Expand Forwards

$$\begin{aligned}
&\preccurlyeq \textsf{\textbf{if B then S}};\ \textsf{\textbf{add}}(V \setminus Y);\ \mathsf{DO}'^n;\ \textsf{\textbf{remove}}(V \setminus Y) \\
&\qquad \textsf{\textbf{else remove}}(V \setminus Y) \textsf{\textbf{ fi}}
\end{aligned}$$

by the induction hypothesis

$$\begin{aligned}
&\approx \textsf{\textbf{if B then S}};\ \textsf{\textbf{remove}}(V \setminus Y);\ \textsf{\textbf{add}}(V \setminus Y); \\
&\qquad\quad \mathsf{DO}'^n;\ \textsf{\textbf{remove}}(V \setminus Y) \\
&\qquad \textsf{\textbf{else remove}}(V \setminus Y) \textsf{\textbf{ fi}}
\end{aligned}$$

by the induction hypothesis

$$\begin{aligned}
&\approx \textsf{\textbf{if B then add}}(V \setminus Y);\ \mathbf{S}';\ \textsf{\textbf{remove}}(V \setminus Y);\ \textsf{\textbf{add}}(V \setminus Y); \\
&\qquad\quad \mathsf{DO}'^n;\ \textsf{\textbf{remove}}(V \setminus Y) \\
&\qquad \textsf{\textbf{else remove}}(V \setminus Y) \textsf{\textbf{ fi}} \\
&\approx \textsf{\textbf{add}}(V \setminus Y);\ \textsf{\textbf{if B then S}}';\ \mathsf{DO}'^n \textsf{\textbf{ fi}};\ \textsf{\textbf{remove}}(V \setminus Y)
\end{aligned}$$

since no variable in $\mathbf{B}$ is assigned in **add**$(V \setminus Y)$, since $\mathsf{vars}(\mathbf{B}) \subseteq Y$

$$
\begin{aligned}
&= \textsf{add}(V \setminus Y);\ \mathsf{DO}'^{n+1};\ \textsf{remove}(V \setminus Y)\\
&\approx \textbf{if B then } \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(V \setminus Y);\ \textsf{add}(V \setminus Y);\\
&\qquad\qquad \mathsf{DO}'^{n};\ \textsf{remove}(V \setminus Y)\\
&\qquad\quad \textbf{else } \textsf{remove}(V \setminus Y)\ \textbf{fi}\\
&\approx \textbf{if B then } \mathbf{S}';\ \mathsf{DO}'^{n};\ \textsf{remove}(V \setminus Y)\ \textbf{else } \textsf{remove}(V \setminus Y)\ \textbf{fi}\\
&\approx \textbf{if B then } \mathbf{S}';\ \mathsf{DO}'^{n}\ \textbf{fi};\ \textsf{remove}(V \setminus Y)\\
&= \mathsf{DO}'^{n+1};\ \textsf{remove}(V \setminus Y)
\end{aligned}
$$

The result follows from the generalised induction rule for semi-refinement. ■

5 Specification of a Slicing Algorithm

In this section we will show how to develop a specification statement which captures the concept of a program slice in WSL.

Recall that a syntactic slice $\mathbf{S} : V \to W$ on a set $X \subseteq W$ of variables is any program $\mathbf{S}' : V \to W$ and set of variables $Y \subseteq V$ such that $\mathbf{S}' \sqsubseteq \mathbf{S}$ and $\mathbf{S}'\ {}_Y\!\preccurlyeq_X \mathbf{S}$, i.e.

$$\mathbf{S}' \sqsubseteq \mathbf{S} \quad \text{and} \quad \Delta \vdash \mathbf{S};\ \textsf{remove}(W \setminus X)\ \preccurlyeq\ \textsf{add}(V \setminus Y);\ \mathbf{S}';\ \textsf{remove}(W \setminus X)\ \preccurlyeq\ \mathbf{S}';\ \textsf{remove}(W \setminus X)$$

In [66] we gave a different definition of syntactic slice, which is equivalent to setting $Y = V$ above. In this paper, we need to calculate a better approximation for the set of variables whose initial values are required by the sliced program. It is impossible to define an algorithm which is guaranteed to compute *only* the variables which are actually required (this is due to the non-computability of the Halting Problem [47]) but a good approximation will allow us to slice a sequence of statements by slicing each statement in turn, starting with the last statement in the sequence.

To convert the formal definition of program slicing into a specification statement which defines program for slicing a subset of WSL, we need the following:

1. A way to represent a WSL program in a WSL variable;
2. A way to represent the relation $\sqsubseteq$ as a WSL predicate on variables containing WSL programs;
3. A way to represent the relation $\preccurlyeq$ as a WSL predicate on variables containing WSL programs.

Any algorithm can only be applied in practice to a finite amount of data, so we can restrict consideration to *finite* WSL programs. A finite WSL program can be represented as a finite list of symbols taken from a countable set of symbols. Therefore the set of all finite WSL programs is a countable set, and we can create a countable sequence of programs which includes *every* finite WSL program.

5.1 Storing a WSL program in a variable

For simplicity, in this paper we will consider the subset of the WSL language consisting of these constructs:

- Specification Statement: $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$
- Assignment: $x := e$
- Add variables: **add**($\mathbf{x}$)
- Remove variables: **remove**($\mathbf{x}$)
- Skip statement: **skip**
- Abort statement: **abort**
- Assertion $\{\mathbf{Q}\}$

- Sequence: $\mathbf{S}_1;\ \mathbf{S}_2;\ \ldots;\ \mathbf{S}_n$
- Conditional statement: **if** $\mathbf{B}_1$ **then** $\mathbf{S}_1$ **elsif** $\mathbf{B}_2$ **then** $\mathbf{S}_2$ **elsif** … **elsif** $\mathbf{B}_n$ **then** $\mathbf{S}_n$ **fi**
- While loop: **while B do S od**
- Initialised local variable: **var** $\langle x := e \rangle$: **S end**
- Uninitialised local variable: **var** $\langle x := \bot \rangle$: **S end**

We will represent each WSL *item* (statement, expression, condition etc.) as a list:

$$\langle s, v, c_1, c_2, \ldots, c_n \rangle$$

where s is the *specific type* of the item (taken from a finite set of distinct types), v is the value of the item (if any) and $c_1, \ldots, c_n$ are the components (if any). For example, the number 3 is represented as a WSL item $\langle \texttt{Number}, 3 \rangle$. The ith component of an item is therefore the element at position $i+2$ in the representation of the item. We use the empty sequence $\langle\rangle$ as the value for an item which has no value. For example, the program:
if $x = y$ **then** $z := 1$ **else** $z := 2$ **fi**

is represented as the sequence:
$\langle \texttt{Cond}, \langle\rangle, \langle \texttt{Guarded}, \langle\rangle, \langle \texttt{Equal}, \langle\rangle, \langle \texttt{Variable}, \text{“x”} \rangle, \langle \texttt{Variable}, \text{“y”} \rangle \rangle,$
$\langle \texttt{Statements}, \langle\rangle, \langle \texttt{Assignment}, \langle\rangle, \langle \texttt{Var\_Lvalue}, \text{“z”} \rangle, \langle \texttt{Number}, 1 \rangle \rangle \rangle \rangle$
$\langle \texttt{Guarded}, \langle\rangle, \langle \texttt{True}, \langle\rangle \rangle,$
$\langle \texttt{Statements}, \langle\rangle, \langle \texttt{Assignment}, \langle\rangle, \langle \texttt{Var\_Lvalue}, \text{“z”} \rangle, \langle \texttt{Number}, 2 \rangle \rangle \rangle \rangle \rangle$

Each specific type has a corresponding generic type, for example `While`, the type of a **while** loop, has generic type `Statement`, while `Number` has generic type `Expression`. The abstract syntax of the internal representation is very simple: the specific type of an item constrains the generic type(s) of its component item(s) as follows:

1. There are either a specific number of components (possibly zero), each with a given generic type; OR
2. There are a variable number of components each of which has the same generic type.

For example, the **while** loop must have precisely two components: the first is of generic type `Condition` and the second of type `Statements`. An item of type `Statements` must have one or more components each of generic type `Statement`. Note that it is important to distinguish between a *statement sequence* which is an item of type `Statements` and a *sequence of statements* which is a sequence of items, each of type `Statement`.

The generic types used in the subset are: `Statements`, `Statement`, `Condition`, `Expression`, `Expressions`, `Lvalue`, `Lvalues` and `Guarded`. An Lvalue is anything which can appear on the left hand side of an assignment statement.

For each expression e, define $\overline{e}$ to be the internal representation of e. For example, if x is a simple variable, then $\overline{x+1}$ is the item:

$$\langle \texttt{Plus}, \langle\rangle, \langle \texttt{Variable}, \text{“x”} \rangle, \langle \texttt{Number}, 1 \rangle \rangle$$

Similarly, for each finite formula $\mathbf{B}$, define $\overline{\mathbf{B}}$ as the internal representation of $\mathbf{B}$, and for each statement $\mathbf{S}$ in the subset of WSL given above, define $\overline{\mathbf{S}}$ as the internal representation of $\mathbf{S}$. For statements, the internal representations are defined as follows:

- $\overline{\mathbf{x} := \mathbf{x}'.\mathbf{Q}} \ =_{\text{DF}}\ \langle \texttt{Spec}, \langle\rangle, \overline{\mathbf{x}}, \overline{\mathbf{Q}} \rangle$
- $\overline{x := e} \ =_{\text{DF}}\ \langle \texttt{Assignment}, \langle\rangle, \overline{x}, \overline{e} \rangle$
- $\overline{\textsf{\textbf{add}}(\mathbf{x})} \ =_{\text{DF}}\ \langle \texttt{Add}, \langle\rangle, \overline{\mathbf{x}} \rangle$
- $\overline{\textsf{\textbf{remove}}(\mathbf{x})} \ =_{\text{DF}}\ \langle \texttt{Remove}, \langle\rangle, \overline{\mathbf{x}} \rangle$

- $\overline{\textsf{skip}} =_{\text{DF}} \langle \texttt{Skip}, \langle\rangle\rangle$
- $\overline{\textsf{abort}} =_{\text{DF}} \langle \texttt{Abort}, \langle\rangle\rangle$
- $\overline{\{\mathbf{Q}\}} =_{\text{DF}} \langle \texttt{Assert}, \langle\rangle, \overline{\mathbf{Q}}\rangle$
- $\overline{\mathbf{S}_1;\ \mathbf{S}_2;\ \ldots;\ \mathbf{S}_n} =_{\text{DF}} \langle \texttt{Statements}, \langle\rangle, \overline{\mathbf{S}_1}, \overline{\mathbf{S}_2}, \ldots, \overline{\mathbf{S}_n}\rangle$
- $\overline{\textsf{if } \mathbf{B}_1 \textsf{ then } \mathbf{S}_1 \textsf{ elsif } \ldots \textsf{ elsif } \mathbf{B}_n \textsf{ then } \mathbf{S}_n \textsf{ fi}} =_{\text{DF}}$
  $\langle \texttt{Cond}, \langle\rangle, \langle \texttt{Guarded}, \langle\rangle, \overline{\mathbf{B}_1}, \overline{\mathbf{S}_1}\rangle, \ldots, \langle \texttt{Guarded}, \langle\rangle, \overline{\mathbf{B}_n}, \overline{\mathbf{S}_n}\rangle\rangle$
- $\overline{\textsf{while } \mathbf{B} \textsf{ do } \mathbf{S} \textsf{ od}} =_{\text{DF}} \langle \texttt{While}, \langle\rangle, \overline{\mathbf{B}}, \overline{\mathbf{S}}\rangle$
- $\overline{\textsf{var } \langle x := e\rangle : \mathbf{S} \textsf{ end}} =_{\text{DF}} \langle \texttt{Var}, \langle\rangle, \langle \overline{x}, \overline{e}, \overline{\mathbf{S}}\rangle$
- $\overline{\textsf{var } \langle x := \bot\rangle : \mathbf{S} \textsf{ end}} =_{\text{DF}} \langle \texttt{Var}, \langle\rangle, \langle \overline{x}, \langle \texttt{Bottom}, \langle\rangle\rangle, \overline{\mathbf{S}}\rangle$

The conditional statement **if** $\mathbf{B}_1$ **then** $\mathbf{S}_1$ **elsif** ... **elsif** $\mathbf{B}_n$ **then** $\mathbf{S}_n$ **fi** is represented as a sequence of guarded items: $\langle \texttt{Cond}, \langle\rangle, G_1, \ldots, G_n\rangle$ where: $G_i = \langle \texttt{Guarded}, \langle\rangle, \overline{\mathbf{B}_i}, \overline{\mathbf{S}_i}\rangle$. If $\mathbf{B}_n$ is **true**, then the last clause is displayed as ... **else** $\mathbf{S}_n$ **fi**, unless $\mathbf{S}_n$ is a single **skip** statement, in which case the last clause is not displayed. So **if B then S fi** is represented as:

$$\langle \texttt{Cond}, \langle\rangle, \langle \texttt{Guarded}, \langle\rangle, \overline{\mathbf{B}}, \overline{\mathbf{S}}\rangle, \langle \texttt{Guarded}, \langle\rangle, \overline{\textsf{true}}, \overline{\textsf{skip}}\rangle\rangle$$

If all of the conditions are false, then the statement aborts, also an empty `Cond` statement is defined to be equivalent to **abort**.

A list of items is represented as $\langle \texttt{Sequence}, \langle\rangle, \ldots\rangle$, so for example if $\mathbf{x}$ is the list of variables $\langle x_1, \ldots, x_n\rangle$ then $\overline{\mathbf{x}} = \langle \texttt{Sequence}, \langle\rangle, \overline{x_1}, \ldots, \overline{x_n}\rangle$.

Conversely, if I is the internal representation of a statement, expression or condition, then let $\widetilde{I}$ be the corresponding statement, expression or condition.

The following $\mathcal{METAWSL}$ functions are used to construct new items and extract the components of an item. The function @ST_To_GT maps each specific type to the corresponding generic type. If I has the value $\langle s, v, c_1, c_2, \ldots, c_n\rangle$ then:

- $\textsf{@Make}(s, v, \langle c_1, \ldots, c_n\rangle) = \langle s, v, c_1, \ldots, c_n\rangle = I$
- $\textsf{@ST}(I) = s$
- $\textsf{@GT}(I) = \textsf{@ST\_To\_GT}(\textsf{@ST}(I))$
- $\textsf{@V}(I) = v$
- $\textsf{@Size}(I) = n$
- $I\hat{\ }m = c_m$
- $\textsf{@Cs}(I) = \langle c_1, \ldots, c_n\rangle$

Note: the internal representation of items in the current implementation of FermaT[1] differs slightly from the above (for efficiency reasons), but the semantics of the accessor and creator functions is identical. In the FermaT implementation, identical items may have different implementations, so items must be compared via the $\mathcal{METAWSL}$ function @Equal? instead of =.

Since the set of finite WSL programs is countable and the set of valid initial state spaces for each program is also countable, we can construct a countable list consisting of all pairs $\langle V, \mathbf{S}\rangle$ where $\mathbf{S}$ is a finite WSL program and V is a valid initial state space for $\mathbf{S}$. Fix on a particular countable list $\langle\langle V_1, \mathbf{S}_1\rangle, \langle V_2, \mathbf{S}_2\rangle, \ldots\rangle$ and let $\langle\langle v_1, S_1\rangle, \langle v_2, S_2\rangle, \ldots\rangle$ be the corresponding list of pairs of interpretations. For each state space V and program $\mathbf{S}$, define:

$$\textsf{Final}(V, \mathbf{S}) =_{\text{DF}} \begin{cases} W & \text{if } \mathbf{S} : V \to W \\ \bot & \text{if there is no } W \text{ such that } \mathbf{S} : V \to W \end{cases}$$

[1] Available from ⟨http://www.cse.dmu.ac.uk/~mward/fermat.html⟩

So for $0 < n < \omega$ we have $v_n = \overline{V_n}$, $S_n = \overline{\mathbf{S}_n}$, $V_n = \widetilde{v_n}$, $\mathbf{S}_n = \widetilde{S_n}$, and $\mathsf{Final}(V_n, \mathbf{S}_n) \neq \bot$. The list $\langle S_1, S_2, \dots \rangle$ will include every possible finite item of type `Statement` or `Statements`.

For example, S_3 might be the representation of the program consisting of a single **skip** statement, in which case: $S_3 = \langle \mathtt{Skip}, \langle\rangle\rangle$, and v_3 might the internal representation of the state space $\{b\}$, ie: $v_3 = \langle \mathtt{Set}, \langle\rangle, \langle \mathtt{Variable}, \text{“b”}\rangle\rangle$.

Note that any finite set of variables is a valid initial state space for the program **skip**, so the list $\langle V_1, V_2, \dots \rangle$ includes all finite sets of variables (with repetitions).

As a running example for the rest of this section, let $\mathbf{S}_1$ be the program $\{a = 1\};\ b := 2;\ c := 3$ and $\mathbf{S}_2$ the program **skip**; **skip**; $c := 3$ and let V_1 and V_2 both be the set of variables $\{a, b, c\}$ and V_3 be $\{c\}$. So:

$$\begin{aligned} S_1 = \langle \mathtt{Statements}, \langle\rangle, &\langle \mathtt{Assert}, \langle\rangle, \langle \mathtt{Equal}, \langle \mathtt{Variable}, \text{“a”}\rangle, \langle \mathtt{Number}, 1\rangle\rangle\rangle, \\ &\langle \mathtt{Assignment}, \langle\rangle, \langle \mathtt{Variable}, \text{“b”}\rangle, \langle \mathtt{Number}, 2\rangle\rangle, \\ &\langle \mathtt{Assignment}, \langle\rangle, \langle \mathtt{Variable}, \text{“c”}\rangle, \langle \mathtt{Number}, 3\rangle\rangle\rangle \end{aligned}$$

and:

$$\begin{aligned} S_2 = \langle \mathtt{Statements}, \langle\rangle, &\langle \mathtt{Skip}, \langle\rangle\rangle, \\ &\langle \mathtt{Skip}, \langle\rangle\rangle, \\ &\langle \mathtt{Assignment}, \langle\rangle, \langle \mathtt{Variable}, \text{“c”}\rangle, \langle \mathtt{Number}, 3\rangle\rangle\rangle \end{aligned}$$

5.2 Definition of Reduction as a WSL Condition

Our next step is to define a WSL condition (a formula in our infinitary logic) which captures the reduction relation for representations of WSL programs. We want a condition $\mathsf{Red}(I_1, I_2)$ with free variables I_1 and I_2 which is true precisely when $\widetilde{I_1} \sqsubseteq \widetilde{I_2}$, i.e. when I_1 and I_2 contain statements and I_1 is a reduction of I_2:

$$\mathsf{Red}(I_1, I_2) \ =_{\text{DF}} \ \bigvee_{0<n<\omega} (I_1 = S_n) \ \wedge \ \bigvee_{0<n<\omega} (I_2 = S_n) \ \wedge \ \bigwedge_{0<n,m<\omega} \big((I_1 = S_n \ \wedge \ I_2 = S_m) \Rightarrow \mathbf{R}_{nm}\big)$$

where:

$$\mathbf{R}_{nm} \ =_{\text{DF}} \ \begin{cases} \textbf{true} & \text{if } \mathbf{S}_n \sqsubseteq \mathbf{S}_m, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

and:

$$\bigwedge_{0<n,m<\omega} \mathbf{Q} \qquad \text{is short for} \qquad \bigwedge_{0<n<\omega} \Big(\bigwedge_{0<m<\omega} \mathbf{Q} \Big)$$

Note that we cannot simply define $\mathsf{Red}(I_1, I_2)$ as $\widetilde{I_1} \sqsubseteq \widetilde{I_2}$ since this is not a formula in the infinitary logic. The relation $\sqsubseteq$ is not necessarily an existing relation symbol in the logic (with the appropriate interpretation), neither is $\widetilde{I}$ a function symbol in the logic.

Each of the subformulae $\mathbf{R}_{nm}$ in the definition is either the formula **true** or the formula **false**, depending on whether or not the relation $\mathbf{S}_n \sqsubseteq \mathbf{S}_m$ holds. For our running example, $\mathbf{S}_2 \sqsubseteq \mathbf{S}_1$, so $\mathbf{R}_{21} = \textbf{true}$ by definition. So, we have:

$$\mathsf{Red}(S_2, S_1) \iff S_2 = S_2 \ \wedge \ S_1 = S_1 \ \wedge \ ((S_2 = S_2 \ \wedge \ S_1 = S_1) \Rightarrow \mathbf{R}_{21})$$

which is equivalent to **true**.

5.3 Definition of Refinement and Semi-Refinement as WSL Conditions

Now we want to define a WSL condition which captures the semi-refinement relation for representations of WSL programs. Recall that the semi-refinement relation $\preccurlyeq$ may be expressed in terms of weakest preconditions where the post condition is either **true** or of the form $\mathbf{x} \neq \mathbf{x}''$ for a particular sequence of variables $\mathbf{x}$.

We start by defining a WSL formula $\mathsf{wp}_\mathbf{t}(I)$ with free variable I which is true precisely when $\mathrm{WP}(\widetilde{I}, \textbf{true})$ is true. As with Red, we cannot simply define $\mathsf{wp}_\mathbf{t}(I)$ as $\mathrm{WP}(\widetilde{I}, \textbf{true})$, since this is not a formula. But it is the case that for each n, $\mathrm{WP}(\mathbf{S}_n, \textbf{true})$ is a formula. So we define:

Definition 5.1

$$\mathsf{wp}_\mathbf{t}(I) \;=_{\mathrm{DF}}\; \bigvee_{0<n<\omega} (I = S_n) \;\wedge\; \bigwedge_{0<n<\omega} \big(I = S_n \Rightarrow \mathrm{WP}(\mathbf{S}_n, \textbf{true})\big)$$

Note that the formula contains within it a copy of the formula $\mathrm{WP}(\mathbf{S}_n, \textbf{true})$ for each WSL program $\mathbf{S}_n$ in the countable list of all finite WSL programs. For our running example:

$$\mathsf{wp}_\mathbf{t}(S_1) \iff a = 1 \quad \text{and} \quad \mathsf{wp}_\mathbf{t}(S_2) \iff \textbf{true}$$

A WSL program which solves the "halting problem" for finite WSL programs is the following:

$$\mathsf{HALTS}(I) \;=_{\mathrm{DF}}\; \textbf{if } \mathsf{wp}_\mathbf{t}(I) \textbf{ then } \mathsf{halts} := 1 \textbf{ else } \mathsf{halts} := 0 \textbf{ fi}$$

This will set the variable halts to 1 if the program whose representation is give in I will halt. Otherwise it sets halts to 0. Note that this program does not contradict the noncomputability of the halting problem [47] since $\mathsf{HALTS}(I)$ is an infinitely large program, and so is not one of the programs in the list $\langle \mathbf{S}_1, \mathbf{S}_2, \dots \rangle$. It is not a "computable function", neither is it an algorithm in the usual sense.

We define a WSL formula $\mathsf{Fin}(I, v, w)$ with free variables I, v and w which is true when w contains the representation of $\mathsf{Final}(\widetilde{v}, \widetilde{I})$:

$$\begin{aligned}\mathsf{Fin}(I, v, w) \;=_{\mathrm{DF}}\; & \bigvee_{0<n<\omega} (I = S_n \;\wedge\; v = v_n) \;\wedge\; \bigvee_{0<m<\omega} (w = v_m) \\ & \wedge \bigwedge_{0<n,m<\omega} \big((v = v_n \;\wedge\; I = S_n \;\wedge\; w = v_m) \Rightarrow \mathbf{F}_{nm}\big)\end{aligned}$$

where:

$$\mathbf{F}_{nm} \;=_{\mathrm{DF}}\; \begin{cases} \textbf{true} & \text{if } V_m = \mathsf{Final}(V_n, \mathbf{S}_n), \\ \textbf{false} & \text{otherwise.} \end{cases}$$

The formula $\mathsf{Fin}(I, v, w)$ captures the trinary relation $\mathbf{S} : V \to W$ defined at the end of Section 3.

We also define a WSL formula $\mathsf{Fins}(I, v, w)$ with free variables I, v and w which is true when w contains the representation of any subset of $\mathsf{Final}(\widetilde{v}, \widetilde{I})$:

$$\begin{aligned}\mathsf{Fins}(I, v, w) \;=_{\mathrm{DF}}\; & \bigvee_{0<n<\omega} (I = S_n \;\wedge\; v = v_n) \;\wedge\; \bigvee_{0<m<\omega} (w = v_m) \\ & \wedge \bigwedge_{0<n,m<\omega} \big((I = S_n \;\wedge\; v = v_n \;\wedge\; w = v_m) \Rightarrow \mathbf{G}_{nm}\big)\end{aligned}$$

where:

$$\mathbf{G}_{nm} \;=_{\mathrm{DF}}\; \begin{cases} \textbf{true} & \text{if } V_m \subseteq \mathsf{Final}(V_n, \mathbf{S}_n), \\ \textbf{false} & \text{otherwise.} \end{cases}$$

Clearly, $\mathbf{F}_{nm} \Rightarrow \mathbf{G}_{nm}$ for all n and m, and: $\mathsf{Fin}(I, v, w) \Rightarrow \mathsf{Fins}(I, v, w)$.

For our running example: $\mathbf{F}_{11} = \mathbf{F}_{22} = \mathbf{G}_{11} = \mathbf{G}_{22} = \mathbf{G}_{13} = \mathbf{G}_{23} = \textbf{true}$, so we have $\mathsf{Fins}(S_1, v_1, v_3) \iff \textbf{true}$ and $\mathsf{Fins}(S_2, v_1, v_3) \iff \textbf{true}$.

Next, we define a WSL formula $\mathsf{wp}_{\neq}(I, v, w)$, with free variables I, v and w which is true precisely when:

$$\mathrm{WP}(\widetilde{I}, \overrightarrow{\widetilde{w}} \neq \overrightarrow{\widetilde{w}}'')$$

is true. To be more precise, I contains the representation of some statement in the list of all WSL programs of interest, say $\mathbf{S}_n$, v contains the corresponding initial state space V_n, w contains some other state space V_m where $V_m \subseteq \mathsf{Final}(V_n, \mathbf{S}_n)$, and $\mathrm{WP}(\mathbf{S}_n, \overrightarrow{V_m} \neq \overrightarrow{V_m}'')$ is true.

$$\mathsf{wp}_{\neq}(I, v, w) =_{\mathrm{DF}} \mathsf{Fins}(I, v, w) \wedge \bigwedge_{0<n,m<\omega} \big((I = S_n \wedge v = v_n \wedge w = v_m) \Rightarrow \mathrm{WP}(\mathbf{S}_n, \overrightarrow{V_m} \neq \overrightarrow{V_m}'')\big)$$

As with $\mathsf{wp}_{\mathbf{t}}$, this formula contains a copy of the formula $\mathrm{WP}(\mathbf{S}_n, \overrightarrow{V_m} \neq \overrightarrow{V_m}'')$ for each WSL program $\mathbf{S}_n$ in the sequence and each state space V_m where $V_m \subseteq \mathsf{Final}(V_n, \mathbf{S}_n)$.

For our running example:

$$\begin{aligned}
\mathsf{wp}_{\neq}(S_1, v_1, v_3) &\iff \textbf{true} \wedge \mathrm{WP}(\mathbf{S}_1, a \neq a'' \wedge b \neq b'' \wedge c \neq c'') \\
&\iff \mathrm{WP}(\{a = 1\};\ b := 2;\ c := 3, a \neq a'' \wedge b \neq b'' \wedge c \neq c'') \\
&\iff a = 1 \wedge 3 \neq c''
\end{aligned}$$

and similarly: $\mathsf{wp}_{\neq}(S_2, v_1, v_3) \iff 3 \neq c''$.

Using these formulae, and Theorem 3.2 we can define a formula $\mathsf{Refine}(I_1, I_2)$ which is true precisely when $\widetilde{I_1}$ is refined by $\widetilde{I_2}$:

Definition 5.2

$$\begin{aligned}
\mathsf{Refine}(I_1, I_2) =_{\mathrm{DF}} & (\mathsf{wp}_{\mathbf{t}}(I_1) \Rightarrow \mathsf{wp}_{\mathbf{t}}(I_2)) \\
& \wedge \exists v, w. \big(\mathsf{Fin}(I_1, v, w) \wedge \mathsf{Fin}(I_2, v, w) \wedge (\mathsf{wp}_{\neq}(I_1, v, w) \Rightarrow \mathsf{wp}_{\neq}(I_2, v, w))\big)
\end{aligned}$$

Using this formula we can construct a specification for *any* program transformation. Any valid transformation (on our finitary subset of WSL) is a refinement of the following specification:

Definition 5.3

$$\mathsf{TRANS}(I) =_{\mathrm{DF}} \langle I \rangle := \langle I' \rangle.(\mathsf{Refine}(I, I') \wedge \mathsf{Refine}(I', I))$$

To prove the correctness of the implementation of any program transformation it is sufficient to prove that the transformation is a refinement of this specification.

We now have all the machinery required to define semi-refinement as a WSL condition.

We define a formula $\mathsf{Semi}(I_1, I_2)$ which is true when $\widetilde{I_1}$ is semi-refined by $\widetilde{I_2}$ (i.e. when $\widetilde{I_2}$ is a semi-refinement of $\widetilde{I_1}$) using Definition 4.1:

Definition 5.4

$$\begin{aligned}
\mathsf{Semi}(I_1, I_2) =_{\mathrm{DF}} & (\mathsf{wp}_{\mathbf{t}}(I_1) \Rightarrow \mathsf{wp}_{\mathbf{t}}(I_2)) \\
& \wedge \exists v, w. \big(\mathsf{Fin}(I_1, v, w) \wedge \mathsf{Fin}(I_2, v, w) \\
& \qquad \wedge (\mathsf{wp}_{\neq}(I_1, v, w) \Leftrightarrow (\mathsf{wp}_{\mathbf{t}}(I_1) \wedge \mathsf{wp}_{\neq}(I_2, v, w)))\big)
\end{aligned}$$

5.4 Definition of the Slicing Relation as a WSL Condition

To define the slicing relation as a WSL condition we need a way to express conditions such as:

$$\forall (V \setminus Y).\, \mathrm{WP}(\mathbf{S}, \mathbf{R})$$

where V is the initial state space and Y is another set of variables.

Define the formula $\mathsf{wpa}_{\mathbf{t}}(y, I, v)$ which is true when:

$$\forall(\widetilde{v} \setminus \widetilde{y}).\, \mathrm{WP}(\widetilde{I}, \textbf{true})$$

is true:

$$\begin{aligned}\mathsf{wpa}_{\mathbf{t}}(y, I, v) \;=_{\mathrm{DF}}\; & \bigvee_{0<k<\omega} (y = v_k) \;\wedge\; \bigvee_{0<n<\omega} (v = v_n) \\ & \wedge \bigwedge_{0<k,n<\omega} \big((I = S_n \;\wedge\; v = v_n \;\wedge\; y = v_k) \Rightarrow \forall(V_n \setminus V_k).\, \mathrm{WP}(\mathbf{S}_n, \textbf{true})\big)\end{aligned}$$

For our running example, recall that V_3 is the set $\{c\}$, and v_3 is the representation of V_3. Then:

$$\mathsf{wpa}_{\mathbf{t}}(v_3, S_2, v_1) \iff \forall a, b.\, \mathrm{WP}(\mathbf{S}_2, \textbf{true}) \iff \textbf{true}$$

Define the formula $\mathsf{wpa}_{\neq}(y, I, v, w)$ which is true when:

$$\forall(\widetilde{v} \setminus \widetilde{y}).\, \mathrm{WP}(\widetilde{I}, \overrightarrow{\widetilde{w}} \neq \overrightarrow{\widetilde{w}}'')$$

is true:

$$\begin{aligned}\mathsf{wpa}_{\neq}(y, I, v, w) \;=_{\mathrm{DF}}\; & \mathsf{Fins}(I, v, w) \;\wedge\; \bigvee_{0<k<\omega} (y = v_k) \\ & \wedge \bigwedge_{0<k,m,n<\omega} \big((I = S_n \;\wedge\; v = v_n \;\wedge\; w = v_m \;\wedge\; y = v_k) \\ & \qquad \Rightarrow \forall(V_n \setminus V_k).\, \mathrm{WP}(\mathbf{S}_n, \overrightarrow{V_m} \neq \overrightarrow{V_m}'')\big)\end{aligned}$$

For our running example:

$$\begin{aligned}\mathsf{wpa}_{\neq}(v_3, S_2, v_1, v_3) & \iff \forall a, b.\, \mathrm{WP}(\mathbf{S}_2, c \neq c'') \\ & \iff \forall a, b.\, \mathrm{WP}(\textbf{skip};\ \textbf{skip};\ c := 3, c \neq c'') \\ & \iff \forall a, b.\, (3 \neq c'') \\ & \iff 3 \neq c''\end{aligned}$$

With the aid of these definitions and Theorem 4.5 we define a formula with free variables I, J, x and y which is true precisely when $\widetilde{J}\ {}_{\widetilde{y}}\!\!\nleq_{\widetilde{x}} \widetilde{I}$ is true:

$$\begin{aligned}\mathsf{Slice}(I, J, x, y) \;=_{\mathrm{DF}}\; \mathsf{Red}(J, I) \;\wedge\; \exists v.\, \big(& \mathsf{Fins}(I, v, x) \;\wedge\; \mathsf{Fins}(J, v, x) \\ & \wedge\; (\mathsf{wp}_{\mathbf{t}}(I) \Rightarrow \mathsf{wpa}_{\mathbf{t}}(y, J, v)) \\ & \wedge\; (\mathsf{wp}_{\neq}(I, v, x) \Leftrightarrow \mathsf{wp}_{\mathbf{t}}(I) \;\wedge\; \mathsf{wpa}_{\neq}(y, J, v, x)) \\ & \wedge\; (\mathsf{wpa}_{\neq}(y, J, v, x) \Leftrightarrow \mathsf{wp}_{\neq}(J, v, x))\big)\end{aligned}$$

For our running example, we have:

$$\begin{aligned}\mathsf{Slice}(S_1, S_2, v_3, v_3) \iff \mathsf{Red}(S_2, S_1) \;\wedge\; \exists v.\, \big(& \mathsf{Fins}(S_1, v, v_3) \;\wedge\; \mathsf{Fins}(S_2, v, v_3) \\ & \wedge\; (\mathsf{wp}_{\mathbf{t}}(S_1) \Rightarrow \mathsf{wpa}_{\mathbf{t}}(v_3, S_2, v)) \\ & \wedge\; (\mathsf{wp}_{\neq}(S_1, v, v_3) \Leftrightarrow \mathsf{wp}_{\mathbf{t}}(S_1) \;\wedge\; \mathsf{wpa}_{\neq}(v_3, S_2, v, v_3)) \\ & \wedge\; (\mathsf{wpa}_{\neq}(v_3, S_2, v, v_3) \Leftrightarrow \mathsf{wp}_{\neq}(S_2, v, v_3))\big)\end{aligned}$$

We know that $\mathsf{Red}(S_2, S_1) \iff \textbf{true}$ (see above), so we will turn our attention to the existential quantifier. Put $v = v_1$ and use the results of the running example to simplify the quantifier to:

$$\begin{aligned}\big(& \textbf{true} \;\wedge\; \textbf{true} \\ & \wedge\; (a = 1 \Rightarrow \textbf{true}) \\ & \wedge\; ((a = 1 \;\wedge\; 3 \neq c'') \Leftrightarrow a = 1 \;\wedge\; 3 \neq c'') \\ & \wedge\; (3 \neq c'' \Leftrightarrow 3 \neq c'')\big)\end{aligned}$$

which is true. So $\mathsf{Slice}(S_1, S_2, v_3, v_3)$ is true, which means that S_2 is a valid slice of S_1 on v_3:

$$\textbf{skip};\ \textbf{skip};\ c := 3\ {}_{\{c\}}{\preccurlyeq}_{\{c\}}\ \{a = 1\};\ b := 2;\ c := 3$$

Now that we have defined a single formula which captures the meaning of "a valid slice", we can define the specification of a slicing program as a single specification statement:

$$\mathsf{SLICE}\ =_{\text{DF}}\ \langle I, x\rangle := \langle I', x'\rangle.\mathsf{Slice}(I, I', x, x')$$

This will assign new values to the variables I and x such that the new value of I is a slice of the original value on the set of variables given in x, and the new value of x is a valid set of required initial variables for this slice. By Theorem 4.5:

$$\Delta \vdash \mathsf{Slice}(I, J, x, y) \iff \Delta \vdash \widetilde{J}\ {}_{\widetilde{y}}{\preccurlyeq}_{\widetilde{x}}\ \widetilde{I}$$

The specification is completely nondeterministic in the sense that *any* valid slice is a possible output of the specification.

Suppose the function $\mathsf{vars}(I)$ returns a representation of the set of all variables in the program whose representation is given in I. Consider the program:

$$x := \mathsf{vars}(I) \cup x$$

Since any program is always a valid slice of itself, this program is a valid slicing algorithm. The program in I is unchanged and the set of required initial variables simply has all the variables in the program added to it, regardless of whether they are needed or not. So this program is a valid refinement of SLICE. (But not a terribly useful one!)

6 Deriving an Implementation

In this section we will use program transformations to derive a (more useful) implementation of the slicing specification. First, consider the simple case where I contains a statement sequence. Apply Transformation 4 (*Splitting a Tautology*) to give:

if $\mathsf{@GT}(I) = \mathtt{Statements}$
  then if $\mathsf{@Size}(I) = 0$
    then SLICE
    else SLICE **fi**
  else SLICE **fi**

If I is an empty statement sequence, then I is the only possible reduction of I, and there can be no change to the set of required variables. So SLICE is equivalent to **skip**. If I contains one or more statements, then we can apply Property 10 of Lemma 4.7. We can slice the last statement and use the resulting output variable set as the input variable set for slicing the rest of the sequence (which may be empty).

First we define some functions for operating on sequences. If L the sequence $\langle l_1, l_2, \dots, l_n\rangle$ then $\ell(L) = n$ is the length of the sequence. If L is not empty, then $L[i] = l_i$ is the ith element, $L[i\,..\,j] = \langle l_i, l_{i+1}, \dots, l_j\rangle$ is a subsequence, $L[i\,..] = L[i\,..\,\ell(L)]$ is a subsequence, $\mathsf{LAST}(L) = L[\ell(L)]$ is the last element and $\mathsf{BUTLAST}(L) = L[1\,..\,\ell(L){-}1]$ is the sequence with the last element removed. For WSL items, we define these functions:

$$\begin{aligned}
\mathsf{butlast_item}(I) &=_{\text{DF}} \mathsf{@Make}(\mathsf{@ST}(I), \langle\rangle, \mathsf{BUTLAST}(\mathsf{@Cs}(I))) \\
\mathsf{last_item}(I) &=_{\text{DF}} I\hat{}\mathsf{@Size}(I) \\
\mathsf{add_item}(I, J) &=_{\text{DF}} \mathsf{@Make}(\mathsf{@ST}(I), \langle\rangle, \mathsf{@Cs}(I) \mathbin{+\!\!+} \langle J\rangle) \\
\mathsf{add_items}(I, L) &=_{\text{DF}} \mathsf{@Make}(\mathsf{@ST}(I), \langle\rangle, \mathsf{@Cs}(I) \mathbin{+\!\!+} L)
\end{aligned}$$

So, for any item I with $\mathsf{@Size}(I) > 0$:

$$I = \mathsf{add\_item}(\mathsf{butlast\_item}(I), \mathsf{last\_item}(I))$$

6.1 Statements

Now, consider the case where I is of type `Statements`. If I has no components, then SLICE is refined by **skip**. If I has one or more components, then we can apply Property 10 of Lemma 4.7 and slice the sequence by slicing the last component and using the resulting set of required variables to slice the remainder of the sequence. Therefore, the specification

$$\{\texttt{@ST}(I) = \texttt{Statements}\};\ \mathsf{SLICE}$$

is refined by the program:

if @Size$(I) = 0$
  then skip
  else var $\langle I_1 := \mathsf{butlast\_item}(I)\rangle$:
      var $\langle I_2 := \mathsf{last}(I)\rangle$:
        $\{I = \mathsf{add\_item}(I_1, I_2)\}$;
        $I := I_2$; SLICE; $I_2 := I$;
        $I := I_1$; $\{\texttt{@ST}(I) = \texttt{Statements}\}$; SLICE; $I_1 := I$;
        $I := \mathsf{add\_item}(I_1, I_2)$ **end end fi**

In applying recursive implementation transformation (Transformation 3) it is not necessary for *all* copies of the specification to be replaced by recursive calls. We can choose certain copies of the specification (for which there is a term which is reduced under some well-founded partial order) and replace these by recursive calls. In this case, the resulting recursive implementation will still contain copies of the specification.

In this case, we want to apply recursive implementation to the second copy of SLICE, using the fact that the non-negative integer function @Size(I) is reduced before the copy of the specification. So we get the following recursive procedure:

proc $\mathsf{slice}_1\ \equiv$
  if @Size$(I) = 0$
    then skip
    else var $\langle I_1 := \mathsf{butlast\_item}(I)\rangle$:
        var $\langle I_2 := \mathsf{last}(I)\rangle$:
          $\{I = \mathsf{add\_item}(I_1, I_2)\}$;
          $I := I_2$; SLICE; $I_2 := I$;
          $I := I_1$; slice_1; $I_1 := I$;
          $I := \mathsf{add\_item}(I_1, I_2)$ **end end fi.**

Before we can remove the recursion, we would like to restructure the code so that the recursive call to slice_1 is not inside a local variable statement. We can move the call out of the I_1 structure by replacing the second reference to I_1 by I, deleting the assignment to I_1 and taking code out of the structure:

var $\langle I_2 := \mathsf{last}(I)\rangle$:
  var $\langle I_1 := \mathsf{butlast\_item}(I)\rangle$:
    $I := I_2$; SLICE; $I_2 := I$; $I := I_1$ **end**;
  slice_1
  $I := \mathsf{add\_item}(I, I_2)$ **end**

Implement the I_2 local variable using a global stack L (this process is described in [54]):

$L := \langle \mathsf{last}(I)\rangle \mathbin{+\!\!+} L$;
var $\langle I_1 := \mathsf{butlast\_item}(I)\rangle$:
  $I := L[1]$; SLICE; $L[1] := I$; $I := I_1$ **end**;
slice_1;
$I := \mathsf{add\_item}(I, L[1])$; $L := L[2\,..]$

Let VAR be the **var** statement above. Now we can apply the generic recursion removal transformation from [54] to get:

proc $\mathsf{slice}_1 \equiv$
  var $\langle L := \langle\rangle\rangle$:
    do if $\mathsf{@Size}(I) = 0$
        then exit(1)
        else $L := \langle \mathsf{last}(I)\rangle \mathbin{+\!\!+} L$; VAR **fi od**;
    while $L \neq \langle\rangle$ **do**
      $I := \mathsf{add\_item}(I, L[1])$; $L := L[2\,..]$ **od end.**

The **while** loop is equivalent to: $I := \mathsf{add\_items}(I, L)$; $L := \langle\rangle$. The **do** ... **od** loop is equivalent to a **while** loop:

while $\mathsf{@Size}(I) \neq 0$ **do** $L := \langle \mathsf{last}(I)\rangle \mathbin{+\!\!+} L$; VAR **od**

I is an empty statement sequence on termination of the loop, so $I := \mathsf{add\_items}(I, L)$ is equivalent to $I := \mathsf{@Make}(\mathtt{Statements}, \langle\rangle, L)$. We get this simplified version:

proc $\mathsf{slice}_1 \equiv$
  var $\langle L := \langle\rangle\rangle$:
    while $\mathsf{@Size}(I) \neq 0$ **do**
      $L := \langle \mathsf{last}(I)\rangle \mathbin{+\!\!+} L$; VAR **od**;
    $I := \mathsf{@Make}(\mathtt{Statements}, \langle\rangle, L)$ **end.**

The statement VAR is:

var $\langle I_1 := \mathsf{butlast\_item}(I)\rangle$:
  $I := L[1]$; SLICE; $L[1] := I$; $I := I_1$ **end**;

Introduce a new variable R and maintain the invariant $R = \mathsf{@Cs}(I_1)$ over the **while** loop:

var $\langle L := \langle\rangle\rangle$:
  var $\langle R := \mathsf{@Cs}(I)\rangle$:
    while $\mathsf{@Size}(I) \neq 0$ **do**
      $L := \langle \mathsf{last}(I)\rangle \mathbin{+\!\!+} L$;
      var $\langle I_1 := \mathsf{butlast\_item}(I)\rangle$:
        $I := L[1]$; SLICE; $L[1] := I$;
        $R := \mathsf{BUTLAST}(R)$;
        $I := I_1$ **end od**;
    $I := \mathsf{@Make}(\mathtt{Statements}, \langle\rangle, L)$ **end**

Now we can replace all references to I and I_1 within the **var** statement by references to R:

var $\langle L := \langle\rangle\rangle$:
  var $\langle R := \mathsf{@Cs}(I)\rangle$:
    while $R \neq \langle\rangle$ **do**
      $L := \langle \mathsf{LAST}(R)\rangle \mathbin{+\!\!+} L$;
      var $\langle I_1 := \mathsf{butlast\_item}(I)\rangle$:
        $I := L[1]$; SLICE; $L[1] := I$;
        $R := \mathsf{BUTLAST}(R)$;
        $I := I_1$ **end od**;
    $I := \mathsf{@Make}(\mathtt{Statements}, \langle\rangle, L)$ **end**

Delete redundant assignments to I and remove I_1 to get:

var $\langle L := \langle\rangle\rangle$:
  var $\langle R := \mathsf{@Cs}(I)\rangle$:
    while $R \neq \langle\rangle$ **do**
      $I := \mathsf{LAST}(R)$; SLICE; $L := \langle I\rangle \mathbin{+\!\!+} L$;
      $R := \mathsf{BUTLAST}(R)$ **fi**;

$I := \text{@Make}(\texttt{Statements}, \langle\rangle, L)$ **end**

The loop processes the components of I in reverse order, so we can transform it to a **for** loop:

var $\langle L := \langle\rangle\rangle$:
  for $I \in \text{REVERSE}(\text{@Cs}(I))$ **do**
    SLICE; $L := \langle I \rangle +\!\!+ L$ **od**;
  $I := \text{@Make}(\texttt{Statements}, \langle\rangle, L)$ **end**

This code is a refinement of the specification $\{\text{@ST}(I) = \texttt{Statements}\}$; SLICE.

This may seem to be a lot of heavy machinery for such a simple result, but the point of this derivation is that the transformations used are completely general purpose operations which can be applied to WSL programs of arbitrary complexity. The heuristics which guide the transformation process are equally generic.

6.2 Local Variable

For the **var** statement, we slice the body on the set $x \setminus \langle \text{@V}(v) \rangle$ where v is the internal representation of the local variable. We then construct the result, depending on whether the local variable is in the initial set of needed variables:

var $\langle v := I\hat{\ }1, e_1 := I\hat{\ }2, x_0 := x \rangle$:
  $I := I\hat{\ }3$;
  $x := x \setminus \{\text{@V}(v)\}$;
  SLICE;
  if $\text{@V}(v) \notin x$
    then $e := \text{@Make}(\texttt{Bottom}, \langle\rangle, \langle\rangle)$ **fi**;
  $I := \text{@Make}(\texttt{Var}, \langle\rangle, \langle v, e, I \rangle)$;
  $x := (x \setminus \{@V(v)\}) \cup (\{@V(v)\} \cap x_0) \cup \text{@Used}(e))$ **end**

(Note that: $\text{@Used}(\text{@Make}(\texttt{Bottom}, \langle\rangle, \langle\rangle)) = \varnothing$).

6.3 Guarded

A single `Guarded` item with condition $\mathbf{B}$ and statement sequence $\mathbf{S}$ is equivalent to $\{\mathbf{B}\}$; $\mathbf{S}$, so a refinement of $\{\text{@GT}(I) = \texttt{Guarded}\}$; SLICE is:

var $\langle B := I\hat{\ }1 \rangle$:
  $I := I\hat{\ }2$; SLICE; $x := x \cup \text{@Used}(B)$;
  $I := \text{@Make}(\texttt{Guarded}, \langle\rangle, \langle B, I \rangle)$ **end**

A `Cond` statement with no components is defined to be equivalent to an **abort**, while a `Cond` statement with one component is defined to be equivalent to $\{\mathbf{B}\}$; $\mathbf{S}$. Note that the statement **if B then S fi** is represented as a `Cond` with two components:

$$\langle \texttt{Cond}, \langle\rangle, \langle \texttt{Guarded}, \langle\rangle, \overline{\mathbf{B}}, \overline{\mathbf{S}} \rangle, \langle \texttt{Guarded}, \langle\rangle, \overline{\textbf{true}}, \overline{\textbf{skip}} \rangle\rangle$$

6.4 Cond

A Cond statement with two or more components can be sliced by slicing the first component and the rest of the statement, and then combining the result.

The specification we wish to refine is: $\{\text{@ST}(I) = \texttt{Cond}\}$; SLICE. Using case 11 in Lemma 4.7 we can refine this specification to:

if $\text{@Size}(I) = 0$
  then $x := \varnothing$
  else var $\langle I_1 := I\hat{\ }1, I_2 := \text{butlast\_item}(I), x_0 := x, x_1 := \varnothing \rangle$:
      $I := I_1$;
      SLICE;

$$
\begin{array}{l}
x_1 := x;\ I_1 := I; \\
x := x_0;\ I := I_2; \\
\{\text{@ST}(I) = \texttt{Cond}\};\ \text{SLICE}; \\
I := \text{@Make}(\texttt{Cond}, \langle\rangle, \langle I_1\rangle \mathbin{+\!\!+} \text{@Cs}(I)); \\
x := x_1 \cup x \ \textbf{end fi}
\end{array}
$$

The non-negative integer function @Size(I) is reduced before the copy of the specification, so we can apply the recursive implementation theorem:

$$
\begin{array}{l}
\textbf{proc}\ \text{slice}_2\ \equiv \\
\quad \textbf{if}\ \text{@Size}(I) = 0 \\
\quad\quad \textbf{then}\ x := \varnothing \\
\quad\quad \textbf{else var}\ \langle I_1 := I\hat{\ }1, I_2 := \text{butlast\_item}(I), x_0 := x, x_1 := \varnothing\rangle : \\
\quad\quad\quad\quad I := I_1; \\
\quad\quad\quad\quad \text{SLICE}; \\
\quad\quad\quad\quad x_1 := x;\ I_1 := I; \\
\quad\quad\quad\quad x := x_0;\ I := I_2; \\
\quad\quad\quad\quad \text{slice}_2; \\
\quad\quad\quad\quad I := \text{@Make}(\texttt{Cond}, \langle\rangle, \langle I_1\rangle \mathbin{+\!\!+} \text{@Cs}(I)); \\
\quad\quad\quad\quad x := x_1 \cup x\ \textbf{end fi.}
\end{array}
$$

As before, we need to ensure that the recursive call is outside the local variable structure before introducing recursion. We use two stacks: L_1 to store the value of I_1 and L_2 to store x_1:

$$
\begin{array}{l}
\textbf{proc}\ \text{slice}_2\ \equiv \\
\quad \textbf{if}\ \text{@Size}(I) = 0 \\
\quad\quad \textbf{then}\ x := \varnothing \\
\quad\quad \textbf{else var}\ \langle I_2 := \text{butlast\_item}(I), x_0 := x\rangle : \\
\quad\quad\quad\quad I := I\hat{\ }1; \\
\quad\quad\quad\quad \text{SLICE}; \\
\quad\quad\quad\quad L_2 := \langle x\rangle \mathbin{+\!\!+} L_2; \\
\quad\quad\quad\quad L_1 := \langle I\hat{\ }1\rangle + +L_1 \\
\quad\quad\quad\quad x := x_0;\ I := I_2\ \textbf{end}; \\
\quad\quad\quad \text{slice}_2 \\
\quad\quad\quad I := \text{@Make}(\texttt{Cond}, \langle\rangle, \langle L_1[1]\rangle \mathbin{+\!\!+} \text{@Cs}(I)); \\
\quad\quad\quad x := L_2[1] \cup x; \\
\quad\quad\quad L_2 := L_2[2\,..]; \\
\quad\quad\quad L_1 := L_1[2\,..]\ \textbf{fi.}
\end{array}
$$

Now apply the generic recursion removal transformation from [54] to get:

$$
\begin{array}{l}
\textbf{proc}\ \text{slice}_2\ \equiv \\
\quad \textbf{var}\ \langle L_1 := \langle\rangle, L_2 := \langle\rangle\rangle : \\
\quad\quad \textbf{while}\ \text{@Size}(I) \neq 0\ \textbf{do} \\
\quad\quad\quad \textbf{var}\ \langle I_2 := \text{butlast\_item}(I), x_0 := x\rangle : \\
\quad\quad\quad\quad I := I\hat{\ }1; \\
\quad\quad\quad\quad \text{SLICE}; \\
\quad\quad\quad\quad L_2 := \langle x\rangle \mathbin{+\!\!+} L_2; \\
\quad\quad\quad\quad L_1 := \langle I\hat{\ }1\rangle + +L_1 \\
\quad\quad\quad\quad x := x_0;\ I := I_2\ \textbf{end}; \\
\quad\quad\quad x := \varnothing; \\
\quad\quad\quad \textbf{while}\ L_1 \neq \langle\rangle\ \textbf{do} \\
\quad\quad\quad\quad I := \text{@Make}(\texttt{Cond}, \langle\rangle, \langle L_1[1]\rangle \mathbin{+\!\!+} \text{@Cs}(I)); \\
\quad\quad\quad\quad x := L_2[1] \cup x; \\
\quad\quad\quad\quad L_2 := L_2[2\,..]; \\
\quad\quad\quad\quad L_1 := L_1[2\,..]\ \textbf{od.}
\end{array}
$$

Introduce a new variable $x_1 := \bigcup_{1 \leq i \leq \ell(L_2)} L_2[i]$. The second **while** loop simplifies to: $I :=$ $\mathsf{@Make}(\mathtt{Cond}, \langle\rangle, \mathsf{REVERSE}(L_1))$; $x := x_1$. The first **while** loop processes the components of I in order, so we can transform it to a **for** loop. The procedure simplifies to:

proc slice_2 $\equiv$
  var $\langle L_1 := \langle\rangle, x_1 := \varnothing\rangle$:
    for $I \in \mathsf{@Cs}(I)$ **do**
      var $\langle x_0 := x\rangle$:
        SLICE;
        $L_1 := \langle I\rangle \mathbin{+\!\!+} L_1$;
        $x_1 := x \cup x_1$;
        $x := x_0$ **end od**;
    $I := \mathsf{@Make}(\mathtt{Cond}, \langle\rangle, \mathsf{REVERSE}(L_1))$;
    $x := x_1$ **end.**

6.5 While

The final case to consider is the **while** loop. Suppose we have the loop **while B do S od** : $V \to V$ with required set $X \subseteq V$, and we want to find a statement $\mathbf{S}'$ and set Y such that:

$$X \subseteq Y \quad \text{and} \quad \mathsf{vars}(\mathbf{B}) \subseteq Y \quad \text{and} \quad \mathbf{S}' \;{}_{Y}\!\!\leqslant\!\!\!\!\!\!\;\;{}_{Y}\; \mathbf{S}$$

If we assume that Y contains all the variables in x, then Property 1 and Property 13 of Lemma 4.7 together show that:

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

The problem is that Property 13 of Lemma 4.7 gives no hint as to how the set Y, or the statement $\mathbf{S}'$, should be computed.

Suppose we have the loop **while B do S od** : $V \to V$ with required set $X \subseteq V$, and we want to find a statement $\mathbf{S}'$ and set Y such that:

$$X \subseteq Y \quad \text{and} \quad \mathsf{vars}(\mathbf{B}) \subseteq Y \quad \text{and} \quad \mathbf{S}' \;{}_{Y}\!\!\lessdot_{Y}\; \mathbf{S}$$

Define a sequence of sets X_i and a sequence of statements $\mathbf{S}'_i$ where $X_0 = X \cup \mathsf{vars}(\mathbf{B})$, $\mathbf{S}'_0 = \mathbf{S}$ and:

$$\mathbf{S}'_i \;{}_{X'_i}\!\!\lessdot_{X_i}\; \mathbf{S} \quad \text{and} \quad X_{i+1} = \mathsf{vars}(\mathbf{B}) \cup X_i \cup X'_i$$

The sequence X_i is increasing and bounded above by the finite set V, so the sequence must converge in a finite number of steps. i.e. there is an X_n such that $X_m = X_n$ for $m \geqslant n$. This X_n, and the corresponding $\mathbf{S}'_n$ satisfies the requirement $\mathbf{S}'_n \;{}_{X_n}\!\!\lessdot_{X_n}\; \mathbf{S}$, so by Lemma 4.7 we have:

$$\textbf{while B do S}'_n \textbf{ od} \;{}_{X_n}\!\!\lessdot_{X_n}\; \textbf{while B do S od}$$

The following code repeatedly slices the loop body until the set of variables converges:

var $\langle B := I\hat{\ }1, I_0 := I\hat{\ }2, x_1 := x \cup \mathsf{@Used}(I\hat{\ }1)\rangle$:
  do $I := I_0$;
    $x := x_1$;
    SLICE;
    if $x \subseteq x_1$ **then exit fi**;
    $x_1 := x_1 \cup x$ **od**;
  $I := \mathsf{@Make}(\mathtt{While}, \langle\rangle, \langle B, I\rangle)$;
  $x := x_1$ **end**

The loop is guaranteed to terminate, because the value of x_1 is increasing on each iteration and is bounded above by $x_o \cup \mathsf{vars}(I_0)$ (where x_0 is the initial value of x).

6.6 The Slicing Algorithm

If none of the variables assigned in I are in x, or I is an **abort**, or assignment, then Lemma 4.7 provides a simple refinement for SLICE. So, putting these results together we see that SLICE is refined by:

if $@\mathsf{GT}(I) = \texttt{Statements}$
  then var $\langle L := \langle\rangle\rangle$:
    for $I \in \mathsf{REVERSE}(@\mathsf{Cs}(I))$ **do**
      SLICE; $L := \langle I \rangle +\!\!+ L$ **od**;
    $I := @\mathsf{Make}(\texttt{Statements}, \langle\rangle, L)$ **end**
elsif $@\mathsf{ST}(I) = \texttt{Abort}$
  then $x := \langle\rangle$
elsif $@\mathsf{Assigned}(I) \cap x = \varnothing$
  then $I := @\mathsf{Make}(\texttt{Skip}, \langle\rangle, \langle\rangle)$
elsif $@\mathsf{ST}(I) = \texttt{Assignment} \vee @\mathsf{ST}(I) = \texttt{Spec}$
  then $x := (x \setminus @\mathsf{Assigned}(I)) \cup @\mathsf{Used}(I)$
elsif $@\mathsf{ST}(I) = \texttt{Var}$
  then SLICE
elsif $@\mathsf{ST}(I) = \texttt{Guarded}$
  then SLICE
elsif $@\mathsf{ST}(I) = \texttt{Cond}$
  then SLICE
elsif $@\mathsf{ST}(I) = \texttt{While}$
  then SLICE
  else ERROR(“`Unexpected type:  `”, $@\mathsf{Type\_Name}(@\mathsf{ST}(I))$) **fi**

The cases where I is a **skip** or assertion are handled by the third branch, since these statements assign no variables so $@\mathsf{Assigned}(I) = \varnothing$.

The specification will **abort** if I contains an unexpected type. We have refined this **abort** to instead call an error routine.

Now, every copy of SLICE appears at a position where the number of nodes in I is smaller than the original value. So we can apply the recursive implementation theorem once more to get a complete implementation of the specification:

proc slice() $\equiv$
  if $@\mathsf{ST}(I) = \texttt{Statements}$
    then var $\langle L := \langle\rangle\rangle$:
      for $I \in \mathsf{REVERSE}(@\mathsf{Cs}(I))$ **do**
        slice; $L := \langle I \rangle +\!\!+ L$ **od**;
      $I := @\mathsf{Make}(\texttt{Statements}, \langle\rangle, L)$ **end**
  elsif $@\mathsf{ST}(I) = \texttt{Abort}$
    then $x := \langle\rangle$
  elsif $@\mathsf{Assigned}(I) \cap x = \varnothing$
    then $I := @\mathsf{Make}(\texttt{Skip}, \langle\rangle, \langle\rangle)$
  elsif $@\mathsf{ST}(I) = \texttt{Assignment} \vee @\mathsf{ST}(I) = \texttt{Spec}$
    then $x := (x \setminus @\mathsf{Assigned}(I)) \cup @\mathsf{Used}(I)$
  elsif $@\mathsf{ST}(I) = \texttt{Var}$
    then var $\langle \mathsf{assign} := I\hat{\ }1 \rangle$:
      var $\langle v := @\mathsf{V}(\mathsf{assign}\hat{\ }1), e := @\mathsf{Used}(\mathsf{assign}\hat{\ }2, x_0 := x \rangle$:
        $I := I\hat{\ }2$;
        slice;
        if $v \notin x$
          then assign $:= @\mathsf{Make}(\texttt{Assign}, \langle\rangle, \langle \mathsf{assign}\hat{\ }1, \mathsf{BOTTOM}\rangle)$ **fi**;

$x := (x \setminus \{v\}) \cup (\{v\} \cap x_0) \cup e)$
$I :=$ @Make(Var, $\langle\rangle$, $\langle$assign, $I\rangle$) **end end**
elsif @ST(I) = Cond
then var $\langle x_1 := \varnothing, x_0 := x, G := \langle\rangle\rangle$:
for guard $\in$ @Cs(I) **do**
$I :=$ guard^2; $x := x_0$; slice;
$G := \langle$@Make(Guarded, $\langle\rangle$, $\langle$guard^1, $I\rangle + +G$;
$x_1 := x_1 \cup$ @Used(guard^1) $\cup\ xod$;
$x := x_1$;
$I :=$ @Make(Cond, $\langle\rangle$, REVERSE(G)) **end**
elsif @ST(I) = While
then var $\langle B := I$^1, $I_0 := I$^2, $x_1 := x \cup$ @Used(I^1)$\rangle$:
do $I := I_0$;
$x := x_1$;
slice;
if $x \subseteq x_1$ **then exit fi**;
$x_1 := x_1 \cup x$ **od**;
$I :=$ @Make(While, $\langle\rangle$, $\langle B, I\rangle$);
$x := x_1$ **end**
else ERROR("Unexpected type: ", @Type_Name(@ST(I))) **fi.**

Essentially this code was implemented as the Simple_Slice transformation in the FermaT transformation system. After fixing a couple of typos, the code worked first time: which is only to be expected, since it was derived from the specification by a process of formal transformation.

6.7 Extensions to the Target Language

The target language, i.e. the language in which the programs to be sliced are written, has been purposefully kept small in order to avoid cluttering the derivation with extraneous details. In this section, we will discuss how the derivation process can be extended to deal with various extensions to the target language.

6.7.1 Pointers

Pointers, pointer arithmetic and variable aliasing can be treated as an entirely orthogonal problem to program slicing. The first step is to compute a "points-to" analysis which determines for each pointer reference a set of possible memory areas that the pointer could possibly refer to, and for each variable the set of other variables it could be aliased with. This analysis is necessarily a conservative one since the problems of determining precise aliasing and precise points-to analysis are non-computable. The most conservative approximation is to assume that any pointer can point to any memory area, but there is an extensive field of research on improved analyses, see [44,69, 70] and many others. Mock et al have used dynamic points-to data with to obtain a bound on the best case slice size improvement that can be achieved with improved pointer precision [29]

Dynamic data allocation is simply another example of pointer analysis and can be handled in the same way.

6.7.2 Unstructured Control Flow

See Section 7.1 for a brief description of how unstructured control flow (eg **goto** statements) can be handled.

Dynamic dispatch can be treated as a conditional procedure call to one of the set of possible targets for the dispatch. Determining the set of possible targets is again an orthogonal problem.

Exceptions can be treated as unstructured branches and handled as described in Section 7.1.

The next section deals with extending the slicing algorithm in a different direction: keeping the target language the same, but generalising from static slicing to semantic slicing.

7 Semantic Slicing

Semantic slicing is defined purely in terms of the semantic relation, without any syntactic constraint. We define the formula SSlice, which is simply Slice with the syntactic constraint deleted, as follows:

$$\begin{aligned}\mathsf{SSlice}(I,J,x,y) \ =_{\text{DF}}\ \exists v.\ \big(&\mathsf{Fins}(I,v,x)\ \wedge\ \mathsf{Fins}(J,v,x)\\ &\wedge\ (\mathsf{wp}_{\mathbf{t}}(I) \Rightarrow \mathsf{wpa}_{\mathbf{t}}(y,J,v))\\ &\wedge\ (\mathsf{wp}_{\neq}(I,v,x) \Leftrightarrow \mathsf{wp}_{\mathbf{t}}(I)\ \wedge\ \mathsf{wpa}_{\neq}(y,J,v,x))\\ &\wedge\ (\mathsf{wpa}_{\neq}(y,J,v,x) \Leftrightarrow \mathsf{wp}_{\neq}(J,v,x))\big)\end{aligned}$$

The formal specification of a semantic slicing algorithm can then be defined as:

$$\mathsf{SSLICE}\ =_{\text{DF}}\ \langle I,x\rangle := \langle I',x'\rangle.\mathsf{SSlice}(I,I',x,x')$$

From the definitions of SSLICE and TRANS it is clear that:

$$\Delta \vdash \mathsf{SSLICE}\ \approx\ \mathsf{SSLICE};\ \mathsf{TRANS} \quad \text{and} \quad \Delta \vdash \mathsf{SSLICE}\ \approx\ \mathsf{TRANS};\ \mathsf{SSLICE}$$

This means that in the derivation of a semantic slicing algorithm, any program transformation can be inserted before or after any copy of SSLICE in the program. It is also trivial to prove that:

$$\Delta \vdash \mathsf{SSLICE}\ \leq\ \mathsf{SLICE}$$

which means that any copy of SSLICE can be replaced by SLICE. In particular, we can use the implementation of SLICE to slice primitive statements (assertion, assignment etc.)

7.1 Other Slicing Algorithms

The FermaT transformation system also includes a Syntactic_Slice transformation which handles unstructured code and procedures with parameters.

As discussed in [66], we can apply arbitrary transformation in the process of slicing, provided that the final program satisfies all the conditions of Definition 4.3: in particular, that it is a reduction of the original program. So we can implement a slicing algorithm as the sequence:

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

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

FermaT's solution is to *destructure* the program to an action system in "basic blocks" format. To slice the action system, FermaT computes the Static Single Assignment (SSA) form of the program, and the control dependencies of each basic block using Bilardi and Pingali's optimal algorithms [8,37]. FermaT tracks control and data dependencies to determine which statements can be deleted from the blocks. Tracking data dependencies is trivial when the program is in SSA form. FermaT links each basic block to the corresponding statement in the original program, so it can determine which statements from the original program have been deleted (in effect, this will "undo" the destructuring step).

7.2 Performance

The slicer presented in this paper produces identical results as FermaT's Syntactic_Slice transformation, once the extra **skip** statements have been deleted. (This can be carried out with FermaT's Delete_All_Skips transformation.) the only exception is for code containing explicit **abort** statements. The syntactic slicer does not currently recognise that control cannot pass through an **abort** statement: it is not sufficient simply to delete the control flow path through the **abort** since this may result in an invalid control flow graph for the whole program. Such a graph can only arise for a non-terminating program: which can be completely sliced away.

As with any dataflow-based slicer, the algorithm does not take into account dataflows from unreachable code. For example, slicing **abort**; $y := x$ on the final value of y will give **abort**; $y := x$, when **abort**; **skip** is also valid. However, slicing $y := x$; **abort** produces **skip**; **abort** since the set of required variables is empty once an **abort** has been seen in the right-to-left processing of a statement sequence.

Execution performance of both slicers is similar: the worst case arises for deeply-nested loops and is $O(n^2)$ where n is the depth of **while** loop nesting.

The biggest difference is that Syntactic_Slice can handle a larger subset of WSL: including unstructured code and procedures.

7.3 Practical Applications

The FermaT Workbench and FermaT Migration Engine are two commercial tools which apply the program transformation theory to the analysis and migration of assembler code. The assembler is translated into equivalent WSL code which can be analysed and transformed using the FermaT framework. In the case of the Workbench, the results of this analysis are presented back to the programmer in terms of the original assembler code. For example, control dependencies and dataflows are computed from the WSL translation. These are used to compute program slices which are presented to the programmer by highlighting the original assembler source file or listing.

In the case of the Migration Engine, the transformed WSL code is translated into the target language (usually C or COBOL). The correctness of the migration process depends critically on the transformation theory since all the important migration stages (translating data representations, restructuring, simplifying and raising the abstraction level) are carried out in WSL via proven correctness-preserving transformations.

8 Related Work

8.1 Formal Development Methods

Producing a program (or a large part of it) from a specification in a single step is a difficult task to carry out, to survey and to verify [6]. Moreover, programmers tend to underestimate the complexity of given problems and to overestimate their own mental capacity [36] and this exacerbates the situation further.

A solution in which a program is developed incrementally by *stepwise refinement* was proposed by Wirth [71]. However, the problem still remains that each step is done intuitively and must then be validated to determine whether the changes that have been made preserve the correctness of the program with respect to some specification, yet do not introduce unwanted side effects.

The next logical stage, improving on stepwise refinement, is to only allow provably semantic-preserving changes to the program. Such changes are called *transformations*. There are several distinct advantages to this approach [6]:

- The final program is correct (according to the initial specification) *by construction*.
- Transformations can be described by *semantic rules* and can thus be used for a whole class of problems and situations.

- Due to formality, the whole process of program development can be supported by the computer. A significant part of transformational programming involves the use of a large number of small changes to be made to the code. Performing such changes by hand would introduce clerical errors and the situation would be no better than the original *ad hoc* methods. However, such clerical work is ideally suited to automation, allowing the computer itself to carry out the monotonous part of the work, allowing the programmer to concentrate on the actual design decisions.

Development approaches in which each refinement step is first proposed by the developer and then verified correct (also by the developer but with automated assistance in the form of theorem provers) have had some success [27,28,72] but have also encountered difficulties in scaling to large programs. The scaling problems are such that some authors relegate formal methods to the specification stage of software development [19] for all but the most critical systems. These approaches are therefore of very limited application to reverse engineering, program comprehension or reengineering tasks [73].

In a survey of transformational programming [35] R. Paige wrote:

> Transformational systems may have the power to perform sophisticated program analysis and to generate software at breakneck speed, but to date they are not sound. Lacking from them is a convenient mechanical facility to prove that each transformation preserves semantics. In order to create confidence in the products of transformational systems we need to prove correctness of specifications and transformations.

The FermaT transformation system provides implementations of a large number of transformations which have been proved correct. The system also provides mechanically checkable correctness conditions for all the implemented transformations.

Xingyuan Zhang et al [75] have developed a formalisation of WSL in the type-theoretical proof assistant Coq. This has been used to mechanically verify the correctness of some simple restructuring transformations [76].

8.2 Formal Reverse Engineering Methods

The approach presented here, in which a large catalogue of proven transformations, together with their correctness conditions, is made available via a semi-automatic transformation system, has been proved capable of scaling up to large software developments and has the further advantage of being applicable in the reverse engineering and reengineering realm. Because the transformations are known to be correct, they can be applied "blindly" to an existing program whose function is not clearly understood in order to restructure and simplify the program into a more understandable form. FermaT is described as "semi-automatic" because the user selects each transformation and point of application, while the system checks the applicability conditions and applies the transformation. It should be noted that some of the transformations are actually meta-transformations which use heuristics to control the transformation process in order to apply a large number of other transformations to the whole target program. Many activities are therefore totally automated: including WSL restructuring and the whole migration process from assembler to C or COBOL (see [53,63]).

Note that proving the correctness of an assembler to WSL translator would require a formal specification of assembler language: which is generally not available. Our solution is to develop translators which, as far as possible, translate each instruction separately using a translation table which gives the mapping between each assembler instruction and its WSL implementation. In effect, the translation table provides a (partial) formal specification for the assembler language. The translator does not need to be concerned about introducing redundant or inefficient code (such as setting a flag which is immediately tested, or assigning data to variables which will be overwritten) since these inefficiencies will be removed by automated transformations. Similarly, the WSL to C and COBOL translators are designed to work with WSL code which has been transformed into a form which is already very close to C or COBOL. So the translation step itself is a simple one-to-one mapping of program structures.

The long range goal of transformational programming is to improve reliability, productivity, maintenance and analysis of software without sacrificing performance [35].

8.3 Refinement

The *Refinement Calculus* approach to program derivation [22,31,33] is superficially similar to our program transformation method. It is based on a wide spectrum language, using Morgan's specification statement [30] and Dijkstra's guarded commands [17]. However, this language has very limited programming constructs: lacking loops with multiple exits, action systems with a "terminating" action, and side-effects. These extensions are essential if transformations are to be used for reverse engineering. The most serious limitation is that the transformations for introducing and manipulating loops require that any loops introduced must be accompanied by suitable invariant conditions and variant functions. This makes the method unsuitable for a practical reengineering method. Morgan remarks (pp 166–167 of [31]) that the whole development history is required for understanding the structure of the program and making safe modifications.

The Z specification notation [43] has recently been used to develop the specification and full refinement proof of a large, industrial scale application [45]. The proofs were carried out by hand but the proof obligations and many of the steps were mechanically type-checked. The author's discuss the trade-offs they had to make to find the right path between greater mathematical formality and the need to press on and "just do" the proofs in any way they could.

Bicarregui and Matthews [7] investigated the use of automated theorem proving technology in the refutation of proof obligations arising from this development method. They note that even a simple design using the B-Method and tools may give rise to many thousands of proof obligations. Although automated theorem provers can help, they found that: "when a prover fails to complete a proof, the developer is left uncertain as to whether the failure is due to the inability of the prover to find a proof or simply because the conjecture is actually false, that is, because there is in fact some defect in the design. In this situation, the developer is often left with the highly specialised task of inspecting the failed proof to try to discover which of these is the case."

9 Conclusions

In this paper we have presented a case study in transforming a specification (in the form of a WSL specification statement) into a WSL implementation via correctness preserving transformations. We defined methods for representing finite WSL programs as data, together with definitions of the reduction relation, refinement and semi-refinement as WSL conditions. These were used to define the slicing relation $\not\leqslant$ as a WSL condition.

These definitions allowed us to define a WSL specification for slicing, such that *any* correct slicing algorithm is a refinement of the specification. Using the WSL program derivation method we have derived a simple slicing algorithm by transforming the slicing specification into an implementation. The simple slicing algorithm is essentially the same as the Simple_Slice transformation implemented in the FermaT transformation system.

FermaT also implements a more powerful syntactic slicing algorithm which can handle procedures, **do** ... **od** loops and action systems, and a semantic slicing algorithm which is described elsewhere [67]. FermaT can be downloaded from:

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

This "transformational" method for algorithm derivation has many advantages, including the following:

1. At each step in the derivation process, we are working with a program which is guaranteed to be a correct implementation of the original specification. The final program is therefore also guaranteed to be correct;
2. The early stages of derivation rely on human intuition and understanding. During these stages, the program being manipulated is still in the form of a non-recursive specification. This means that it is easy to take a vague "idea" about how the algorithm might be implemented and use the idea to elaborate the specification into a more detailed specification. For example, the derivation of a binary search algorithm might starts with the simple idea: pick an element in the array and compare it with the value we are searching for.
3. Only in the later stages of derivation do we need to introduce recursion and iteration. At this point we have available several powerful automatic transformations, so there is less need to rely on human intuition. The later stages of the derivation are almost totally automatic.
4. There is rarely any need to introduce loop invariants or loop variant functions. Loop invariants are typically simple, localised invariants which relate two data structures. These are used to replace one data structure by an equivalent one.
5. There are no huge lists of "proof obligations" which need to be discharged before the derivation can be considered correct.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman & M. Wand, "Revised^5 Report on the Algorithmic Language Scheme," Feb., 1998, ⟨http://ftp-swiss.ai.mit.edu/~jaffer/r5rs_toc.html⟩.

[2] Torben Amtoft, "Slicing for Modern Program Structures: a Theory for Eliminating Irrelevant Loops," *Information Processing Letters* 106 (2008), 45–51.

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

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

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

[6] F. L. Bauer, B. Moller, H. Partsch & P. Pepper, "Formal Construction by Transformation—Computer Aided Intuition Guided Programming," *IEEE Trans. Software Eng.* 15 (Feb., 1989).

[7] Juan C. Bicarregui & Brian M. Matthews, "Proof and Refutation in Formal Software Development ," *In 3rd Irish Workshop on Formal Software Development* (July, 1999).

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

[9] D. Binkley, "Precise Executable Interprocedural Slices," *ACM Letters on Programming Languages and Systems* 2 (Mar., 1993), 31–45.

[10] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss & Bogden Korel, "A Formalisation of the Relationship between Forms of Program Slicing," *Science of Computer Programming* 62 (2006), 228–252.

[11] Dave Binkley & Mark Harman, "A Survey of Empirical Results on Program Slicing," in *Advances in Computers*, Marvin Zelkowitz, ed. #62, Academic Press, San Diego, CA, 2004.

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

[13] David W. Binkley & Keith B. Gallagher, "A Survey of Program Slicing," *Advances in Computers* 43 (1996), 1–52.

[14] Richard Bird & Oege de Moor, *The Algebra of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[15] Z. Chen, A. Cau, H. Zedan & H. Yang, "A Refinement Calculus for the Development of Real-Time Systems," *5th Asia Pacific Software Engineering Conference (APSEC98)*, Taipei, Taiwan (1998).

[16] Radhie Cousot, ed., *Verification, Model Checking, and Abstract Interpretation 6th International Conference, VMCAI 2005, Paris, France, January 17–19, 2005*, Lect. Notes in Comp. Sci. #3385, Springer-Verlag, New York–Heidelberg–Berlin, 2005.

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

[18] E. W. Dijkstra, "By way of introduction," EWD 1041, Feb., 1989.

[19] A. Hall, "Seven Myths of Formal Methods," *IEEE Software* 7 (Sept., 1990), 11–19.

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

[21] John Hatcliff, Matthew B. Dwyer & Hongjun Zheng, "Slicing Software for Model Construction," *Journal of Higher-order and Symbolic Computation* 13 (Dec., 2000), 315–353.

[22] C. A. R. Hoare, I. J. Hayes, H. E. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey & B. A. Sufrin, "Laws of Programming," *Comm. ACM* 30 (Aug., 1987), 672–686.

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

[24] C. B. Jones, K. D. Jones, P. A. Lindsay & R. Moore, *mural: A Formal Development Support System*, Springer-Verlag, New York–Heidelberg–Berlin, 1991.

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

[26] D. E. Knuth, *Sorting and Searching*, The Art of Computer Programming #2, Addison Wesley, Reading, MA, 1969.

[27] K. C. Lano & H. P. Haughton, "Formal Development in B," *Information and Software Technology* 37 (June, 1995), 303–316.

[28] Kevin Lano, *The B Language and Method: A Guide to Practical Formal Development*, Springer-Verlag, ISBN 3-540-76033-4, 1996.

[29] Markus Mock, Darren C. Atkinson, Craig Chambers & Susan J. Eggers, "Improving Program Slicing With Dynamic Points-to Data," *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, New York, NY, USA (2002).

[30] C. C. Morgan, "The Specification Statement," *Trans. Programming Lang. and Syst.* 10 (1988), 403–419.

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

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

[33] C. C. Morgan, K. Robinson & Paul Gardiner, "On the Refinement Calculus," Oxford University, Technical Monograph PRG-70, Oct., 1988.

[34] M. Neilson, K. Havelund, K. R. Wagner & E. Saaman, "The RAISE Language, Method and Tools," *Formal Aspects of Computing* 1 (1989), 85–114 .

[35] R. Paige, "Future Directions In Program Transformations," *ACM Computing Surveys* 28A (1996), 170–174.

[36] H. Partsch & R. Steinbrügen, "Program Transformation Systems," *Computing Surveys* 15 (Sept., 1983).

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

[38] H. A. Priestley & M. Ward, "A Multipurpose Backtracking Algorithm," *J. Symb. Comput.* 18 (1994), 1–40, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/backtr-t.ps.gz⟩.

[39] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer & John Hatcliff, "A New Foundation For Control-Dependence and Slicing for Modern Program Structures," *Proceedings of the European Symposium On Programming (ESOP'05), Edinburg, Scotland* 3444 (Apr., 2005), 77–93.

[40] Thomas Reps, "Algebraic properties of program integration," *Science of Computer Programming,* 17 (1991), 139–215.

[41] Thomas Reps & Wuu Yang, "The Semantics of Program Slicing," *Computer Sciences Technical Report* 777 (June, 1988).

[42] Nuno F. Rodrigues & Luís S. Barbosa, "Program slicing by calculation," *Journal of Universal Computer Science* 12 (2006), 828–848.

[43] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall International, Hemel Hempstead, 1992, Second Edition.

[44] Bjarne Steensgaard, "Points-to Analysis in Almost Linear Time," POPL96, 1996.

[45] Susan Stepney, David Cooper & Jim Woodcock, "More Powerful Z Data Refinement: pushing the state of the art in industrial refinement," in *The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, September 1998*, Jonathan P. Bowen, Andreas Fett & Michael G. Hinchey, eds., Lect. Notes in Comp. Sci. #1493, Springer-Verlag, New York–Heidelberg–Berlin, 1998, 284–307.

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

[47] Alan Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society* 2 (1936), 230–265.

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

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

[50] M. Ward, "A Recursion Removal Theorem," Springer-Verlag, Proceedings of the 5th Refinement Workshop, London, 8th–11th January, New York–Heidelberg–Berlin, 1992, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/ref-ws-5.ps.gz⟩.

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

[52] M. Ward, "Program Analysis by Formal Transformation," *Comput. J.* 39 (1996), ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/topsort-t.ps.gz⟩.

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

[54] M. Ward, "Recursion Removal/Introduction by Formal Transformation: An Aid to Program Development and Program Comprehension," *Comput. J.* 42 (1999), 650–673.

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

[56] 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).

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

[58] M. Ward, "Specifications from Source Code—Alchemists' Dream or Practical Reality?," *4th Reengineering Forum, September 19-21, 1994, Victoria, Canada* (Sept., 1994).

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

[60] M. Ward & K. H. Bennett, "A Practical Program Transformation System for Reverse Engineering," in *IEEE Conference on Software Maintenance-1993*, Montreal, Canada, 1993.

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

[62] Martin Ward, "The FermaT Assembler Re-engineering Workbench," *International Conference on Software Maintenance (ICSM), 6th–9th November 2001, Florence, Italy* (2001).

[63] Martin Ward, "Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations," *Science of Computer Programming, Special Issue on Program Transformation* 52 (2004), 213–255, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/migration-t.ps.gz⟩.

[64] Martin Ward, "Properties of Slicing Definitions," *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, Los Alamitos, California, USA (Sept., 2009).

[65] Martin Ward & Hussein Zedan, "MetaWSL and Meta-Transformations in the FermaT Transformation System," *29th Annual International Computer Software and Applications Conference, Edinburgh, UK, November 2005* (2005).

[66] Martin Ward & Hussein Zedan, "Slicing as a Program Transformation," *Trans. Programming Lang. and Syst.* 29 (Apr., 2007), 1–52, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/slicing-t.ps.gz⟩.

[67] Martin Ward, Hussein Zedan, Matthias Ladkau & Stefan Natelberg, "Conditioned Semantic Slicing for Abstraction; Industrial Experiment," *Software Practice and Experience* 38 (Oct., 2008), 1273–1304, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/slicing-paper-final.pdf⟩.

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

[69] Robert P. Wilson, "Efficient Context-Sensitive Pointer Analysis For C Programs," Stanford University, Computer Systems Laboratory, Ph.D. Thesis, Dec., 1997.

[70] Robert P. Wilson & Monica S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs," Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, June, 1995.

[71] N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM* 14 (1971), 221–227.

[72] John Wordsworth, *Software Engineering with B*, Addison Wesley Longman, ISBN 0-201-40356-0., 1996.

[73] H. Yang & M. Ward, *Successful Evolution of Software Systems*, Artech House, Boston, London, 2003, ISBN-10 1-58053-349-3 ISBN-13 978-1580533492.

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

[75] Xingyuan Zhang, Malcolm Munro, Mark Harman & Lin Hu, *Weakest Precondition for General Recursive Programs Formalized in Coq*, Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heidelberg–Berlin, 2002, Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs).

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