Reengineering Software Systems for Flexibility and Reliability

Hongji Yang and Martin Ward

De Montfort University
Leicester, England

Contents

Chapter 4

WSL and Transformation Theory

The previous chapter gave an overview of a number of different formal methods and discussed their application to systems evolution. In this chapter we will focus on one of the most successful formal methods for reengineering sequential systems: the WSL program transformation theory and supporting FermaT workbench.

4.1 Introduction

A computer program is traditionally thought of as a list of detailed instructions, intended to be executed on a machine in order to produce a particular result. For example, the program in Figure 4.1 is intended to set z to the value x^n for non-negative integer values of n. It also sets n to zero. Another

$z := 1;$
while $n > 0$ **do**
    $z := z * x;$
    $n := n - 1$ **od**

Figure 4.1: A simple program

way to think of a computer program is as a description of a function which translates an input state to an output state. If we start the program in Figure 4.1 in a state where x has the value 2 and n has the value 3, then it will run for a while (passing through various intermediate states) and then terminate in a state where z has the value 8 and n has the value 0.

$$z := x^n;\ n := 0$$

Figure 4.2: Another program

Another way of describing the same mathematical function is the program in Figure 4.2. In this case, there is only one intermediate state.

A *specification* is also a description of a function, but in this case it does not have to be an executable program. For example, a program which sets x to a value which when squared equals 4 might be described as:

$$x := x'.(x'^2 = 4)$$

Informally this specification says "assign a new value x' to x so that the condition $x'^2 = 4$ is satisfied." (The prime on x' allows us to describe a relationship between the *old* value of x and the new value x' which is about to be assigned to x).

In this case, there are two possible cases for the final value of x: $+2$ and -2. The specification does not specify which value is required, so we can assume that an implementor of the specification is allowed to choose whichever value is most convenient. To capture this range of implementation choices, the function we are describing must map an initial state to a *set* of possible final states.

A possible implementation of our specification is the simple assignment $x := 2$. This is a *refinement* of the original specification because the set of possible final states for the implementation is a *subset* of the final states for the program.

If a program $\mathbf{S}_1$ is a refined by another program $\mathbf{S}_2$ then we write $\mathbf{S}_1 \leq \mathbf{S}_2$. If also $\mathbf{S}_2 \leq \mathbf{S}_1$ then we say that the two programs are *equivalent* and write $\mathbf{S}_1 \approx \mathbf{S}_2$. In this case, the functions described by the two programs are identical: even if the programs themselves may look completely different.

If our specifications are written in a formally defined mathematical language, then it is possible to *prove* that a given program is a correct implementation of a given specification. For most programs however we want to break down this proof into a number of steps with a number of intermediate stages between specification and program. The easiest way to do this is to include specifications as part of our programming language: then all the intermediate stages can be written in the same language and all the proof steps can be carried out in that language. If our language also includes low-level programming constructs then it is called a *wide spectrum language* since it covers the whole spectrum from abstract mathematical specifications to executable implementations.

A *program transformation* is an operation which can be applied to a program to generate another equivalent program (provided any given applicability conditions are satisfied). This uses a wide spectrum language called WSL for which a powerful set of transformations can be used for refining specifications into programs, reverse engineering programs into specifications and analysing the properties of programs.

4.2 Background

The following requirements went into the development of the WSL language and transformation theory:

1. General specifications in any "sufficiently precise" notation should be included in the language. For sufficiently precise we will mean anything which can be expressed in terms of mathematical logic with suitable notation. This will allow a wide range of forms of specification, for example **Z** specifications [9] and **VDM** [11] both use the language of mathematical logic and set theory (in different notations) to define specifications;

2. Nondeterministic programs. Since we do not want to have to specify everything about the program we are working with (certainly not in the first versions) we need some way of specifying that some executions will not necessarily result in a particular outcome but one of an allowed range of outcomes. The implementor can then use this latitude to provide a more efficient implementation which still satisfies the specification;

3. A well-developed catalogue of proven transformations which do not require the user to discharge complex proof obligations before they can be applied. In particular, it should be possible to introduce, analyse and reason about imperative and recursive constructs without requiring loop invariants;

4. Techniques are needed to bridge the "abstraction gap" between specifications and programs;

5. Applicable to real programs—not just those in a "toy" programming language with few constructs. This is achieved by the (programming) language independence and extendibility of the notation via "definitional transformations";

6. Scalable to large programs: this implies a language which is expressive enough to allow automatic translation from existing programming languages, together with the ability to cope with unstructured programs and a high degree of complexity.

The FermaT transformation system which is built on the transformation theory has applications in the following areas:

- Improving the maintainability (in particular, flexibility and reliability, and hence extending the lifetime) of existing mission-critical software systems;

- Translating programs to modern programming languages, for example from obsolete Assembler languages to modern high-level languages;

- Developing and maintaining safety-critical applications. Such systems can be developed by transforming high-level specifications down to efficient low level code with a very high degree of confidence that the code correctly implements every part of the specification. When enhancements or modifications are required, these can be carried out on the appropriate specification, followed by "re-running" as much of the formal development as possible. Alternatively, the changes could be made at a lower level, with formal inverse engineering used to determine the impact on the formal specification;

- Extracting reusable components from current systems, deriving their specifications and storing the specification, implementation and development strategy in a repository for subsequent reuse;

- Reverse engineering from existing systems to high-level specifications, followed by subsequent re-engineering and evolutionary development.

The WSL language is built up in a series of stages or levels, starting with a very small and mathematically tractable "kernel language".

The next two sections develop the theory of how to prove the correctness of a program transformation. It is not necessary for the user to understand this theory in order to *use* program transformations in a reverse engineering or re-engineering project. Program transformation users who are not interested in the theory are encouraged to skip to Section 4.5.

4.3 Syntax and Semantics of the Kernel Language

4.3.1 Syntax

Our kernel language consists of four primitive statements, two of which contain formulae of infinitary first order logic, and three compound statements. Let $\mathbf{P}$ and $\mathbf{Q}$ be any formulae, and $\mathbf{x}$ and $\mathbf{y}$ be any non-empty lists of variables. The following are primitive statements:

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

3. **Add variables:** $\mathsf{add}(\mathbf{x})$ adds the variables in $\mathbf{x}$ to the state space (if they are not already present) and assigns arbitrary values to them;

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

There is a rather pleasing duality between the assertion and guard statements, and the add and remove statements.

For any kernel language statements $\mathbf{S}_1$ and $\mathbf{S}_2$, the following are also kernel language statements:

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

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

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

At first sight, this kernel language may seem to be missing some essential programming constructs such as assignment statements and **if** statements. But the guard statement can be composed with a nondeterministic statement to get a deterministic result. For example, an assignment such as $x := 1$ is constructed by giving x an arbitrary value and then restricting its value to the one required: $\mathsf{add}(\langle x \rangle);\ [x = 1]$. For an assignment such as $x := x + 1$, where the new value of x depends on the old value, we need to record the required new value of x in a new variable, x' say, before copying it into x. So we can construct $x := x + 1$ as follows:

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

An **if** statement such as **if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi** is constructed from a nondeterministic choice with guards to make the choice deterministic:

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

For Dijkstra's guarded commands [7] such as: **if** $\mathbf{B}_1\ \rightarrow\ \mathbf{S}_1\ \Box\ \mathbf{B}_2\ \rightarrow\ \mathbf{S}_2$ **fi** we need to ensure that the command will abort in the case where none of the guard conditions are true:

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

Three fundamental statements can be defined immediately:

$$\textbf{abort}\ =_{\mathrm{DF}}\ \{\textbf{false}\} \qquad \textbf{null}\ =_{\mathrm{DF}}\ [\textbf{false}] \qquad \textbf{skip}\ =_{\mathrm{DF}}\ \{\textbf{true}\}$$

where **true** and **false** are universally true and universally false formulae. The **abort** statement never terminates when started in any initial state. **skip** is a statement which always terminates immediately in the same state in which it was started. **null** is a rather unusual statement: it always terminates but the set of final states is empty. This statement is a "correct refinement" of *any* specification whatsoever. Morgan [15] uses the term "miracle" for such statements. Clearly, any null statement, and guard statements in general cannot be directly implemented: if a program terminates, then it must terminate in some state or other, and a program cannot in general force a condition to be true without changing the value of a variable.

Null statements are nonetheless a useful theoretical tool, but as it is only null-free statements which are implementable, it is important to be able to distinguish easily which statements are null-free. This is the motivation for the definition of our specification statement in the next section.

The kernel language statements have been described as "The quarks of programming" — mysterious objects which (in the case of the guard at least) are not implementable in isolation, but which in combination, form the familiar "atomic" operations of assignment, **if** statements etc.

4.3.2 The Specification Statement

A specification describes *what* a program should do while abstracting away the implementation details of *how* the result is to be achieved. In mathematical terms, a specification is a description of the relationship between input and output states of the program which does not necessarily describe how this relationship is to be achieved. Suppose we have a list $\mathbf{x}$ of variables which are the outputs of the program and suppose that the formula $\mathbf{Q}$ describes the relationship between the new values $\mathbf{x}'$ which we wish to assign to $\mathbf{x}$, and the original values. This specification is described by the statement

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q}$$

This statement 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 values $\mathbf{x}'$ which satisfy $\mathbf{Q}$ then the statement aborts. The formal definition of this specification statement is:

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

The initial assertion ensures that this statement is null-free.

As an example, we can specify a program to sort the array A using a single specification statement:

$$A := A'.(\mathsf{sorted}(A') \ \wedge \ \mathsf{permutation\_of}(A', A))$$

This says "assign a new value A' to A which is a sorted array and a permutation of the original value of A", it precisely describes *what* we want our sorting program to do without saying *how* it is to be achieved. In other words, it is not biased towards a particular sorting algorithm. In [22] we take this specification as our starting point for the "derivation by formal transformation" of several efficient sorting algorithms, including insertion sort, quicksort and a hybrid sort.

The simple assignment $v := e$, where v is a variable and e is an expression, is defined as the specification statement $\langle v \rangle := \langle v' \rangle.(v' = e)$.

Morgan and others [14,15,16,17] use a different specification statement, written

$$\mathbf{x}\colon [\mathsf{Pre}, \mathsf{Post}]$$

where $\mathbf{x}$ is a sequence of variables and Pre and Post are formulae if *finitary* first-order logic. This statement is guaranteed to terminate for all initial states which satisfy Pre and will terminate in a state which satisfies Post, while only assigning to variables in the list $\mathbf{x}$. In our notation an equivalent

statement is {Pre}; add(x); [Post]. The disadvantage of this notation is that it is not necessarily null-free (the statement $\langle\rangle : [\mathbf{true}, \mathbf{false}]$, for example, is equivalent to **null**). As a result, the user is responsible for ensuring that he never accidentally refines a specification into an (unimplementable) null statement.

4.3.3 States and State Transformations

The functions which are defined by WSL programs are called *state transformations*. These functions map an initial state to a set of possible final states, with a special "state", denoted $\perp$, to include the possibility that the program may never terminate. A state s other than $\perp$ is a partial function which gives the values of all the variables in the *state space* (the set of variables on which the program operates).

For example, the state transformation for the assignment $i := i + 1$ maps each initial state s to a singleton set $\{s'\}$ of final states, where s' gives the value $s(i) + 1$ to the variable i and for all other variables, $s'(x) = s(x)$.

Despite the large amount of research and development on "stateless" functional programming, the vast majority of programs in the world are written in imperative languages so for a reverse engineering technology it is important (if not imperative...) that we can cope easily with imperative programs.

4.3.4 Refinement of State Transformations

When we say that one program is a refinement of another, we mean that any specification which the program satisfies is guaranteed to be satisfied by the program's refinement. In other words, the refinement is "at least as good" at satisfying specifications as the original program. A specification of a program can be defined by giving a set of states (those initial states for which the program's behaviour is to be specified) called the *defined set* and for each of these initial states, a set of allowed final states. A program satisfies the specification if, for each initial state in the defined set, the program is guaranteed to terminate in one of the allowed final states. A specification can therefore be given in the form of a state transformation f where $f(s)$ contains $\perp$ if s is not in the defined set of states, and for every other s, $f(s)$ is the set of allowed final states. Conversely, any state transformation also defines a specification.

We can therefore define "satisfaction of a specification" as a relation between state transformations. We can also define a *refinement* of a state transformation to be a state transformation which satisfies all the specifications satisfied by the first state transformation.

With these definitions it turns out that refinement and satisfaction are identical concepts: a state transformation f_2 is a refinement of state transformation f_1 if and only if it satisfies f_1 (considered as a specification).

4.3.5 Recursion

A program containing calls to a procedure whose definition is not provided can be thought of as a function from state transformations to state transformations; since the "incomplete" program can be completed by filling in the body of the procedure. For a recursive procedure call, we "fill in" the procedure body with copies of itself: but this means that the result of the "fill in" is still incomplete since it will still contain recursive calls. However, the expanded program is "nearer" to completion in some sense which we will make precise. A recursive procedure can be considered as the "limit" formed by joining together the results of infinite sequence of such filling-in operations. More formally:

Definition 4.3.1 *Recursion:* Suppose we have a function $\mathcal{F}$ which maps the set of state transformations $F_{\mathcal{H}}(V, V)$ to itself. We want to define a recursive state transformation from $\mathcal{F}$ as the limit of the sequence of state transformations $\mathcal{F}(\Omega)$, $\mathcal{F}(\mathcal{F}(\Omega))$, $\mathcal{F}(\mathcal{F}(\mathcal{F}(\Omega)))$, ... With the definition of state transformation given above, this limit $(\mu.\mathcal{F})$ has a particularly simple and elegant definition:

$$(\mu.\mathcal{F}) =_{\text{DF}} \bigsqcup_{n<\omega} \mathcal{F}^n(\Omega) \qquad \text{i.e., for each } s \in V_{\mathcal{H}}\text{:} \quad (\mu.\mathcal{F})(s) = \bigcap_{n<\omega} \mathcal{F}^n(\Omega)(s)$$

From this definition we see that $\mathcal{F}((\mu.\mathcal{F})) = (\mu.\mathcal{F})$. So the state transformation $(\mu.\mathcal{F})$ is a fixed point for the function $\mathcal{F}$; it is, in fact, the *least* fixed point.

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

4.3.6 Weakest Preconditions

We define the weakest precondition, $\text{wp}(f, e)$ of a state transformation f and a condition on the final state e to be the weakest condition on the initial state space such that if s satisfies this condition then all elements of $f(s)$ satisfy e. A condition on states is simply a set of states: the set of states

which satisfy the condition. The special state $\perp$ is defined as not satisfying any condition. So $\mathrm{wp}(f, e)$ is simply the set of proper initial states s such that $f(s)$ is in e.

The importance of weakest preconditions is shown by the fact that the refinement relation can be characterised using weakest preconditions. State transformation f_1 is refined by f_2 if and only if for every final state e we have $\mathrm{wp}(f_1, e) \subseteq \mathrm{wp}(f_2, e)$.

This characterisation of refinement still requires us to examine every possible final state in order to determine if one state transformation is a refinement of another. A theorem in [25] shows that it is only necessary to examine two special postconditions: the condition **true** and the condition $\mathbf{x} \neq \mathbf{x}'$, where $\mathbf{x}$ is a list of all the variables used in the program and $\mathbf{x}'$ is a list of variables not used anywhere in the program (and the length of the two lists is the same).

The fact that refinement can be defined directly from the weakest precondition will later prove to be vitally important.

4.3.7 Weakest Preconditions of Statements

We can also define a weakest precondition for kernel language statements as a formula of infinitary logic. Infinitary logics are an extension of first order logic which allows conjunction and disjunction over infinite lists of formulae. See [12,19] for a general introduction to infinitary logics. These were first used to define the semantics of programs by Engeler [8] and are used to express weakest preconditions by Back [3].

WP is a function which takes a statement (a syntactic object) and a formula from our infinitary logic $\mathcal{L}$ (another syntactic object) and returns another formula in $\mathcal{L}$.

Definition 4.3.2 *For any kernel language statement* $\mathbf{S}\colon V \to W$*, and formula* $\mathbf{R}$ *whose free variables are all in* W*, we define* $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ *as follows:*

1. $\mathrm{WP}(\{\mathbf{P}\}, \mathbf{R}) =_{\mathrm{DF}} \mathbf{P} \wedge \mathbf{R}$
2. $\mathrm{WP}([\mathbf{Q}], \mathbf{R}) =_{\mathrm{DF}} \mathbf{Q} \Rightarrow \mathbf{R}$
3. $\mathrm{WP}(\mathsf{add}(\mathbf{x}), \mathbf{R}) =_{\mathrm{DF}} \forall \mathbf{x}.\, \mathbf{R}$
4. $\mathrm{WP}(\mathsf{remove}(\mathbf{x}), \mathbf{R}) =_{\mathrm{DF}} \mathbf{R}$
5. $\mathrm{WP}((\mathbf{S}_1;\ \mathbf{S}_2), \mathbf{R}) =_{\mathrm{DF}} \mathrm{WP}(\mathbf{S}_1, \mathrm{WP}(\mathbf{S}_2, \mathbf{R}))$
6. $\mathrm{WP}((\mathbf{S}_1 \sqcap \mathbf{S}_2), \mathbf{R}) =_{\mathrm{DF}} \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \wedge \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$

7. $\mathrm{WP}((\mu X.\mathbf{S}), \mathbf{R}) =_{\mathrm{DF}} \bigvee_{n<\omega} \mathrm{WP}((\mu X.\mathbf{S})^n, \mathbf{R})$

where $(\mu X.\mathbf{S})^0 = \textbf{abort}$ and $(\mu X.\mathbf{S})^{n+1} = \mathbf{S}[(\mu X.\mathbf{S})^n/X]$ which is $\mathbf{S}$ with all occurrences of X replaced by $(\mu X.\mathbf{S})^n$.

For the fundamental statements we have:

$$\mathrm{WP}(\textbf{abort}, \mathbf{R}) = \textbf{false}$$
$$\mathrm{WP}(\textbf{skip}, \mathbf{R}) = \mathbf{R}$$
$$\mathrm{WP}(\textbf{null}, \mathbf{R}) = \textbf{true}$$

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

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

For Morgan's specification statement $\mathbf{x}\colon [\mathsf{Pre}, \mathsf{Post}]$ we have:

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

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

For the **if** statement discussed in Section 4.3.1:

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

Similarly, for the Dijkstra guarded command:

$$\mathrm{WP}(\textbf{if } \mathbf{B}_1 \to \mathbf{S}_1 \,\square\, \mathbf{B}_2 \to \mathbf{S}_2 \textbf{ fi}, \mathbf{R}) \iff (\mathbf{B}_1 \vee \mathbf{B}_2) \wedge (\mathbf{B}_1 \Rightarrow \mathrm{WP}(\mathbf{S}_1, \mathbf{R})) \wedge (\mathbf{B}_2 \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R}))$$

The weakest precondition "captures" the semantics of a program in the sense that, for any two programs $\mathbf{S}_1\colon V \to W$ and $\mathbf{S}_2\colon V \to W$, the statement $\mathbf{S}_2$ is a correct refinement of $\mathbf{S}_1$ if and only if the formula

$$\big(\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}')\big) \wedge \big(\mathrm{WP}(\mathbf{S}_1, \textbf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \textbf{true})\big)$$

is a theorem of first order logic, where $\mathbf{x}$ is a list of all variables assigned to by either $\mathbf{S}_1$ or $\mathbf{S}_2$, and $\mathbf{x}'$ is a list of new variable. This means that proving a refinement or implementation or equivalence amounts to proving a theorem of first order logic. Back [3,4] and Morgan [15,16] both use weakest preconditions in this way, but Back has to extend the logic with a new predicate symbol to represent the postcondition, and Morgan has to use second order logic with quantification over formulae.

4.4 Proving the Correctness of a Refinement

We can define refinement between statements as the refinements of their interpretations under some structure. This is called *semantic refinement*:

Definition 4.4.1 *Semantic Refinement of statements:* If $\mathbf{S}, \mathbf{S}' \colon V \to W$ have no free statement variables and $\mathrm{int}_M(\mathbf{S}, V) \leq \mathrm{int}_M(\mathbf{S}', V)$ for a structure M of $\mathcal{L}$ then we say that $\mathbf{S}$ is refined by $\mathbf{S}'$ under M and write $\mathbf{S} \leq_M \mathbf{S}'$. If Δ is a set of sentences in $\mathcal{L}$ (formulae with no free variables) and $\mathbf{S} \leq_M \mathbf{S}'$ is true for every structure M in which each sentence in Δ is true then we write $\Delta \models \mathbf{S} \leq \mathbf{S}'$. A structure in which every element of a set Δ of sentences is true is called a *model* for Δ.

It is also useful to be able to prove the correctness of a refinement of statements directly from their weakest preconditions, without first having to calculate the corresponding state transformations. From the last chapter we know that refinement can be characterised by two special weakest preconditions. This is the motivation for the proof-theoretic definition of statement refinement which uses the weakest precondition WP:

Definition 4.4.2 *Proof-Theoretic Refinement:* If $\mathbf{S}, \mathbf{S}' \colon V \to W$ have no free statement variables and $\mathbf{x}$ is a sequence of all variables assigned to in either $\mathbf{S}$ or $\mathbf{S}'$, and the formulae $\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathrm{WP}(\mathbf{S}', \mathbf{x} \neq \mathbf{x}')$ and $\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathrm{WP}(\mathbf{S}', \mathbf{x} \neq \mathbf{x}')$ are provable from the set Δ of sentences, then we say that $\mathbf{S}$ is refined by $\mathbf{S}'$ and write: $\Delta \vdash \mathbf{S} \leq \mathbf{S}'$.

The next theorem shows that, for countable sets Δ, these two notions of refinement are equivalent:

Theorem 4.4.3 *If $\mathbf{S}, \mathbf{S}' \colon V \to W$ have no free statement variables and Δ is any countable set of sentences of $\mathcal{L}$ then:*

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

This theorem provides two different methods for proving a refinement. More importantly though, it proves the connection between the intuitive model of a program as something which starts in one state and terminates (if at all) in some other state, and the weakest preconditions $\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}')$ and $\mathrm{WP}(\mathbf{S}, \mathsf{true})$. For a nondeterministic program there may be several possible final states for each initial state. This idea is precisely captured by the state transformation model of programs and refinement. In the "predicate transformer" model of programs, which forms the foundation for [15] and others, the *meaning* of a program $\mathbf{S}$ is defined to be a function which maps a

postcondition $\mathbf{R}$ to the weakest precondition $\mathrm{WP}(\mathbf{S}, \mathbf{R})$. This model certainly does *not* "correspond closely with the way that computers operate" ([15], P.180), although it does have the advantage that weakest preconditions are generally easier to reason about than state transformations. So a theorem which proves the equivalence of the two models allows us to prove refinements using weakest preconditions, while doing justice to the more intuitive model.

The theorem also illustrates the importance of using the infinitary logic $\mathcal{L}_{\omega_1\omega}$ rather than a higher-order logic, or indeed a larger infinitary logic. Back and von Wright [5] describe an implementation of the refinement calculus, based on (finitary) higher-order logic using the refinement rule $\forall \mathbf{R}.\, \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ where the quantification is over all predicates (boolean state functions). However, the completeness theorem fails for all higher-order logics. Karp [12] proved that the completeness theorem holds for $\mathcal{L}_{\omega_1\omega}$ and fails for all infinitary logics larger than $\mathcal{L}_{\omega_1\omega}$. Finitary logic is not sufficient since it is difficult to determine a finite formula giving the weakest precondition for an arbitrary recursive or iterative statement. Using $\mathcal{L}_{\omega_1\omega}$ (the smallest infinitary logic) we simply form the infinite disjunction of the weakest preconditions of all finite truncations of the recursion or iteration. We avoid the need for quantification over formulae because, with our proof-theoretic refinement method, the two postconditions $\mathbf{x} \neq \mathbf{x}'$ and **true** are sufficient. Thus we can be confident that the proof method is both consistent and complete, in the sense that:

1. If the weakest precondition formula can be proved, for statement $\mathbf{S}_1$ and $\mathbf{S}_2$, then $\mathbf{S}_2$ is certainly a refinement of $\mathbf{S}_1$, and

2. If $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ then there certainly exists a proof the corresponding WP formula.

Basing our transformation theory on any other logic would not provide the two different proof methods we require.

Definition 4.4.4 *Statement Equivalence:* If $\Delta \vdash \mathbf{S} \leq \mathbf{S}'$ and $\Delta \vdash \mathbf{S}' \leq \mathbf{S}$ then we say that statements $\mathbf{S}$ and $\mathbf{S}'$ are *equivalent* and write: $\Delta \vdash \mathbf{S} \approx \mathbf{S}'$. Similarly, if $\Delta \models \mathbf{S} \leq \mathbf{S}'$ and $\Delta \vdash \mathbf{S} \leq \mathbf{S}'$ then we write $\Delta \vdash \mathbf{S} \approx \mathbf{S}'$. From Theorem 4.4.3 we have: $\Delta \models \mathbf{S} \approx \mathbf{S}'$ iff $\Delta \vdash \mathbf{S} \approx \mathbf{S}'$.

4.4.1 Expressing a Statement as a Specification

The formulae $\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}')$ and $\mathrm{WP}(\mathbf{S}, \mathbf{true})$ tell us everything we need to know about $\mathbf{S}$ in order to determine whether a given statement is equivalent to it. In fact, as the next theorem shows, if we also know $\mathrm{WP}(\mathbf{S}, \mathbf{false})$

(which is always **false** for null-free programs) then we can construct a specification statement equivalent to **S**. Although this would seem to solve all reverse engineering problems at a stroke, and therefore be a great aid to software maintenance and reengineering, the theorem has fairly limited value for practical programs: especially those which contain loops or recursion. This is partly because there are many different possible representations of the specification of a program, only some of which are useful for software maintenance. In particular the maintainer wants a short, high-level, abstract version of the program, rather than a mechanical translation into an equivalent specification (see [27] for a discussion on defining different levels of abstraction). In practice, a number of techniques are needed including a combination of automatic processes and human guidance to form a practical program analysis system. An example of such a system is the FermaT system [6,29,30] which uses transformations developed from the theoretical foundations presented here.

The theorem is of considerable theoretical value however in showing the power of the specification statement: in particular it tells us that the specification statement is certainly sufficiently expressive for writing the specification of *any* computer program whatsoever. Secondly, we will use the theorem in Chapter three to add a **join** construct to the language and derive its weakest precondition. This means that we can use **join** to write programs and specifications, without needing to extend the kernel language. Thirdly, we use it in Chapter three to add arbitrary (countable) join and choice operators to the language, again without needing to extend the kernel language.

Theorem 4.4.5 *The Representation Theorem: Let* $\mathbf{S}\colon V \to V$*, be any kernel language statement and let* $\mathbf{x}$ *be a list of all the variables assigned to by* $\mathbf{S}$*. Then for any countable set* Δ *of sentences:*

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

4.4.2 Some Basic Transformations

In this section we prove some fundamental transformations of recursive programs. The general induction rule shows how the truncations of a recursion capture the semantics of the full recursion—each truncation contains some information about the recursion, and the set of all truncations is sufficient for proving refinement and equivalence. This induction rule proves to be an essential tool in the development of a transformation catalogue: we will use it almost immediately in the proof of a fold/unfold transformation (Lemma 4.4.9).

Lemma 4.4.6 *The Induction Rule for Recursion:* If Δ is any countable set of sentences and the statements $\mathbf{S}, \mathbf{S}' \colon V \to V$ have the same initial and final state spaces, then:

(i) $\Delta \vdash (\mu X.\mathbf{S})^k \ \leq \ (\mu X.\mathbf{S})$ for every $k < \omega$;

(ii) If $\Delta \vdash (\mu X.\mathbf{S})^n \ \leq \ \mathbf{S}'$ for all $n < \omega$ then $\Delta \vdash (\mu X.\mathbf{S}) \ \leq \ \mathbf{S}'$.

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. This is easily proved by our usual induction on the structure of statements. The induction steps use the following Lemma:

Lemma 4.4.7 *Replacement:* if $\Delta \vdash \mathbf{S}_1 \ \leq \ \mathbf{S}'_1$ and $\Delta \vdash \mathbf{S}_2 \ \leq \ \mathbf{S}'_2$ then:

1. $\Delta \vdash (\mathbf{S}_1; \ \mathbf{S}_2) \ \leq \ (\mathbf{S}'_1; \ \mathbf{S}'_2)$;
2. $\Delta \vdash (\mathbf{S}_1 \ \sqcap \ \mathbf{S}_2) \ \leq \ (\mathbf{S}'_1 \ \sqcap \ \mathbf{S}'_2)$;
3. $\Delta \vdash (\mu X.\mathbf{S}_1) \ \leq \ (\mu X.\mathbf{S}'_1)$.

Proof: Cases (1) and (2) follow by considering the corresponding weakest preconditions. For case (3) use the induction hypothesis to show that for all $n < \omega$: $(\mu X.\mathbf{S}_1)^n \ \leq \ (\mu X.\mathbf{S}'_1)^n$ (since $(\mu X.\mathbf{S}_1)^n$ has a lower depth of recursion nesting than $(\mu X.\mathbf{S}_1)$) and then apply the induction rule for recursion. ■

We can use these lemmas to prove a much more useful induction rule which is not limited to a single recursive procedure, but can be used on statements containing one or more recursive components. For any statement $\mathbf{S}$, define $\mathbf{S}^n$ to be $\mathbf{S}$ with each recursive statement replaced by its nth truncation.

Lemma 4.4.8 *The General Induction Rule for Recursion:* If $\mathbf{S}$ is any statement with bounded nondeterminacy, and $\mathbf{S}'$ is another statement such that $\Delta \vdash \mathbf{S}^n \ \leq \ \mathbf{S}'$ for all $n < \omega$, then $\Delta \vdash \mathbf{S} \ \leq \ \mathbf{S}'$.

The next lemma uses the general induction rule to prove a transformation for folding (and unfolding) a recursive procedure by replacing all occurrences of the call by copies of the procedure. In [21] we generalise this transformation to a "partial unfolding" where selected recursive calls may be conditionally unfolded or replaced by a copy of the procedure body.

Lemma 4.4.9 *Fold/Unfold:* For any $\mathbf{S} \colon V \to V$:

$$\Delta \vdash (\mu X.\mathbf{S}) \ \approx \ \mathbf{S}[(\mu X.\mathbf{S})/X]$$

4.4.3 Proof Rules for Implementations

In this subsection we will develop two general proof rules. The first is for proving the correctness of an potential implementation **S**, of a specification expressed in the form $\{\mathbf{P}\};\ \mathbf{x} := \mathbf{x}'.\mathbf{Q}$. The second is for proving that a given recursive procedure statement is a correct implementation of a given statement. This latter rule is very important in the process of transforming a specification, probably expressed using recursion, into a recursive procedure which implements that specification. In [21,23,24,26] techniques are presented for transforming recursive procedures into various iterative forms. This theorem is also useful in deriving *iterative* implementations of specifications, since very often the most convenient derivation is via a recursive formulation.

Implementation of Specifications

The first proof rule is based on a proof rule in Back [3], we have extended this to include recursion and guard statements. This proof rule provides a means of proving that a statement **S** is a correct implementation of a specification $\{\mathbf{P}\};\ \mathbf{x} := \mathbf{x}'.\mathbf{Q}$. Any **Z** specification, for example, can be cast into this form.

Theorem 4.4.10 Let Δ be a countable set of sentences of $\mathcal{L}$. Let V be a finite nonempty set of variables and $\mathbf{S}\colon V \to W$ a statement. Let $\mathbf{y}$ be a list of all the variables in $V - \tilde{\mathbf{x}}$ which are "assigned to" somewhere in **S**. Let $\mathbf{x}_0$, $\mathbf{y}_0$ be lists of distinct variables not in **S** or V with $\ell(\mathbf{x}_0) = \ell(\mathbf{x})$ and $\ell(\mathbf{y}_0) = \ell(\mathbf{y})$.

$$\begin{array}{rl} \text{If} & \Delta \vdash (\mathbf{P} \wedge \mathbf{x} = \mathbf{x}_0 \wedge \mathbf{y} = \mathbf{y}_0) \Rightarrow \mathrm{WP}(\mathbf{S}, \mathbf{Q}[\mathbf{x}_0/\mathbf{x}, \mathbf{x}/\mathbf{x}'] \wedge \mathbf{y} = \mathbf{y}_0) \\ \text{then} & \Delta \vdash \{\mathbf{P}\};\ \mathbf{x} := \mathbf{x}'.\mathbf{Q} \ \leq\ \mathbf{S} \end{array}$$

The premise states that if $\mathbf{x}_0$ and $\mathbf{y}_0$ contain the initial values of $\mathbf{x}$ and $\mathbf{y}$ then **S** preserves the value of $\mathbf{y}$ and sets $\mathbf{x}$ to a value $\mathbf{x}'$ such that the relationship between the initial value of $\mathbf{x}$ and $\mathbf{x}'$ satisfies **Q**.

This theorem is really only useful for simple implementations of a single specification statement. More complex specifications will be implemented as recursive or iterative procedures: in either case we can use the following theorem to develop a recursive implementation as the first stage. This can be transformed into an iterative program (if required) using the techniques on recursion removal in [21,23,24,26].

Recursive Implementation of General Statements

In this section we prove an important theorem on the recursive implementation of statements. We use it to develop a method for transforming a general specification into an equivalent recursive statement. These transformations 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. The theorem is used in the algorithm derivations of [22,28] and [21].

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 $\mathbf{S}[\mathbf{S}'/X]$ (which denotes $\mathbf{S}$ with all occurrences of X replaced by $\mathbf{S}'$). In other words, if we replace recursive calls in $\mathbf{S}$ by copies of $\mathbf{S}'$ then we get a refinement of $\mathbf{S}'$;

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 $\preccurlyeq$ is suitable. To prove that the value of $\mathbf{t}$ is reduced it is sufficient to prove that if $\mathbf{t} \preccurlyeq t_0$ initially, then the assertion $\{\mathbf{t} \prec t_0\}$ 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:

Theorem 4.4.11 If $\preccurlyeq$ 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 $\mathbf{S}$ then if

$$\forall t_0.\, ((\mathbf{P} \wedge \mathbf{t} \preccurlyeq t_0) \Rightarrow \mathbf{S}' \;\leq\; \mathbf{S}[\{\mathbf{P} \wedge \mathbf{t} \prec t_0\};\, \mathbf{S}'/X]) \tag{4.1}$$

then $\mathbf{P} \Rightarrow (\mathbf{S}' \;\leq\; (\mu X.\mathbf{S}))$

4.5 Algorithm Derivation

It is frequently possible to *derive* a suitable procedure body $\mathbf{S}$ from the statement $\mathbf{S}'$ by applying transformations to $\mathbf{S}'$, splitting it into cases etc., until we get a statement of the form $\mathbf{S}[\mathbf{S}'/X]$ which is still defined in terms of $\mathbf{S}'$. If we can find a suitable variant function for $\mathbf{S}[\mathbf{S}'/X]$ then we can apply the theorem and refine $\mathbf{S}[\mathbf{S}'/X]$ to $(\mu X.\mathbf{S})$ which is no longer defined in terms of $\mathbf{S}'$.

As an example we will consider the familiar factorial function. Let $\mathbf{S}'$ be the statement $r := n!$. We can transform this (by appealing to the definition of factorial) to show that:

$$\mathbf{S}' \approx \textsf{if } n = 0 \textsf{ then } r := 1 \textsf{ else } r := n.(n-1)! \textsf{ fi}$$

Separate the assignment:

$$\begin{array}{l}\mathbf{S}' \approx \textsf{if } n = 0 \\ \qquad\quad \textsf{then } r := 1 \\ \qquad\quad \textsf{else } n := n - 1;\ r := n!;\ n := n + 1;\ r := n.r \textsf{ fi}\end{array}$$

So we have:

$$\mathbf{S}' \approx \textsf{if } n = 0 \textsf{ then } r := 1 \textsf{ else } n := n - 1;\ \mathbf{S}';\ n := n + 1;\ r := n.r \textsf{ fi}$$

The positive integer n is decreased before the copy of $\mathbf{S}'$, so if we set $\mathbf{t}$ to be n, Γ to be $\mathbb{N}$ and $\preccurlyeq$ to be $\leqslant$ (the usual order on natural numbers), and $\mathbf{P}$ to be **true** then we can prove that for all $n \leqslant t_0$, $\mathbf{S}'$ is refined by:

$$\textsf{if } n = 0 \textsf{ then } r := 1 \textsf{ else } n := n - 1;\ \{n < t_0\};\ \mathbf{S}';\ n := n + 1;\ r := n.r \textsf{ fi}$$

So we can apply Theorem 4.4.11 to prove that $\mathbf{S}'$ is refined by:

$$(\mu X.\ \textsf{if } n = 0 \textsf{ then } r := 1 \textsf{ else } n := n - 1;\ X;\ n := n + 1;\ r := n.r \textsf{ fi})$$

and we have derived a recursive implementation of factorial.

This theorem is a fundamental result towards the aim of a system for transforming specifications into programs since it "bridges the gap" between a recursively defined specification and a recursive procedure which implements it. It is of use even when the final program is iterative rather than recursive since many algorithms may be more easily and clearly specified as recursive functions—even if they may be more efficiently implemented as iterative procedures. This theorem may be used by the programmer to transform the recursively defined specification into a recursive procedure or function which can then be transformed into an iterative procedure.

The theorem may also be used "in reverse" to prove that a given specification is a valid abstraction of a given program: this is used for reverse engineering in Chapter 8.

4.6 Extending the Kernel Language

The kernel language we have developed is particularly elegant and tractable but is too primitive to form a useful wide spectrum language for the transformational development of programs. For this purpose we need to extend

the language by defining new constructs in terms of the existing ones using "definitional transformations". A series of new "language levels" is built up, with the language at each level being defined in terms of the previous level: the kernel language is the "level zero" language which forms the foundation for all the others. Each new language level automatically inherits the transformations proved at the previous level, these form the basis of a new transformation catalogue. Transformations of each new language construct are proved by appealing to the definitional transformation of the construct and carrying out the actual manipulation in the previous level language. This technique has proved extremely powerful and has led to the development of a practical transformation system (FermaT) which implements a large number of transformations. Over the last sixteen years, the WSL language and transformation theory have been developed in parallel: we have only added a new construct to the language *after* we have developed a sufficiently complete set of transformations for dealing with that construct. We believe that this is one of the reasons for the success of our language, as witnessed by the practical utility of the program transformation tool.

The first level language consists of the following constructs:

1. Sequential composition: The sequencing operator is associative so we can eliminate the brackets:

$$\mathbf{S}_1;\ \mathbf{S}_2;\ \mathbf{S}_3;\ \ldots;\ \mathbf{S}_n\ =_{\text{DF}}\ (\ldots((\mathbf{S}_1;\ \mathbf{S}_2);\ \mathbf{S}_3);\ \ldots;\ \mathbf{S}_n)$$

2. Deterministic Choice: We can use guards to turn a nondeterministic choice into a deterministic choice:

$$\textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi}\ =_{\text{DF}}\ (([\mathbf{B}];\ \mathbf{S}_1)\ \sqcap\ ([\neg\mathbf{B}];\ \mathbf{S}_2))$$

3. Specification statement:

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

4. Simple Assignment: If $\mathbf{Q}$ is of the form $\mathbf{x}' = \mathbf{t}$ where $\mathbf{t}$ is a list of terms which do not contain $\mathbf{x}'$ then we abbreviate the assignment as follows:

$$\mathbf{x} := \mathbf{t}\ =_{\text{DF}}\ \mathbf{x} := \mathbf{x}'.(\mathbf{x}' = \mathbf{t})$$

If $\mathbf{x}$ contains a single variable, we write $x := t$ for $\langle x \rangle := \langle t \rangle$;

5. Nondeterministic Choice: The "guarded command" of Dijkstra [7]:

$$\begin{array}{ll} \textbf{if } \mathbf{B}_1 \to \mathbf{S}_1 & =_{\text{DF}} \ (\{\mathbf{B}_1 \vee \mathbf{B}_2 \vee \cdots \vee \mathbf{B}_n\}; \\ \Box\ \mathbf{B}_2 \to \mathbf{S}_2 & \quad (\ldots((\lceil \mathbf{B}_1 \rceil;\ \mathbf{S}_1) \sqcap \\ \quad \ldots & \quad\quad (\lceil \mathbf{B}_2 \rceil;\ \mathbf{S}_2)) \sqcap \\ \Box\ \mathbf{B}_n \to \mathbf{S}_n \textbf{ fi} & \quad \ldots)) \end{array}$$

6. Deterministic Iteration: We define a **while** loop using a new recursive procedure X which does not occur free in $\mathbf{S}$:

$$\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od } =_{\text{DF}} \ (\mu X.((\lceil \mathbf{B} \rceil;\ \mathbf{S}) \ \sqcap \ [\neg \mathbf{B}]))$$

7. Nondeterministic Iteration:

$$\begin{array}{ll} \textbf{do } \mathbf{B}_1 \to \mathbf{S}_1 & =_{\text{DF}} \ \textbf{while } (\mathbf{B}_1 \vee \mathbf{B}_2 \vee \cdots \vee \mathbf{B}_n) \textbf{ do} \\ \Box\ \mathbf{B}_2 \to \mathbf{S}_2 & \quad\quad \textbf{if } \mathbf{B}_1 \to \mathbf{S}_1 \\ \quad \ldots & \quad\quad \Box\ \mathbf{B}_2 \to \mathbf{S}_2 \\ \Box\ \mathbf{B}_n \to \mathbf{S}_n \textbf{ od} & \quad\quad\quad \ldots \\ & \quad\quad \Box\ \mathbf{B}_n \to \mathbf{S}_n \textbf{ fi od} \end{array}$$

8. Initialised Local Variables:

$$\textbf{begin } \mathbf{x} := \mathbf{t} : \mathbf{S} \textbf{ end } =_{\text{DF}} \ (\mathsf{add}(\mathbf{x});\ (\lceil \mathbf{x} = \mathbf{t} \rceil;\ (\mathbf{S};\ \mathsf{remove}(\mathbf{x}))))$$

9. Counted Iteration. Here, the loop body $\mathbf{S}$ must not change i, b, f or s:

$$\begin{array}{ll} \textbf{for } i := b \textbf{ to } f \textbf{ step } s \textbf{ do } \mathbf{S} \textbf{ od } =_{\text{DF}} & \textbf{begin } i := b: \\ & \quad \textbf{while } i \leqslant f \textbf{ do} \\ & \quad\quad \mathbf{S};\ i := i + s \textbf{ od end} \end{array}$$

10. Block with procedure calls:

$$\textbf{begin } \mathbf{S} \textbf{ where proc } X \ \equiv \ \mathbf{S}' \textbf{. end } =_{\text{DF}} \ \mathbf{S}[(\mu X.\mathbf{S}')/X]$$

One aim for the design of the first level language is that it should be easy to determine which statements are potentially null. A guard statement such as $[x = 1]$ is one example: if the preceding statements do not allow 1 as a possible value for x at this point then the statement is null. The guard [**false**] is another example which is always null. If a state transformation is non-null for every initial state then it is called *null-free*. We claim that all first-level language statements without explicit guard statements are null free. (This is why we do not include Morgan's specification statement $x\colon [\mathsf{Pre}, \mathsf{Post}]$ in the

first level language, because it cannot be guaranteed null-free. For example the specification $\langle\rangle$: [**true**, **false**] is equivalent to [**false**] which is everywhere null).

A null-free statement will satisfy Dijkstra's "Law of the Excluded Miracle" [7]:

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

.

The level two language introduces multi-**exit** loops and Action systems (cf [1,2]). Level three adds local variables and parameters to procedures, functions and expressions with side effects.

4.7 Example Transformations

This section introduces some basic program transformations which are useful in their own right and which also form the "building blocks" for more powerful transformations.

4.7.1 Notation

Sequences: $s = \langle a_1, a_2, \ldots, a_n\rangle$ is a sequence, the ith element a_i is denoted $s[i]$, $s[i\,..\,j]$ is the subsequence $\langle s[i], s[i{+}1], \ldots, s[j]\rangle$, where $s[i\,..\,j] = \langle\rangle$ (the empty sequence) if $i > j$. The length of sequence s is denoted $\ell(s)$, so $s[\ell(s)]$ is the last element of s. We use $s[i\,..]$ as an abbreviation for $s[i\,..\,\ell(s)]$.

Sequence concatenation: $s +\!\!+\, t = \langle s[1], \ldots, s[\ell(s)], t[1], \ldots, t[\ell(t)]\rangle$.

Subsequences: The assignment $s[i\,..\,j] := t[k\,..\,l]$ where $j{-}i = l{-}k$ assigns s the value $\langle s[1], \ldots, s[i-1], t[k], \ldots, t[l], s[j+1], \ldots, s[\ell(s)]\rangle$.

Stacks: Sequences are also used to implement stacks, for this purpose we have the following notation: For a sequence s and variable x: $x \xleftarrow{\text{pop}} s$ means $x := s[1];\ s := s[2\,..]$. For a sequence s and expression e: $s \xleftarrow{\text{push}} e$ means $s := \langle e\rangle +\!\!+\, s$.

Map: The map operator $*$ returns the sequence obtained by applying a given function to each element of a given sequence: $(f * \langle a_1, a_2, \ldots, a_n\rangle) = \langle f(a_1), f(a_2), \ldots, f(a_n)\rangle$.

Reduce: The reduce operator / applies an associative binary operator or function to a list and returns the resulting value: $(\oplus/\langle a_1, a_2, \ldots, a_n\rangle) =$

$a_1 \oplus a_2 \oplus \cdots \oplus a_n$. So, for example, if s is a list of integers then $+/s$ is the sum of all the integers in the list, if q is a list of lists then $+/(\ell * q) = \ell(+\!\!+/q)$ is the total length of all the lists in q.

Projection: The projection functions π_1, π_2,... are defined as $\pi_1(\langle x, y\rangle) = x$, $\pi_2(\langle x, y\rangle) = y$, and more generally, for any sequence s: $\pi_i(s) = s[i]$.

The operation of splitting a sequence into a sequence of non-empty sections at the points where a predicate fails is generally useful so we will define the following notation:

Suppose we have a sequence p which we want to split into sections at those points i where the predicate $B(p[i], p[i+1])$ is false. In other words, we want to define a new sequence of non-empty sequences q such that the concatenation of the sequences in q is equal to p (i.e. $+\!\!+/q = p$) and B is true within each section and false on the boundary from one section to the next.

Define the function $\mathsf{index}_q \colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ by $\mathsf{index}_q(j,k) = +/(\ell * q[1\,..\,j-1]) + k$. This function maps the position of an element in the q structure (the kth component of the jth subsequence) into the corresponding position in the p structure. For all $j \in 1\,..\,\ell(q)$ and $k \in 1\,..\,\ell(q[j])$ we have $p = +\!\!+/q \Rightarrow p[\mathsf{index}_q(j,k)] = q[j][k]$. On this domain, index_q is 1–1, so it has a well-defined inverse. This inverse index_q^{-1} maps an index i of p to a pair $\langle j, k\rangle$ such that $p[i] = q[j][k]$. So the function $\mathsf{section}_q = \pi_1 \cdot \mathsf{index}_q^{-1}$ will give the section in q in which an element of p occurs.

With this notation, we can define a split function $\mathsf{split}(p, B) = q$ which splits p into non-empty sections with the section breaks occurring between those pairs of elements of p where B is false. The formal definition uses $\mathsf{section}_q$ to find the "section breaks":

Definition 4.7.1 $\mathsf{split}(p, B) = q$ *where:*

$$
\begin{aligned}
&(+\!\!+/q) = p \ \wedge\ \langle\rangle \notin \mathsf{set}(q) \\
&\quad \wedge\ \forall i \in 1\,..\,\ell(p) - 1.\ \big((B(p[i], p[i+1]) \Rightarrow \mathsf{section}_q(i+1) = \mathsf{section}_q(i)) \\
&\qquad\qquad \wedge\ (\neg B(p[i], p[i+1]) \Rightarrow \mathsf{section}_q(i+1) = \mathsf{section}_q(i) + 1)\big)
\end{aligned}
$$

The split function will be used in the re-engineering case study in Chapter 9.

4.7.2 Examples of Transformations

In this section we describe a few of the transformations we will use later:

Expand IF statement

The **if** statement:

$$\textbf{if B then } S_1 \textbf{ else } S_2 \textbf{ fi}; \ S$$

can be expanded over the following statement to give:

$$\textbf{if B then } S_1; \ S \textbf{ else } S_2; \ S \textbf{ fi}$$

4.7.3 Loops and exits

Statements of the form **do S od**, where **S** is a statement, are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form **exit**(n) (where n is an integer, *not* a variable or expression) which causes the program to exit the n enclosing loops. To simplify the language we disallow **exit**s which leave a block or a loop other than an unbounded loop. This type of structure is described in [13] and more recently in [20];

A simple transformation is the following: If **S** is a *proper sequence* then:

$$\Delta \vdash \textbf{do if B then exit fi}; \ S \textbf{ od} \ \approx \ \textbf{while } \neg B \textbf{ do } S \textbf{ od}$$

A proper sequence is any statement within which each **exit**(n) occurs nested within at least n loops. Such a statement cannot therefore terminate any enclosing **do** ... **od** loop: the next statement to be executed will always be the next statement in the sequence.

If $\mathbf{S}_1$ is a proper sequence, then the loop:

$$\textbf{do } S_1; \ S_2 \textbf{ od}$$

can be "inverted" to:

$$S_1; \ \textbf{do } S_2; \ S_1 \textbf{ od}$$

This transformation can be used in the forward direction in order to move the **exit** statements closer to the top of the loop (preparatory to converting to a **while** loop perhaps). It can also be used in the reverse direction to merge the two copies of statement $\mathbf{S}_1$ into a single copy ans so reduce the size of the program.

4.7.4 Action Systems

An *action* is a parameterless procedure acting on global variables (cf [1,2]). It is written in the form $A \equiv \mathbf{S}.$ where A is a statement variable (the name of the action) and **S** is a statement (the action body). A set of (mutually

recursive) actions is called an *action system*. An occurrence of a statement **call** X within the action body refers to a call of another action. The action bodies may include calls to the special action Z, which does not have a body. Instead, a **call** Z causes immediate termination of the whole action system even if there are unfinished recursive calls.

A *regular* action system is one in which execution of any action body always leads to an action call (which may be a **call** Z). Within such a system, no action call ever returns and the system can only terminate by calling Z. Such an action system is equivalent to a collection of labels and **goto** statements: in fact, any program which is implemented using labels and **goto**s can be translated into a regular action system.

4.8 Why Invent WSL?

For restructuring purposes it is useful to work within a language which has the following features:

- Simple, regular, and formally defined semantics;
- Simple, clear, and unambiguous syntax;
- A wide range of transformations with simple, mechanically-checkable correctness conditions.

No existing programming language which is widely in use today meets *any* of these criteria.

For reverse engineering it is extremely useful to work within a single wide-spectrum language within which both low-level programs and high-level abstract specifications are easily represented.

For migration between programming languages it is important that the transformation system language should not be biased towards a particular source or target language.

These are the considerations which led to the development of the WSL language. The language has been developed gradually over the last ten years, in parallel with the development of the transformation theory. This parallel development has ensured that WSL is ideally suited for program transformation work: the design of the language ensures that developing and proving the correctness of transformations is straightforward and, most importantly, the correctness conditions for the transformations are easy to check mechanically. This last point was important for the success of our transformation system.

We believe that the formal foundations of our language and transformation theory were essential to the success of the project. The practice of implementing any reasonable-looking transformation *without* a formal proof of correctness is very dangerous: the author has discovered errors in transformations published in reputable journals [Arsac Syntactic 1982], but the errors were only uncovered after having attempted (and failed) to prove that the transformations were correct. Since our tool works by applying a vast number of transformations in sequence, any unreliability in the transformations will have serious repercussions on the reliability of the tool. In practice, the work on proving the correctness of known transformations has been a major driving force in the discovery of new transformations.

As a result, it turns out that, unlike any existing programming language, WSL is *not* Turing equivalent, for the following two reasons.

1. WSL includes constructs such as "guard" and "join" which are not implementable. For example, the guard $[x > 0]$ is guaranteed to terminate, does not change the value of any variable, and guarantees that $x > 0$ on termination. Guards and join are very useful for writing specifications, and therefore equally important for reverse engineering. (see below)

2. Even if one were to exclude the "miracles" introduced by guard and join, WSL is still more powerful than a Turing machine, since it is based on first order logic. It is possible to write a WSL program which solves the halting problem for Turing machines. (But not possible to write a Turing machine which solves the halting problem for Turing machines).

To prove the second point, note the following:

1. There exists a Turing machine $\mathsf{will\_term}(x, n)$ which can determine if the given (encoding of a) Turing machine x will terminate in n or fewer steps. (The machine "simulates" x for n steps—the details can be found in any text on computation theory);

2. This Turing machine can be translated into a WSL program $\mathbf{T}$ which takes x and n as input variables and sets output variable r to 0 or 1 as appropriate;

3. From $\mathbf{T}$ one can construct the formula $\mathrm{WP}(\mathbf{T}, r = 1)$, with free variables x and n, which is true iff the value of x encodes a Turing machine which will terminate in n or fewer steps;

4. Then the WSL procedure:

   proc Will_Term(x) $\equiv$
     if $\exists n \in \mathbb{N}.\, \mathrm{WP}(\mathbf{T}, r = 1)$ **then** $r := 1$
                                else $r := 0$ **fi.**

   will set r to 1 iff the given x encodes a Turing machine which will terminate in *any* number of steps. (Note $\mathbb{N}$ denotes the set of all positive integers). This completes the proof.

General formulae, unrestricted set operations, references to infinite objects (eg the set $\mathbb{N}$ above), and so on, are not allowed in executable programming languages, but are essential for a useable specification language. Therefore, they are equally important for a language which is to form the basis for a reverse engineering system.

For a useable program transformation system it is *essential* that the base language satisfies the "replacement property". Informally, the property is: "Replacing any component of a program by a semantically equivalent component will result in a semantically equivalent program". This property is foundational to how our tool works: select a component, apply a transformation, select another component... However, the authors are aware of *no* commercial programming language which satisfies this property.

For example, in C the statement `x = x*2 + 1` is equivalent to `x = x*2; x = x + 1`. But the statement `if (y == 0) x = x*2 + 1` is *not* equivalent to `if (y == 0) x = x*2; x = x + 1`.

In JOVIAL, among many other restrictions, there is a limit to the level of nesting of FOR loops. Therefore any transformation for replacing a GOTO construct with an equivalent FOR loop will fail in certain positions[1]. Particular implementations of other languages will almost certainly have similar limitations and restrictions which make it extremely difficult, if not impossible, to discover valid semantics-preserving transformations in that language: let alone prove their correctness.

An obvious disadvantage of working in a separate language to the source language of the legacy system is that translators to and from WSL will have to be written. Fortunately, for the "old fashioned" languages typical of legacy systems, this is not much more difficult than writing a parser for the language, which in turn is a simple application of well-developed compiler technology for which there is a wide variety of tool support available. In addition, there are three important advantages to our approach:

[1] We recall encountering a similar problem with BASIC on a Compukit UK101 microprocessor system many years ago!

1. Using a collection of translators for different languages, it becomes possible to migrate from one language to another via WSL. We are currently working on an Assembler to COBOL II migration: the aim is to produce "high level" COBOL II, not something which looks as though it was written by an Assembler programmer!

2. The second is that the "translator" can be very simple-minded and not have to worry about introducing redundancies, dead code, unstructured code etc. Once we are within the formal language and transformation system, such redundancies and infelicities can be eliminated automatically by applying a series of general-purpose restructuring, simplification and data-flow analysis transformations.

3. Thirdly, our sixteen year's work on transformation theory can be re-applied to a new language simply by writing a translator for that language. It would be impossible to re-use the development work for a COBOL transformation system in the development of a JOVIAL transformation system. Even a different version of COBOL could invalidate many transformations and involve a lot of re-work.

Based on our results, translation to a formal language is the best way to set about any serious reverse engineering or migration work.

Our work has been criticised by some practitioners for its emphasis on the use of formal methods and formally specified languages. This is odd because the programming language and its support libraries form the basic building materials for software engineering. But no serious engineer would expect to build with components whose properties are not precisely, formally, concisely specified (eg. this beam is specified to be able to take this much load under these operating conditions, etc.) No serious engineer would tolerate "standard" components which differ in an undefined way in their properties and behaviour from supplier to supplier. A serious engineer does not think twice about screwing a nut from one supplier onto a bolt from another supplier: he expects them to fit as a matter of course! A serious engineer expects to have to master a certain amount of mathematics in order to do his or her job properly: differential equations, integration, fluid dynamics, stress modelling, etc. This is far more than the elementary set theory and logic required to understand WSL.

With regard to the "undefined" behaviour of many commercial languages in the presence of syntactic or semantic errors (including out of bounds subscripts and the infamous "buffer overflow" problem) Hoare [10] said:

> In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.

This was way back in 1960.

D. L. Parnas at the International Conference on Software Engineering in Baltimore, Maryland in 1993 [18] made the following points on the relationship between software engineers and "real" engineering:

- "Engineering" is defined as "The use of science and technology to build useful artifacts";
- Classical engineers use mathematics to describe their products (calculus, PDEs, nonlinear functions, etc.);
- Computer systems designers should use engineering methods if they are to deserve the name "Software Engineers". This will include the use of mathematics.

4.9 References

[1] Arsac, J., "Transformation of Recursive Procedures," in *Tools and Notations for Program Construction*, D. Neel, Ed. Cambridge: Cambridge University Press, pp. 211–265, 1982.

[2] Arsac, J., "Syntactic Source to Source Transforms and Program Manipulation," *Comm. ACM*, 22, no. 1, pp. 43–54, Jan., 1979.

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

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

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

[6] Bull, T., "An Introduction to the WSL Program Transformer," presented at Conference on Software Maintenance 26th–29th November 1990, San Diego, Nov., 1990.

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

[8] Engeler, E., *Formal Languages: Automata and Structures*. Chicago, Markham, 1968.

[9] Hayes, I. J., *Specification Case Studies*. Englewood Cliffs, NJ, Prentice-Hall, 1987.

[10] Hoare, C. A. R., "The Emperor's Old Clothes: The 1980 ACM Turing Award Lecture," *Comm. ACM*, 24, no. 4, pp. 75–83, Feb., 1981.

[11] Jones, C. B., *Systematic Software Development using VDM*. Englewood Cliffs, NJ, Prentice-Hall, 1986.

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

[13] Knuth, D. E., "Structured Programming with the GOTO Statement," *Comput. Surveys*, 6, no. 4, pp. 261–301, 1974.

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

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

[16] Morgan, C. C. and K. Robinson, "Specification Statements and Refinements," *IBM J. Res. Develop.*, 31, no. 5, 1987.

[17] Morgan, C. C. and T. Vickers, *On the Refinement Calculus*. New York–Heidelberg–Berlin, Springer-Verlag, 1993.

[18] Parnas, D. L., Presentation at the International Conference on Software Engineering, Baltimore, 21st–23rd May 1993.

[19] Scott, D., "Logic with Denumerably Long Formulas and Finite Strings of Quantifiers," in *Symposium on the Theory of Models*, J. Addison, L. Henkin and A. Tarski, Eds. Amsterdam: North-Holland, pp. 329–341, 1965.

[20] Taylor, D., "An Alternative to Current Looping Syntax," *SIGPLAN Notices*, 19, no. 12, pp. 48–53, Dec., 1984.

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

[22] Ward, M., "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/sorting-t.ps.gz`⟩.

[23] Ward, M., "A Recursion Removal Theorem—Proof and Applications," Durham University, Technical Report, 1991, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/rec-proof-t.ps.gz`⟩.

[24] Ward, M., "A Recursion Removal Theorem," Springer-Verlag, New York–Heidelberg–Berlin, Proceedings of the 5th Refinement Workshop, London, 8th–11th January, 1992, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/ref-ws-5.ps.gz`⟩.

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

[26] Ward, M., "Recursion Removal/Introduction by Formal Transformation: An Aid to Program Development and Program Comprehension," *Comput. J.*, 42, no. 8, pp. 650–673, 1999.

[27] Ward, M., "A Definition of Abstraction," *J. Software Maintenance: Research and Practice*, 7, no. 6, pp. 443–450, Nov., 1995, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/abstraction-t.ps.gz`⟩.

[28] Ward, M., "Derivation of Data Intensive Algorithms by Formal Transformation," *IEEE Trans. Software Eng.*, 22, no. 9, pp. 665–686, Sept., 1996, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/sw-alg.ps.gz`⟩.

[29] Ward, M. and K. H. Bennett, "A Practical Program Transformation System For Reverse Engineering," presented at Working Conference on Reverse Engineering, May 21–23, 1993, Baltimore MA, 1993, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/icse.ps.gz`⟩.

[30] Ward, M., F. W. Calliss and M. Munro, "The Maintainer's Assistant," presented at Conference on Software Maintenance 16th–19th October 1989, Miami Florida, 1989, ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/MA-89.ps.gz`⟩.