Foundations for a Practical Theory of Program Refinement and Transformation

M. P. Ward*

martin@gkc.org.uk

http://www.cse.dmu.ac.uk/∼mward/

1994

Abstract

A wide spectrum language is presented, which is designed to facilitate the proof of the correctness of refinements and transformations. Two different proof methods are introduced and used to prove some fundamental transformations, including a general induction rule (Lemma 3.9) which enables transformations of recursive and iterative programs to be proved by induction on their finite truncations. A theorem for proving the correctness of recursive implementations is presented (Theorem 3.21), which provides a method for introducing a loop, without requiring the user to provide a loop invariant. A powerful, general purpose, transformation for removing or introducing recursion is described and used in a case study (Section 5) in which we take a small, but highly complex, program and apply formal transformations in order to uncover an abstract specification of the behaviour of the program. The transformation theory supports a transformation system, called FermaT, in which the applicability conditions of each transformation (and hence the correctness of the result) are mechanically verified. These results together considerably simplify the construction of viable program transformation tools; practical consequences are briefly discussed.

KEYWORDS: program transformation, refinement, reverse engineering, formal methods, wide spectrum language, specification statement.

1 Introduction

There has been much research in recent years on the formal development of programs by refining a specification to an executable program via a sequence of intermediate stages, where each stage is proved to be equivalent to the previous one, and hence the final program is a correct implementation of the specification. However, there has been very little work on applying program transformations to reverse-engineering and program understanding. This may be because of the considerable technical difficulties involved: in particular, a refinement method has total control over the structure and organisation of the final program, while a reverse-engineering method has to cope with any code that gets thrown at it: including unstructured ("spaghetti") code, poor documentation, misuse of data structures, programming "tricks", and undiscovered errors. Any *practical* program transformation method cannot afford to ignore this problem, because it cannot ignore the huge body of source code currently in use, and it cannot ignore the inevitable need for maintenance, enhancement and modification of the programs developed using formal refinement. A particular problem with most refinement methods is that the introduction of a loop construct requires the user to determine a suitable invariant for the loop, together with a variant expression, and to prove:

*Department of Computer Science University of Durham, Durham, UK

1. That the invariant is preserved by the body of the loop;
2. The variant function is decreased by the body of the loop;
3. The invariant plus terminating condition are sufficient to implement the specification.

To use this method for reverse engineering would require the user to determine the invariants for arbitrary (possibly large and complex) loop statements. This is extremely difficult to do for all but the smallest "toy" programs. Therefore, a different approach to reverse engineering is required: the approach presented in this paper does not require the use of loop invariants to deal with arbitrary loops, although if invariants are available, the information they provide can be made use of.

Our aim in this paper is to present the foundations for a practical theory of program refinement and transformation which can be applied to program development and reverse engineering, although our primary focus will be on the latter problem. The theory is based on the concept of a "Wide Spectrum Language", which includes both low-level programming constructs and high-level abstract specifications within a single language. Such a language forms an ideal tool for developing methods for formal program development, and also for formal reverse engineering (for which we have coined the term *inverse engineering*), because the proof that a program correctly implements a specification, or that a specification correctly captures the behaviour of a program, can be achieved by means of formal (semantic-preserving) transformations in the language.

Over the last twelve years we have been developing this wide spectrum language (called WSL), in parallel with the development of a transformation theory and proof methods, together with methods for program development and inverse engineering. The fundamental transformations presented in this paper form the basic toolkit used in [War89, War92, War99] to extend the kernel WSL language into a powerful programming and specification language, and to develop an extensive catalogue of program transformations in the extended language.

The *Refinement Calculus* approach to program derivation [HHJ87, Mor94, MRG88] is superficially similar to our program transformation method. It is based on a wide spectrum language, using Morgan's specification statement [Mor88] and Dijkstra's guarded commands [Dij76]. 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 reverse-engineering method. Morgan remarks (pp 166–167 of [Mor94]) that the whole development history is required for understanding the structure of the program and making safe modifications. Unfortunately, there are many billions of lines of code in existence for which we do not have the luxury of a complete development history! All we have is the code, and some incomplete, inaccurate, and out of date documentation and comments. All we can rely on is the code itself—and the refinement calculus cannot help us to understand the code, since we need to understand it (to the extent of providing suitable invariants and variant functions for all the loops) before we can start calculating with it. By contrast, the program transformation approach is equally suitable for forward and reverse engineering—we give an example of reverse engineering using formal transformations in Section 5. Starting with the source code of a program we can apply transformations to simplify its structure and uncover a specification, without initially understanding anything of the purpose of the program.

In developing a model based theory of semantic equivalence, we use the popular approach of defining a core "kernel" language with denotational semantics, and permitting definitional extensions in terms of the basic constructs. In contrast to other work (for example, [BMP89, Bir87, Par84]) we do not use a purely applicative kernel; instead, the concept of state is included, using a *specification statement* which also allows specifications expressed in first order logic as part of the language, thus providing a genuine wide spectrum language.

Fundamental to our approach is the use of infinitary first order logic (see [Kar64]) both to express the weakest preconditions of programs [Dij76] and to define assertions and guards in the kernel language. Engeler [Eng68] was the first to use infinitary logic to describe properties of programs; Back [Bac80] used such a logic to express the weakest precondition of a program as a logical formula, although his kernel language was limited to simple iterative programs. We use a different kernel language which includes recursion and guards, so that Back's language is a subset of ours. We show that the introduction of infinitary logic as part of the language (rather than just the metalanguage of weakest preconditions), together with a combination of proof methods using both denotational semantics and weakest preconditions, is a powerful theoretical tool which allows us to prove some general transformations and representation theorems.

The denotational semantics of the kernel language is based on the semantics of infinitary first order logic. Kernel language statements are interpreted as functions which map an initial state to a set of final states. This *set* of final states, rather than a single final state, models the nondeterminacy in the language: for a deterministic program this set will contain a single final state. A program $\mathbf{S}_1$ is a refinement of $\mathbf{S}_2$ if, for each initial state, the set of final states for $\mathbf{S}_1$ is a subset of the final states for $\mathbf{S}_2$. Back and von Wright [BaW90] note that the refinement relation can be characterised using weakest preconditions in higher order logic (where quantification over formulae is allowed). For any two programs $\mathbf{S}_1$ and $\mathbf{S}_2$, the program $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$ if the formula $\forall \mathbf{R}.\, \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$. This approach to refinement has two problems:

1. It has not been proved that for *all* programs $\mathbf{S}$ and formulae $\mathbf{R}$, there exists a finite formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ which expresses the weakest precondition of $\mathbf{S}$ for postcondition $\mathbf{R}$. Can proof rules justified by an appeal to WP in *finitary* logic be justifiably applied to arbitrary programs, for which the appropriate *finite* $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ may not exist? This problem does not occur with infinitary logic, since $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ has a simple definition for all programs $\mathbf{S}$ and all (infinitary logic) formulae $\mathbf{R}$;
2. Second order logic is *incomplete* in the sense that not all true statements are provable. So even if the refinement is true, there is no guarantee that the refinement can be proved.

This paper presents solutions to both of these problems. Using infinitary logic allows us to give a simple definition of the weakest precondition of *any* statement (including an arbitrary loop) for any postcondition. Secondly, we show that for each pair of statements $\mathbf{S}_1$ and $\mathbf{S}_2$ there is a single postcondition $\mathbf{R}$ such that $\mathbf{S}_1$ is a refinement of $\mathbf{S}_2$ iff both $\mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ and $\mathrm{WP}(\mathbf{S}_1, \mathsf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathsf{true})$, so no quantification over formulae or Boolean predicates is required. Thirdly, the infinitary logic we use is complete: this means that if there *is* a refinement then there is also guaranteed to be a proof of the corresponding formula—although the proof may be infinitely long! However, it is perfectly practical to construct infinitely long proofs: in fact the proofs of many transformations involving recursion or iteration are infinite proofs constructed by induction (see Section 3 for example). Thus infinitary logic is both necessary and sufficient for proving refinements and transformations.

We consider the following criteria to be important for any practical wide-spectrum 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 [Hay87] and **VDM** [Jon86] both use the language of mathematical logic and set theory (in different notations) to define specifications. The "Representation Theorem" (Theorem 3.5) proves that our specification statement is sufficient to specify *any* WSL program (and therefore any computable function, since WSL is certainly Turing complete);
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 speci-

fying 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 to bridge the "abstraction gap" between specifications and programs. See Section 3.4.3 and [War93, YoW93] for examples;
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". See [War90, War92, War96] for examples;
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. See [WaB93] for example.

A system which meets all these requirements would have immense practical importance in the following areas:

- Improving the maintainability (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 at the appropriate level of abstraction, 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. The use of the **join** construct (Section 4.1) as an indexing mechanism is discussed in [War91].

1.1 Outline of the Paper

In Section 2 we describe the syntax and denotational semantics of the kernel language. We make use of two different formulations of Dijkstra's weakest precondition [Dij76]: the first (denoted wp) is a function which maps the semantics of a program and a condition on the final state space to a condition on the initial state space (a condition on a state space is simply a set of states: those states which satisfy the condition). The second (denoted WP) is a function which maps the syntax of a program and a formula of first order logic to another formula of first order logic. We prove that these two definitions are equivalent, given a suitable interpretation of formulae as state conditions.

In Section 3 we develop some basic proof rules for proving the correctness of transformations. These enable us to prove a "Representation Theorem" (Theorem 3.5) which shows that any program can be automatically transformed into an equivalent specification. We also develop a fundamental transformation for the recursive implementation of specifications, and a proof rule for proving termination of recursive programs. These results make the connection between abstract, recursively-defined specifications and the recursive and iterative procedures which implement them.

In Section 4 we present the first set of extensions to the kernel language. We also introduce a **join** construct: the **join** of two programs or specifications is a program which simultaneously meets all the specifications which either of its components can meet. Equivalently, the **join** of two programs is the weakest program which refines both components. This is a powerful tool for specifying complex programs in that different aspects of the program can be specified separately (for example: mainline code and error cases) and then **join**ed together. We have also recently started using **join** as a form of parallel operator for programs with shared memory (see [YoW93]). Finally, in Section 5 we give an example of a powerful transformation for both forward and reverse engineering and its application to a reverse engineering problem.

2 Syntax and Semantics of the Kernel Language

2.1 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 sequences 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. The arbitrary values may of course be restricted to particular values by a subsequent guard;
4. **Remove variables:** $\mathsf{remove}(\mathbf{y})$ removes the variables in $\mathbf{y}$ from the state space (if they are present).

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

This very simple kernel language is all we need to construct our wide spectrum language, for example an assignment such as $x := 1$ is constructed by adding x and restricting its value: $(\mathsf{add}(\langle x\rangle);\ [x = 1])$. For an assignment such as $x := x + 1$ we need to record the new value of x in a new variable, x' say, before copying it into x. So we can construct $x := x + 1$ as follows: $(\mathsf{add}(\langle x'\rangle);\ ([x' = x + 1];\ (\mathsf{add}(\langle x\rangle);\ ([x = x'];\ \mathsf{remove}(x')))))$

Three fundamental statements can be defined immediately:

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

where **true** and **false** are universally true and universally false formulae, defined: $\textbf{true} =_{\text{DF}} \forall x.\,(x = x)$ and $\textbf{false} =_{\text{DF}} \neg\forall x.\,(x = x)$.

A statement whose set of final states may be empty is called a "null statement", an example is the guard, [**false**], which is a "correct refinement" of *any* specification whatsoever (see the definition of refinement in Section 2.4). Clearly, any null statement, and guard statements in general cannot be directly implemented, but they are nonetheless a useful theoretical tool. Since 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 are (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.

2.2 The Specification Statement

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

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

An important property of this specification statement is that it is guaranteed null-free. A simple example is the assignment $\langle x \rangle := \langle x' \rangle.(x' = x + 1)$ which increments the value of x by one, without affecting any other variable. For a more interesting 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 [War90] 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.

Morgan and others [Mor88, Mor94, MoR87, MoV93] 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 of *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 $(\{\mathsf{Pre}\};\ (\mathsf{add}(\mathbf{x});\ [\mathsf{Post}]))$. The disadvantage with Morgan's notation is that it makes the user responsible for ensuring that he never refines a specification into an (unimplementable) null statement.

2.3 Conditional Statements

Conditional statements are defined using a combination of guards and nondeterministic choice, for example: **if** $\mathbf{B}$ **then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi** can be expressed in the kernel language as:

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

while a two-way guarded command [Dij76] such as: **if** $\mathbf{B}_1 \ \rightarrow \ \mathbf{S}_1 \ \square \ \mathbf{B}_2 \ \rightarrow \ \mathbf{S}_2$ **fi** can be expressed as

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

Note that the statement will abort if neither $\mathbf{B}_1$ nor $\mathbf{B}_2$ is true. This will be extended to an n-way guarded command in the obvious way, see Section 4 for the first set of extensions to the kernel language.

2.4 Semantics of the Kernel Language

In this section we will describe the denotational semantics [Ten76] of kernel statements in terms of mathematical objects which we call "state transformations". Unlike the CIP project [BB85, BMP89, BaT87, BaT87] and others (eg [BaW82, Bir87]) our kernel language will have state introduced right from the start so that it can cope easily with imperative programs. We also use a constructive rather than algebraic style of specification, since Majester [Maj77] has shown some fundamental limits with purely algebraic specification methods. Our experience is that an imperative kernel language with functional extensions is more tractable than a functional kernel language with imperative extensions. Unlike Bird [Bir87] we did not want to be restricted to a purely functional language since this is incompatible with the aims of a true wide spectrum language.

A *proper state* is a function which gives values to a particular (finite, non-empty) collection of variables. There is also a special state $\perp$ which indicates nontermination or error. As discussed above, the semantics of statements are based on the semantics of infinitary logic. A *structure* for the logic is a set of values and a function which maps the constant symbols, predicate symbols and function symbols of the logic to elements, predicates and functions on the set of values. Such a structure defines an interpretation of formulae as state predicates (sets of proper states) and statements as state transformations (functions from a state to a set of states). If $\perp \in f(s)$ for state transformation f and initial state s, then $f(s)$ includes all possible final states. See [War89] for the details of the interpretation of statements. If V and W are finite non-empty sets of variables, and $\mathbf{S}$ is a statement, then we write $\mathbf{S}\colon V \to W$ (a trinary relation) if V and W are consistent input and output state spaces for $\mathbf{S}$. For example, the program $\mathsf{add}(\mathbf{x})$ must include $\mathbf{x}$ in its output state space, while $\mathsf{remove}(\mathbf{x})$ must not. The program $(\mathsf{add}(\mathbf{x}) \sqcap \mathsf{remove}(\mathbf{x}))$ has *no* consistent input and output state spaces.

The refinement relation for state transformations is very simple: state transformation f_1 is refined by state transformation f_2, written $f_1 \leq f_2$, if and only if they have the same initial and final state spaces, and for each initial state s, $f_2(s) \subseteq f_1(s)$.

We define three functions for composing state transformations: sequential composition, choice and join:

$$\begin{aligned}
(f_1; f_2)(s) &=_{\text{DF}} \bigcup \{\, f_2(s') \mid s' \in f_1(s) \,\} \\
(f_1 \sqcap f_2)(s) &=_{\text{DF}} f_1(s) \cup f_2(s) \\
(f_1 \sqcup f_2)(s) &=_{\text{DF}} f_1(s) \cap f_2(s)
\end{aligned}$$

Sequencing and choice correspond with the first two compound statement types in the kernel language. The join construct is not included in the kernel language because it is expressable in terms of the existing kernel language constructs. The interpretation of join is that any specification satisfied by either f_1 or f_2 will also be satisfied by $(f_1 \sqcup f_2)$. Compare this with choice where only those specifications satisfied by *both* f_1 and f_2 will be satisfied by $(f_1 \sqcap f_2)$. If one of f_1 and f_2 does not terminate then the join acts the same as the terminating state transformation. If f_1 and f_2 are *inconsistent* on s (i.e. $f_1(s)$ and $f_2(s)$ have no states in common) then the result of the join will be null on s.

Let V and W be finite non-empty sets of variables, and $\mathcal{H}$ be a non-empty set of values. We write $V_{\mathcal{H}}$ for the set of states $\{\perp\} \cup \mathcal{H}^V$ and $E_{\mathcal{H}}(V)$ for the set of state predicates $\wp(\mathcal{H}^V)$ (the set of all sets of proper states), and $F_{\mathcal{H}}(V, W)$ for the set of state transformations from V to W. These are the functions $f : V_{\mathcal{H}} \to \wp(W_{\mathcal{H}})$ such that $\perp \in F(\perp)$ and $\forall s \in V_{\mathcal{H}}. \perp \in f(s) \Rightarrow f(s) = W_{\mathcal{H}}$. Given a structure M for the infinitary logic, the function int_M maps a statement and an initial state space to the interpretation of the statement as a state transformation. If $\mathbf{S}$ is any statement and V and W are finite, non-empty sets of variables where $\mathbf{S}\colon V \to W$, then $\mathrm{int}_M(\mathbf{S}, V)$ is the state transformation which gives the meaning of $\mathbf{S}$. (Note that we don't need to include W as a parameter to int_M since it is uniquely defined by $\mathbf{S}$ and V).

Three fundamental state transformations in $F_{\mathcal{H}}(V,V)$ are: Ω, Θ and Λ. These give the semantics of the statements **abort**, **null** and **skip**. For each proper $s \in V_{\mathcal{H}}$:

$$\Omega(s) \;=_{\text{DF}}\; V_{\mathcal{H}} \qquad \Theta(s) \;=_{\text{DF}}\; \varnothing \qquad \Lambda(s) \;=_{\text{DF}}\; \{s\}$$

Thus Ω is nowhere defined and never terminates, Λ is everywhere defined and non-null and always terminates in the same state it is started in, and Θ is everywhere defined and everywhere null.

With these definitions we see that $\langle F_{\mathcal{H}}(V,W), \leq \rangle$ forms a lattice structure [DaP90] with Θ as the top element, Ω as the bottom element, $\sqcap$ as the lattice meet operator, and $\sqcup$ as the lattice join operator. If for every $F \subseteq F_{\mathcal{H}}(V,W)$ we define $\bigsqcup F$ and $\bigsqcap F$ by $(\bigsqcup F)(s) \;=_{\text{DF}}\; \bigcap\{\, f(s) \mid f \in F \,\}$ and $(\bigsqcap F)(s) \;=_{\text{DF}}\; \bigcup\{\, f(s) \mid f \in F \,\}$ then we have a *complete* lattice structure (see [DaP90]). It is order isomorphic to a sublattice of $\langle \wp(V_{\mathcal{H}} W_{\mathcal{H}}), \subseteq \rangle$ whose meet is $\cap$ and join is $\cup$, the embedding ψ is defined:

$$\text{For } f \in F_{\mathcal{H}}(V,W)\text{:} \quad \psi(f) = \{\, \langle s,t \rangle \mid s \in V_{\mathcal{H}} \wedge t \in f(s) \,\}$$

ψ is join and meet preserving and is a 1–1 map so it is a 1–1 lattice homomorphism (see [DaP90]). The inverse function $\phi\colon \wp(V_{\mathcal{H}} W_{\mathcal{H}}) \to F_{\mathcal{H}}(V,W)$ defined for $\rho \subseteq V_{\mathcal{H}} W_{\mathcal{H}}$ by:

$$\phi(\rho)(s) \;=_{\text{DF}}\; \begin{cases} W_{\mathcal{H}} & \text{if } s = \perp \\ \{\, t \in W_{\mathcal{H}} \mid \langle s,t \rangle \in \rho \,\} & \text{otherwise} \end{cases}$$

ϕ is an onto lattice homomorphism which is also join and meet preserving.

The operator in $\langle \wp(V_{\mathcal{H}} W_{\mathcal{H}}), \subseteq \rangle$ which corresponds to sequential composition of state transformations in $\langle F_{\mathcal{H}}(V,W), \leq \rangle$ is relational composition, which is defined as follows. For relations ρ and σ and elements s and t:

$$\langle s,t \rangle \in (\rho \circ \sigma) \iff \exists s'.\, \langle s,s' \rangle \in \rho \wedge \langle s',t \rangle \in \sigma$$

The embedding ψ maps $(\cdot\,;\cdot)$ to $(\cdot \circ \cdot)$. From these remarks it is clear that we could have chosen a *relation on states* to represent the semantics of a program: instead we choose the more intuitive state transformations.

We next define a general recursion primitive which is sufficiently powerful to express all the usual recursive and iterative programming constructs. 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, so the result of the "fill in" is still incomplete (but "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 2.1 *Recursion:* Suppose we have a continuous function $\mathcal{F}$ (a function with bounded nondeterminism) 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:

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

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 easily shown to be 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 $(\mu.\mathcal{F})$.

2.5 Weakest Preconditions of Statements

We define the weakest precondition, $\mathrm{wp}(f, e)$ 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. So the weakest precondition is a function $\mathrm{wp}\colon F_{\mathcal{H}}(V, W)E_{\mathcal{H}}(W) \to E_{\mathcal{H}}(V)$ where $\mathrm{wp}(f, e) =_{\mathrm{DF}} \{ s \in V_{\mathcal{H}} \mid f(s) \subseteq e \}$. Note that since $\perp \in f(\perp)$ for any f, we have $f(\perp) \nsubseteq e$, so $\perp \notin \mathrm{wp}(f, e)$ for any f and e, so $\mathrm{wp}(f, e)$ is indeed in $E_{\mathcal{H}}(V)$.

The next theorem shows how refinement can be characterised using weakest preconditions:

Theorem 2.2 *For any state transformations $f_1, f_2 \in F_{\mathcal{H}}(V, W)$:*

$$f_1 \;\leq\; f_2 \iff \forall e \in E_{\mathcal{H}}(W).\, \mathrm{wp}(f_1, e) \subseteq \mathrm{wp}(f_2, e)$$

Proof: Assume $f_1 \;\leq\; f_2$ and let $e \in E_{\mathcal{H}}(W)$. Let $s \in \mathrm{wp}(f_1, e)$. Then $f_1(s) \subseteq e$. Therefore $f_2(s) \subseteq e$ by the definition of refinement. So $s \in \mathrm{wp}(f_2, e)$. But this is true for any $s \in \mathrm{wp}(f_1, e)$, hence $\mathrm{wp}(f_1, e) \subseteq \mathrm{wp}(f_2, e)$.

Conversely, suppose $\mathrm{wp}(f_1, e) \subseteq \mathrm{wp}(f_2, e)$ for any $e \in E_{\mathcal{H}}(W)$. Let $s \in V_{\mathcal{H}}$ be such that $\perp \notin f_1(s)$, then $f_1(s) \in E_{\mathcal{H}}(W)$. So put $e = f_1(s)$. Now, $s \in \mathrm{wp}(f_1, f_1(s))$ trivially, so $s \in \mathrm{wp}(f_2, f_1(s))$. Hence $f_2(s) \subseteq f_1(s)$. But this is true for every s such that $\perp \notin f(s)$. If $\perp \in f(s)$ then $f(s) = W_{\mathcal{H}}$ by definition, so $f_2(s) \subseteq f_1(s)$ trivially. So $f_1 \;\leq\; f_2$ as required. ■

The fact that refinement can be defined directly from the weakest precondition will later prove to be vitally important. A similar theorem to this one was used in [Bac80], our next result improves on this by showing that there is one particular postcondition which (together with the **true** postcondition) is sufficient to characterise refinement of state transformations. This allows us to eliminate the quantification over state conditions in Theorem 2.2. Later on, when we are interpreting formulae of first order logic as state conditions, this result will avoid the need for second order logic (i.e. quantification over formulae), or the "trick" of extending the logic with a new predicate symbol (which represents the unrestricted postcondition).

Theorem 2.3 *Let f_1, f_2 be any state transformations in $F_{\mathcal{H}}(V, W)$ and let $\mathbf{x}$ be any list of variables containing all the variables whose values may be changed by either of f_1 or f_2. Let $\mathbf{x}'$ be a list of new variables, of the same length as $\mathbf{x}$. We may assume, without loss of generality, that the variables in $\mathbf{x}'$ are in both V and W, and that f_1 and f_2 are independent of $\mathbf{x}'$ (i.e. the final value of $\mathbf{x}$ is not dependent on the initial value of $\mathbf{x}'$). Let e be the state condition in $E_{\mathcal{H}}(W)$ defined by:*

$$e = \{\, s \in E_{\mathcal{H}}(W) \mid s \neq \perp \;\wedge\; \forall x \in \tilde{\mathbf{x}}.\, s(x) \neq s(x') \,\}$$

so e is the interpretation of the formula $\mathbf{x} \neq \mathbf{x}'$. Let $e_{\mathbf{true}} = E_{\mathcal{H}}(W) \backslash \{\perp\}$ which is the interpretation of the formula **true**. *Then:*

$$f_1 \;\leq\; f_2 \iff \big(\mathrm{wp}(f_1, e) \subseteq \mathrm{wp}(f_2, e)\big) \;\wedge\; \big(\mathrm{wp}(f_1, e_{\mathbf{true}}) \subseteq \mathrm{wp}(f_2, e_{\mathbf{true}})\big)$$

Proof: The forward implication follows directly from Theorem 2.2, so suppose we have $\mathrm{wp}(f_1, e) \subseteq \mathrm{wp}(f_2, e)$ and $\mathrm{wp}(f_1, e_{\mathbf{true}}) \subseteq \mathrm{wp}(f_2, e_{\mathbf{true}})$.

- Let s be any element of $V_{\mathcal{H}}$ and t any element of $f_2(s)$, we need to prove $t \in f_1(s)$. If $t = \perp$ then $s \notin \mathrm{wp}(f_2, e_{\mathbf{true}})$ so $s \notin \mathrm{wp}(f_1, e_{\mathbf{true}})$ and $\perp \in f_1(s)$ as required;
- If $t \neq \perp$ then t assigns some values ($\mathbf{d}$ say) to the variables in $\mathbf{x}$, the values of all other variables in t must be the same as in s;
- Let t' and s' be formed from t and s by changing the values of $\mathbf{x}'$ to equal $\mathbf{d}$;
- Then since f_2 is independent of $\mathbf{x}'$ we must have $t' \in f_2(s')$. t' satisfies the interpretation of $\mathbf{x} = \mathbf{x}'$ so $t' \notin e$ so $f_2(s') \nsubseteq e$;
- This means that $s' \notin \mathrm{wp}(f_2, e)$ so from the premise we must have $s' \notin \mathrm{wp}(f_1, e)$;

- So there exists $t'' \in f_1(s')$ which is not in e;
- $t'' \neq \perp$ since $\perp \notin f_1(s)$ and s and s' only differ on $\mathbf{x}'$ and f_1 is independent of $\mathbf{x}'$;
- So t'' is a proper state which is not in e, i.e. $\mathbf{x} = \mathbf{x}'$ is interpreted as true in t'';
- This means t'' assigns values to $\mathbf{x}$ which match $\mathbf{x}'$. So these values must be the values $\mathbf{d}$;
- But these are the same values t' assigns to $\mathbf{x}$ and as f_1 and f_2 can only change the values of variables in $\mathbf{x}$ we must have $t' = t''$.

Thus $t' \in f_1(s')$ and as f_1 is independent of $\mathbf{x}'$ this means $t \in f_1(s)$ as required. ■

We will define a weakest precondition for *statements* (the WP discussed in Section 1.1) as a formula of infinitary logic. 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

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

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 [Bac80, Bac88] and Morgan [Mor94, MoR87] 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 Boolean predicates.

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

Definition 2.4 *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 $(\mu X.\mathbf{S})^0 = \textsf{\textbf{abort}}$ 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 $(\mu X.\mathbf{S})^n$. (In general, for statements $\mathbf{S}$, $\mathbf{T}$ and $\mathbf{T}'$, the notation $\mathbf{S}[\mathbf{T}'/\mathbf{T}]$ means "$\mathbf{S}$ with $\mathbf{T}'$ instead of each $\mathbf{T}$").

For the fundamental statements we have: $\mathrm{WP}(\textsf{\textbf{abort}}, \mathbf{R}) = \textsf{\textbf{false}}$, $\mathrm{WP}(\textsf{\textbf{skip}}, \mathbf{R}) = \mathbf{R}$ and $\mathrm{WP}(\textsf{\textbf{null}}, \mathbf{R}) = \textsf{\textbf{true}}$.

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

$$\begin{aligned}
\mathrm{WP}(\mathbf{x} := \mathbf{x}'.\mathbf{Q}, \mathbf{R}) &\iff \mathrm{WP}((\{\exists \mathbf{x}'.\, \mathbf{Q}\};\ (\mathsf{add}(\mathbf{x}');\ ([\mathbf{Q}];\ (\mathsf{add}(\mathbf{x});\ ([\mathbf{x} = \mathbf{x}'];\ \mathsf{remove}(\mathbf{x}')))))), \mathbf{R}) \\
&\iff \exists \mathbf{x}' \mathbf{Q} \wedge \forall \mathbf{x}'.\, (\mathbf{Q} \Rightarrow \forall \mathbf{x}.\, (\mathbf{x} = \mathbf{x}' \Rightarrow \mathbf{R})) \\
&\iff \exists \mathbf{x}' \mathbf{Q} \wedge \forall \mathbf{x}'.\, (\mathbf{Q} \Rightarrow \mathbf{R}[\mathbf{x}'/\mathbf{x}])
\end{aligned}$$

(recall that since the variables $\mathbf{x}'$ have been removed, they cannot occur free in $\mathbf{R}$).

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

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

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 the formula: $\mathsf{Pre} \wedge (\mathrm{WP}(\mathbf{S}, \textbf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}, \mathsf{Post}))$.

For the **if** statement discussed in Section 2.3:

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

Similarly, for the Dijkstra guarded command:

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

A fundamental result, first proved by Back [Bac80] for his kernel language, which also holds for our kernel language, is that these two forms of weakest precondition are equivalent:

Theorem 2.5 Let $\mathbf{S}$ be any statement and $\mathbf{R}$ be any postcondition for $\mathbf{S}$. If $\mathbf{S}$ is interpreted in some structure of $\mathcal{L}$ as the state transformation f and $\mathbf{R}$ as the state predicate e, then the interpretation of the formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ is the state predicate $\mathrm{wp}(f, e)$. Hence, the formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ captures the essential properties of $\mathbf{S}$ for refinement purposes. More formally:

$$\mathrm{int}_M(\mathrm{WP}(\mathbf{S}, \mathbf{R}), V) = \mathrm{wp}(\mathrm{int}_M(\mathbf{S}, V), \mathrm{int}_M(\mathbf{R}, W))$$

Proof: By induction on the structure of $\mathbf{S}$ and the depth of recursion nesting. ■

3 Proof Rules and Basic Transformations

Semantic refinement between statements is defined in the obvious way, as the refinement of the interpretations:

Definition 3.1 *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.

Proof theoretic refinement is defined from the weakest precondition formula WP, applied to a particular postcondition:

Definition 3.2 *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 $\mathrm{WP}(\mathbf{S}, \textbf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}', \textbf{true})$ are provable from the set Δ of sentences, then we say that $\mathbf{S}$ is refined by $\mathbf{S}'$ and write: $\Delta \vdash \mathbf{S} \leq \mathbf{S}'$.

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

Theorem 3.3 *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:*

$$\Delta \models \mathbf{S} \leq \mathbf{S}' \quad \textit{if and only if} \quad \Delta \vdash \mathbf{S} \leq \mathbf{S}'$$

Proof: Omitted ■

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 $\mathrm{WP}(\mathbf{S}, \textsf{true})$. 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 [Mor94] 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" ([Mor94], 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 $\mathcal{L}_{\omega_1\omega}$ rather than a higher-order logic, or indeed a larger infinitary logic. Back and von Wright [BaW90] 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 [Kar64] 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 $\textsf{true}$ are sufficient. Thus we can be confident that the proof method is both consistent and complete, in the sense that:

1. If $\big(\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}')\big) \;\wedge\; \big(\mathrm{WP}(\mathbf{S}_1, \textsf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \textsf{true})\big)$ 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 this completeness result.

Definition 3.4 *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 3.3 we have: $\Delta \models \mathbf{S} \;\approx\; \mathbf{S}'$ iff $\Delta \vdash \mathbf{S} \;\approx\; \mathbf{S}'$.

3.1 Expressing a Statement as a Specification

The formulae $\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}')$ and $\mathrm{WP}(\mathbf{S}, \textsf{true})$ 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 $\mathrm{WP}(\mathbf{S}, \textsf{false})$ (which is always **false** for null-free programs) then we can construct a specification statement equivalent to $\mathbf{S}$. Although this would seem to solve all reverse engineering problems at a stroke, and therefore be a great aid to software maintenance and reverse engineering, 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 [War95] 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 Maintainer's Assistant [Bul90, WaB93, WCM89] 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 Section 4.1 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 Section 4.2 to add arbitrary (countable) join and choice operators to the language, again without needing to extend the kernel language.

Theorem 3.5 *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:*

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

Proof: Without loss of generality we may assume $\tilde{\mathbf{x}} \subseteq V$. Let $V' = V \cup \tilde{\mathbf{x}}'$, let M be any model for Δ, and let $f = \mathrm{int}_M(\mathbf{S}, V)$ and $f' = \mathrm{int}_M(\mathbf{S}, V')$.

Now, f' is the program f with its initial and final state spaces extended to include $\mathbf{x}'$, hence if s is any state in $V_{\mathcal{H}}$ and $t \in f(s)$ and s' is any extension of the state s to $\mathbf{x}'$ then there exists $t' \in f'(s')$, the corresponding extension of t, with $t'(z) = s'(z)$ for every $z \in \tilde{\mathbf{x}}'$ and $t'(z) = t(z)$ for every $z \notin \tilde{\mathbf{x}}'$.

If $\mathbf{S}$ is null for some initial state then $\mathrm{WP}(\mathbf{S}, \textsf{\textbf{false}})$ is true for that state, hence the guard is false and the specification is also null. If $\mathbf{S}$ is undefined then $\mathrm{WP}(\mathbf{S}, \textsf{\textbf{true}})$ is false, and the specification statement is also undefined. So we only need to consider initial states for which $\mathbf{S}$ is both defined and non-null. Fix s as any element of $V_{\mathcal{H}}$ such that $f(s)$ is non-empty and $\perp \notin f(s)$. Then for any extension s' of s, $f'(s')$ is non-empty and $\perp \notin f'(s')$. Let:

$$g = \mathrm{int}_M(\textsf{add}(\mathbf{x}');\ [\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}')], V)$$

For each $t \in V_{\mathcal{H}}$ we define $s_t \in V'_{\mathcal{H}}$ to be the extension of s which assigns to $\mathbf{x}'$ the same values which t assigns to $\mathbf{x}$. We will prove the following Lemma:

Lemma 3.6 *For every* $t \in f(s)$ *there is a corresponding* $s_t \in g(s)$ *and every element of* $g(s)$ *is of the form* s_t *for some* $t \in f(s)$*.*

Proof: Let t be any element of $f(s)$. t gives a value to each variable in $\mathbf{x}$ and therefore can be used to define an extension $s_t \in V'_{\mathcal{H}}$ of s where the values assigned to $\mathbf{x}'$ by s_t are the same values which t assigns to $\mathbf{x}$. (The values given to any other variables are the same in s, t, and s_t since $\mathbf{S}$ can only affect the values of variables in $\mathbf{x}$.) Then we claim:

$$s_t \notin \mathrm{int}_M(\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'), V')$$

To prove this we note that a possible final state for f' on initial state s_t is the extension $t' \in V'_{\mathcal{H}}$ of the state t, where t' gives the same values to $\mathbf{x}'$ as s_t (the initial state). But these values are the same as the values t (and hence t') gives to $\mathbf{x}$, so t' does not satisfy the condition $\mathrm{int}_M(\mathbf{x} \neq \mathbf{x}', V')$. So not all final states for f' on initial state s_t satisfy the condition, so s_t does not satisfy $\mathrm{int}_M(\mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'), V')$. Hence:

$$s_t \in \mathrm{int}_M(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}'), V')$$

and hence $s_t \in g(s)$.

Conversely, every final state $t' \in g(s)$ leaves the values of $\mathbf{x}$ the same as in s and assigns values to $\mathbf{x}'$ such that $t' \notin \mathrm{wp}(f', \mathrm{int}_M(\mathbf{x} \neq \mathbf{x}', V'))$ is satisfied. For each such t' we can define a state $t \in V_{\mathcal{H}}$ which assigns to $\mathbf{x}$ the values t' assigns to $\mathbf{x}'$. Then $t' = s_t$ where s_t is as defined above. Suppose $t \notin f(s)$, then every terminal state in $f(s)$ must have values assigned to $\mathbf{x}$ which differ from those s_t assigns to $\mathbf{x}'$. But then every terminal state of $f'(s_t)$ would satisfy $\mathrm{int}_M(\mathbf{x} \neq \mathbf{x}', V')$ and hence s_t, and therefore t', would satisfy $\mathrm{wp}(f', \mathrm{int}_M(\mathbf{x} \neq \mathbf{x}', V'))$ which is a contradiction. So $t \in f(s)$ as required. This completes the proof of the Lemma. ■

To complete the main proof, we note that the state transformation $g' = \text{int}_M(\mathsf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'], W)$ maps each s_t to the set $\{t\}$. Hence $f(s) = (g;\ g')(s)$ and this holds for all initial states s on which $\mathbf{S}$ is defined and determinate. Hence $\mathbf{S}$ is equivalent to the given specification. ∎

For a general statement $\mathbf{S}\colon V \to W$ we have the corollary:

Corollary 3.7 *Let* $\mathbf{S}\colon V \to W$*, be any kernel language statement and let* $\mathbf{x}$ *be a list of all the variables assigned to by* $\mathbf{S}$*. Without loss of generality we may assume that* $W \subseteq V$ *(Any variables added by* $\mathbf{S}$ *are already in the initial state space). Let* $\mathbf{y}$ *be a list of the variables removed by* $\mathbf{S}$*, so* $\tilde{\mathbf{x}} \cap \tilde{\mathbf{y}} = \varnothing$*. Then for any countable set* Δ *of sentences:*

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

This theorem gives us an alternative representation for the weakest precondition of a statement:

Corollary 3.8 *For any statement* $\mathbf{S}$*:*

$$\begin{aligned}&\mathrm{WP}(\mathbf{S}, \mathbf{R}) \iff \\ &\quad \mathrm{WP}(\mathbf{S}, \mathsf{false}) \vee \big(\exists \mathbf{x}'.\, \neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \ \wedge\ \mathrm{WP}(\mathbf{S}, \mathsf{true}) \ \wedge\ \forall \mathbf{x}'.\, (\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathbf{R}[\mathbf{x}'/\mathbf{x}])\big)\end{aligned} \tag{1}$$

where $\mathbf{x}$ is the variables assigned to by $\mathbf{S}$ as above.

Proof: Convert $\mathbf{S}$ to its specification equivalent using Theorem 3.5, take the weakest precondition for $\mathbf{R}$ and simplify the result. ∎

This corollary is useful in that it expresses the weakest precondition of a statement for any postcondition as a simple formula involving the postcondition itself and weakest preconditions of fixed formulae.

3.2 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 3.12).

Lemma 3.9 *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 3.10 *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 (including nested recursion). For any statement $\mathbf{S}$, define $\mathbf{S}^n$ to be $\mathbf{S}$ with each recursive statement replaced by its nth truncation.

Lemma 3.11 *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 $\Delta \vdash \mathbf{S} \ \leq\ \mathbf{S}'$.

Proof: Omitted. ■

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 [War89] 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 3.12 *Fold/Unfold:* For any $\mathbf{S}\colon V \to V$:

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

Proof:

$$\begin{aligned}
\mathrm{WP}((\mu X.\mathbf{S}), \mathbf{R}) &\iff \bigvee_{n<\omega} \mathrm{WP}((\mu X.\mathbf{S})^n, \mathbf{R}) \\
&\iff \bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}[(\mu X.\mathbf{S}^{n-1})/X], \mathbf{R}) \\
&\iff \bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}[(\mu X.\mathbf{S}^{n})/X], \mathbf{R}) \\
&\iff \mathrm{WP}(\mathbf{S}[(\mu X.\mathbf{S})/X], \mathbf{R})
\end{aligned}$$

by the general induction rule for recursion. ■

3.3 Proving Termination of Recursive Statements

In this section we prove some important theorems which show how the termination of a recursive statement can be proved, with the aid of a well-founded order relation on the state. First we have a lemma on the maintenance of invariants:

Lemma 3.13 *Invariant Maintenance*

(i) If for any statement $\mathbf{S}_1$ we can prove: $\Delta \vdash \{\mathbf{P}\};\ \mathbf{S}[\mathbf{S}_1/X] \ \leq\ \mathbf{S}[\{\mathbf{P}\};\ \mathbf{S}_1/X]$ then $\Delta \vdash \{\mathbf{P}\};\ (\mu X.\mathbf{S}) \ \leq\ (\mu X.\{\mathbf{P}\};\ \mathbf{S})$

(ii) If in addition $\Delta \vdash \{\mathbf{P}\};\ \mathbf{S}_1 \ \leq\ \mathbf{S}_1;\ \{\mathbf{P}\}$ implies $\Delta \vdash \{\mathbf{P}\};\ \mathbf{S}[\mathbf{S}_1/X] \ \leq\ \mathbf{S}[\mathbf{S}_1/X];\ \{\mathbf{P}\}$ then $\Delta \vdash \{\mathbf{P}\};\ (\mu X.\mathbf{S}) \ \leq\ (\mu X.\mathbf{S});\ \{\mathbf{P}\}$.

Our next theorem will prove that if we can find a well-founded order on some part of the state, and a term $\mathbf{t}$ which is reduced under this order before each recursive call, then the recursive procedure will terminate. This generalises the termination rule in [Dij76] which uses the usual order on the natural numbers and is restricted to deterministic programs. (This generalisation has been discovered by several authors independently). By generalising this to *any* well-founded order we can simplify termination proofs: for example a lexical order for a bounded sequence of positive integer state variables or array elements is well-founded, but not order-isomorphic to the natural numbers.

Definition 3.14 A partial order $\preccurlyeq$ on some set Γ is *well-founded* if every non-empty subset of Γ has a minimal element under $\preccurlyeq$, i.e.:

$$\forall \Gamma' \subseteq \Gamma.\,(\Gamma' \neq \varnothing \Rightarrow \exists \zeta \in \Gamma'.\,\forall \lambda \in \Gamma'.\,(\zeta \preccurlyeq \lambda))$$

We write $\zeta \prec \lambda$ if $(\zeta \preccurlyeq \lambda \ \wedge\ \zeta \neq \lambda)$. We also define minimal$(\zeta)$ to mean "ζ is a minimal element in Γ", i.e.:

$$\text{minimal}(\zeta) \ =_{\text{DF}}\ \forall \zeta' \in \Gamma.\, (\zeta' \preccurlyeq \zeta \Rightarrow \zeta' = \zeta)$$

Theorem 3.15 *Proving Termination:* 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

$$\forall t_0.\, ((\mathbf{P} \ \wedge\ \mathbf{t} \preccurlyeq t_0) \Rightarrow \text{WP}(\mathbf{S}[\{\mathbf{P} \ \wedge\ \mathbf{t} \prec t_0\}/X], \textsf{\textbf{true}})) \tag{2}$$

then $\mathbf{P} \Rightarrow \text{WP}((\mu X.\mathbf{S}), \textsf{\textbf{true}})$.

Proof: Omitted. ■

Let $t_{\min}$ be the least element of $\{\, t_0 \in \Gamma \mid \text{WP}(\mathbf{S}[\{\mathbf{t} \prec t_0\}/X], \textsf{\textbf{true}}) \,\}$ if this set is non-empty. If $t_{\min}$ exists and $t_{\min} \preccurlyeq \mathbf{t}$ initially then any calls of X in the execution of $\mathbf{S}$ must be started in a state in which $\mathbf{t}$ is smaller than in the initial state, because the smallest bound greater than any possible value of $\mathbf{t}$ at a call of X is less than or equal to the initial value of $\mathbf{t}$.

We write the predicate $t_{\min} \preccurlyeq \mathbf{t}$ as $\text{Wdec}_X(\mathbf{S}, \mathbf{t})$, it is false if there is no t_0 such that $\text{WP}(\mathbf{S}[\{\mathbf{t} \prec t_0\}/X], \textsf{\textbf{true}})$ holds, so we define Wdec as follows:

$$\begin{aligned}\text{Wdec}_X(\mathbf{S}, \mathbf{t}) \ =_{\text{DF}}\ & \exists t_0.\, \text{WP}(\mathbf{S}[\{\mathbf{t} \prec t_0\}/X].\textsf{\textbf{true}}) \\ & \quad \wedge\ \forall t_{\min}.\, \big((\text{WP}(\mathbf{S}[\{\mathbf{t} \prec t_{\min}\}/X], \textsf{\textbf{true}}) \\ & \qquad\qquad \wedge\ \forall t_0.\, (\text{WP}(\mathbf{S}[\{\mathbf{t} \prec t_0\}/X], \textsf{\textbf{true}}) \Rightarrow t_{\min} \preccurlyeq t_0)\big) \Rightarrow t_{\min} \preccurlyeq \mathbf{t}\big)\end{aligned}$$

Theorem 3.16 With this definition of Wdec we have:

$$\forall t_0.\, (\mathbf{t} \preccurlyeq t_0 \Rightarrow \text{WP}(\mathbf{S}[\{\mathbf{t} \prec t_0\}/X], \textsf{\textbf{true}})) \iff \text{Wdec}_X(\mathbf{S}, \mathbf{t}) \tag{3}$$

Putting these results together we have:

Corollary 3.17 *Weakest preconditions of recursive procedures. If*

(i) $\mathbf{P} \Rightarrow \text{WP}(\mathbf{S}[\{\mathbf{P}\}/X], \textsf{\textbf{true}})$ *and*

(ii) *For any* $\mathbf{S}_1$*:* $\Delta \vdash \{\mathbf{P}\};\ \mathbf{S}_1 \ \leq\ \mathbf{S}_1;\ \{\mathbf{P}\} \implies \Delta \vdash \{\mathbf{P}\};\ \mathbf{S}[\mathbf{S}_1/X] \ \leq\ \mathbf{S}[\mathbf{S}_1/X];\ \{\mathbf{P}\}$ *and*

(iii) *For any* $\mathbf{S}_1$*:* $\Delta \vdash \{\mathbf{P}\};\ \mathbf{S}[\mathbf{S}_1/X] \ \leq\ \mathbf{S}[\{\mathbf{P}\};\ \mathbf{S}_1/X]$ *and*

(iv) *For some term* $\mathbf{t}$ *taking values in* Γ *on which* $\preccurlyeq$ *is well-founded:* $\mathbf{P} \Rightarrow \text{Wdec}_X(\mathbf{S}, \mathbf{t})$

then $\mathbf{P} \Rightarrow \text{WP}((\mu X.\mathbf{S}), \mathbf{P})$*.*

The second premise states that if all recursive calls of $\mathbf{S}$ in $(\mu X.\mathbf{S})$ were replaced by any statement which preserved $\mathbf{P}$ then the statement so formed would preserve $\mathbf{P}$.

3.4 Proof Rules for Implementations

In this section we will develop two general proof rules. The first is for proving the correctness of an potential implementation $\mathbf{S}$, 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. Note that theorem is also useful in deriving *iterative* implementations of specifications, since very often the most convenient derivation is via a recursive formulation. In [War89, War91a, War92, War99] techniques are presented for transforming recursive procedures into various iterative forms. See also Section 5.1.

3.4.1 Implementation of Specifications

The first proof rule is based on a proof rule in Back [Bac80], we have extended this to include recursion and guard statements. This proof rule provides a means of proving that a statement $\mathbf{S}$ is a correct implementation of a specification $\{\mathbf{P}\};\ \mathbf{x} := \mathbf{x}'.\mathbf{Q}$. Note that, for example, any **Z** specification can be cast into this form.

Theorem 3.18 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 $\mathbf{S}$. Let $\mathbf{x}_0$, $\mathbf{y}_0$ be lists of distinct variables not in $\mathbf{S}$ 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 $\mathbf{S}$ 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 $\mathbf{Q}$. This was proved in [Bac80] for iterative statements, the extension to include recursive statements and guards is straightforward.

Corollary 3.19 *By the same assumptions as above we have:*

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

Corollary 3.20 *For the assignment $x := t$ we have:*

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

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 [War89, War90, War91a, War92, War99, War96].

3.4.2 Recursive Implementation of General Statements

In this section we present 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.

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 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 $\preccurlyeq$ 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 3.21 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

$$\Delta \vdash \{\mathbf{P} \wedge \mathbf{t} \preccurlyeq t_0\};\ \mathbf{S}' \ \leq\ \mathbf{S}[\{\mathbf{P} \wedge \mathbf{t} \prec t_0\};\ \mathbf{S}'/X]) \tag{4}$$

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

Proof: Omitted. ■

3.4.3 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 get:

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

Separate the assignment:

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

So we have:

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

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:

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

So we can apply Theorem 3.21 to get:

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

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.

This approach to algorithm derivation should be contrasted with the **Z** refinement approach which consists of these steps [McN89]:

1. Postulate a possible refinement for the specification;
2. Determine the proof obligations (these are the theorems which must be proved in order to prove that the proposed refinement is valid);
3. Attempt to discharge the proof obligations, perhaps with the aid of a mechanical proof checker.

If an error is made in step (1), the step (3) will of course fail, but in practice few if any proof obligations are discharged. Despite the increasing availability of automatic theorem provers, the proof for a large program is still a major activity, Sennett in [Sen90] indicates that for "real" sized programs it is impractical to discharge more than a tiny fraction of the proof obligations. He presents a case study of the development of a simple algorithm, for which the implementation of one function gave rise to over one hundred theorems which required proofs. Larger programs will require many more proofs. In practice, since few if any of these proofs will be rigorously carried out, what claims to be a formal method for program development turns out to be a formal method for program specification, together with an *informal* development method. In contrast, using our approach the developer is always working with a guaranteed correct implementation of the specification. With the aid of a suitable transformation system, even the applicability conditions of the transformations can be mechanically checked.

4 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 prototype transformation system which currently implements over six hundred transformations, accessible through a simple user interface [Bul90]. More recently the prototype has been completely redeveloped into a practical transformation system with fewer, but more powerful, transformations [WaB93, WaB95]

For the last twelve 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. Specification statement: This was discussed in Section 2:

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

2. 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 $\langle x \rangle := \langle t \rangle$;

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

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

4. Deterministic Choice: We can use guards to turn a nondeterministic choice into a deterministic choice as discussed in Section 2.3:

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

5. Nondeterministic Choice: The "guarded command" of Dijkstra [Dij76]:

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

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

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

7. Nondeterministic Iteration:

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

8. Initialised Local Variables:

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

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

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

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 $\langle\rangle\colon [\textbf{true}, \textbf{false}]$ is equivalent to [**false**] which is everywhere null). So all statements in the first level language, with no explicit guard statements, satisfy Dijkstra's "Law of the Excluded Miracle" [Dij76]: $\mathrm{WP}(\mathbf{S}, \textbf{false}) \iff \textbf{false}$.

The second level language introduces multi-**exit** loops and Action systems (cf [Ars82, Ars79]). The third level adds local variables and parameters to procedures, functions and expressions with side effects. See [War89] for the definitions of these language levels.

4.1 Expressing JOIN in terms of the First Level Language

Recall that the **join** of two programs is the weakest program which meets all specifications satisfied by either component. We can use the Representation Theorem (Theorem 3.5) to define a **join** operator on statements which have identical initial and final state spaces. We first define the **join** of two specification statements, then we can translate the two given statements into equivalent specifications and **join** them together.

The **join** of the specification statements $\mathbf{x} := \mathbf{x}'.\mathbf{Q}_1$ and $\mathbf{x} := \mathbf{x}'.\mathbf{Q}_2$ is defined:

$$\begin{aligned} \textbf{join}\ \mathbf{x} := \mathbf{x}'.\mathbf{Q}_1 \sqcup \mathbf{x} := \mathbf{x}'.\mathbf{Q}_2\ \textbf{nioj}\ =_{\mathrm{DF}}\ & \textbf{if}\ \neg\exists \mathbf{x}'.\, \mathbf{Q}_1 \to \mathbf{x} := \mathbf{x}'.\mathbf{Q}_2 \\ & \square\ \neg\exists \mathbf{x}'.\, \mathbf{Q}_2 \to \mathbf{x} := \mathbf{x}'.\mathbf{Q}_1 \\ & \square\ \exists \mathbf{x}'.\, \mathbf{Q}_1 \wedge \exists \mathbf{x}'.\, \mathbf{Q}_2 \to [\exists \mathbf{x}'.\, (\mathbf{Q}_1 \wedge \mathbf{Q}_2)]; \\ & \qquad\qquad \mathbf{x} := \mathbf{x}'.(\mathbf{Q}_1 \wedge \mathbf{Q}_2)\ \textbf{fi} \end{aligned}$$

Definition 4.1 *The* **join** *construct:* Suppose $\mathbf{S}_1, \mathbf{S}_2 \colon V \to W$ are two statements and $\mathbf{x}$ is a list of all the variables assigned to by either statement and $\mathbf{y}$ a list of the variable removed from the state space (i.e. $\tilde{\mathbf{y}} = V - W$ and $\tilde{\mathbf{x}} \subseteq W$). Note that $\mathbf{x}$ and $\mathbf{y}$ are independent of the particular choice for V and W, provided $\mathbf{S}_1, \mathbf{S}_2 \colon V \to W$. Let $\mathbf{x}'$ be a list of new variables, the same length as $\mathbf{x}$. Then the **join** of $\mathbf{S}_1$ and $\mathbf{S}_2$ is defined:

$$\begin{aligned} \textbf{join}\ \mathbf{S}_1 \sqcup \mathbf{S}_2\ \textbf{nioj}\ =_{\mathrm{DF}}\ & \textbf{if}\ \neg\mathrm{WP}(\mathbf{S}_1, \textbf{true}) \to \mathbf{S}_2 \\ & \square\ \neg\mathrm{WP}(\mathbf{S}_2, \textbf{true}) \to \mathbf{S}_1 \\ & \square\ \mathrm{WP}(\mathbf{S}_1, \textbf{true}) \wedge \mathrm{WP}(\mathbf{S}_2, \textbf{true}) \\ & \quad \to [\exists \mathbf{x}'.\, (\neg\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \wedge \neg\mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}'))]; \\ & \qquad \mathbf{x} := \mathbf{x}'.(\neg\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \wedge \neg\mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}')); \\ & \qquad \mathsf{remove}(\mathbf{y})\ \textbf{fi} \end{aligned}$$

From this we can calculate the weakest precondition of a **join** construct:

$$\begin{aligned} & \mathrm{WP}(\textbf{join}\ \mathbf{S}_1 \sqcup \mathbf{S}_2\ \textbf{nioj}, \mathbf{R}) \\ & \quad \iff (\mathrm{WP}(\mathbf{S}_1, \textbf{true}) \vee \mathrm{WP}(\mathbf{S}_2, \mathbf{R})) \wedge (\mathrm{WP}(\mathbf{S}_2, \textbf{true}) \vee \mathrm{WP}(\mathbf{S}_1, \mathbf{R})) \wedge \\ & \qquad\quad \mathrm{WP}((\mathbf{S}_1 \sqcap \mathbf{S}_2), \textbf{true}) \Rightarrow \forall \mathbf{x}'.\, (\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \vee \mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}') \vee \mathbf{R}[\mathbf{x}'/\mathbf{x}]) \end{aligned}$$

The effect of joining two statements (or specifications) is to produce a statement which combines all the properties of both components. For example, if $\mathbf{S}_1$ is $\{\mathrm{even}(x)\};\ x := x/2$ and $\mathbf{S}_2$ is $\{\mathrm{odd}(x)\};\ x := x+1$ then **join** $\mathbf{S}_1 \sqcup \mathbf{S}_2$ **nioj** is **if** $\mathrm{even}(x)$ **then** $x := x/2$ **else** $x := x+1$ **fi**. Compare this with the definition of the **if** statement using guards (Section 2.3).

If $\mathbf{S}_1$ is $x := x'.(2|x)$ which sets x to any multiple of 2, and $\mathbf{S}_2$ is $x := x'.(3|x)$ which sets x to any multiple of 3, then **join** $\mathbf{S}_1 \sqcup \mathbf{S}_2$ **nioj** is $x := x'.(6|x)$ which sets x to any multiple of 6. If the two **join**ed statements are inconsistent (e.g. if they have no final states in common, for example $x := 1$ and $x := 2$) then their join will be the **null** statement [**false**].

We have chosen to define **join** in terms of a guard and other statements. We could just as easily have chosen to make **join** a primitive construct and defined a guard statement using **join** thus:

$$[\mathbf{Q}]\ =_{\mathrm{DF}}\ \textbf{if Q then skip else begin}\ x\colon\ \textbf{join}\ x := 1 \sqcup x := 2\ \textbf{nioj end fi}$$

where x is an otherwise unused variable and 1 and 2 are constant symbols representing distinct values in $\mathcal{H}$.

We chose to make the guard statement primitive because its WP is so much simpler than the WP of the **join** statement.

The following theorem expresses an important (if not the *most* important) property of the **join** construct: if a statement $\mathbf{S}$ is a refinement of both $\mathbf{S}_1$ and $\mathbf{S}_2$ then it is a refinement of the join **join** $\mathbf{S}_1 \sqcup \mathbf{S}_2$ **nioj**. This is important because **join** is useful for expressing specifications but is not directly implementable. So transformations which eliminate **join**s are valuable practical tools for program derivation.

Theorem 4.2 *If* $\Delta \vdash \mathbf{S}_1 \ \leq\ \mathbf{S}$ *and* $\Delta \vdash \mathbf{S}_2 \ \leq\ \mathbf{S}$ *then* $\Delta \vdash \textsf{join}\ \mathbf{S}_1 \sqcup \mathbf{S}_2\ \textsf{nioj} \ \ \leq\ \ \mathbf{S}$.

4.2 Arbitrary join and choice operators

The definitions of $\bigsqcup F$ and $\bigsqcap F$ for any set of state transformations $F \subseteq F_{\mathcal{H}}(V, W)$ in section 2.4 make $F_{\mathcal{H}}(V, W)$ into a complete lattice. For statements the representation theorem (Theorem 3.5) can be used to express the join and choice of any *countable* sequence of statements. For the nondeterministic choice of the countable sequence of statements $\langle \mathbf{S}_n \mid n < \omega \rangle$ we have:

$$\mathrm{WP}(\bigsqcap_{n<\omega} \mathbf{S}_n, \mathbf{R}) = \bigwedge_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \mathbf{R})$$

From this we use the representation theorem to define:

$$\begin{aligned}\bigsqcap_{n<\omega} \mathbf{S}_n \ =_{\mathrm{DF}}\ & [\neg \textstyle\bigwedge_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \textsf{false})];\\ & \mathbf{x} := \mathbf{x}'.(\neg \textstyle\bigwedge_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \mathbf{x} \neq \mathbf{x}') \ \wedge\ \textstyle\bigwedge_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \textsf{true}));\\ & \textsf{remove}(\mathbf{y})\end{aligned}$$

For the countable join $\bigsqcup_{n<\omega} \mathbf{S}_n$ we have the following weakest precondition:

$$\mathrm{WP}(\bigsqcup_{n<\omega} \mathbf{S}_n, \mathbf{R}) = \forall \mathbf{x}'.\, \big(\bigwedge_{n<\omega}(\mathrm{WP}(\mathbf{S}_n, \textsf{true}) \Rightarrow \neg \mathrm{WP}(\mathbf{S}_n, \mathbf{x} \neq \mathbf{x}')) \Rightarrow \mathbf{R}[\mathbf{x}'/\mathbf{x}]\big) \ \wedge\ \bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \textsf{true})$$

From this we see that:

$$\begin{aligned}\mathrm{WP}(\bigsqcup_{n<\omega} \mathbf{S}_n, \textsf{false}) \ &\Longleftrightarrow\ \forall \mathbf{x}'.\ \bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \mathbf{x} \neq \mathbf{x}') \ \wedge\ \bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \textsf{true})\\ \mathrm{WP}(\bigsqcup_{n<\omega} \mathbf{S}_n, \textsf{true}) \ &\Longleftrightarrow\ \bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \textsf{true})\end{aligned}$$

and

$$\mathrm{WP}(\bigsqcup_{n<\omega} \mathbf{S}_n, \mathbf{x} \neq \mathbf{x}') \ \Longleftrightarrow\ \bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \mathbf{x} \neq \mathbf{x}')$$

So using the representation theorem we can define:

$$\begin{aligned}\bigsqcup_{n<\omega} \mathbf{S}_n \ =_{\mathrm{DF}}\ & [\forall \mathbf{x}'.\ \textstyle\bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \mathbf{x} \neq \mathbf{x}') \ \wedge\ \textstyle\bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \textsf{true})];\\ & \mathbf{x} := \mathbf{x}'.(\neg \textstyle\bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \mathbf{x} \neq \mathbf{x}') \ \wedge\ \textstyle\bigvee_{n<\omega} \mathrm{WP}(\mathbf{S}_n, \textsf{true}));\\ & \textsf{remove}(\mathbf{y})\end{aligned}$$

This result means that we could simplify the kernel language even further by removing the recursive statement and replacing it by a definitional transformation of the form:

$$\begin{aligned}(\mu X.\mathbf{S}) \ =_{\mathrm{DF}}\ & [\forall \mathbf{x}'.\ \textstyle\bigvee_{n<\omega} \mathrm{WP}((\mu X.\mathbf{S})^n, \mathbf{x} \neq \mathbf{x}') \ \wedge\ \textstyle\bigvee_{n<\omega} \mathrm{WP}((\mu X.\mathbf{S})^n, \textsf{true})];\\ & \mathbf{x} := \mathbf{x}'.(\neg \textstyle\bigvee_{n<\omega} \mathrm{WP}((\mu X.\mathbf{S})^n, \mathbf{x} \neq \mathbf{x}') \ \wedge\ \textstyle\bigvee_{n<\omega} \mathrm{WP}((\mu X.\mathbf{S})^n, \textsf{true}))\end{aligned}$$

where $(\mu X.\mathbf{S})^n$ is the usual truncated recursion.

For nondeterministic choice, the representation theorem provides (as a special case of countable choice) the definitional transformation:

$$\begin{aligned}(\mathbf{S}_1 \ \sqcap\ \mathbf{S}_2) \ =_{\mathrm{DF}}\ & [\neg \mathrm{WP}(\mathbf{S}_1, \textsf{false}) \ \vee\ \neg \mathrm{WP}(\mathbf{S}_2, \textsf{false})];\\ & \mathbf{x} := \mathbf{x}'.((\neg \mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \ \vee\ \neg \mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}'))\\ & \qquad\qquad \wedge\ \mathrm{WP}(\mathbf{S}_1, \textsf{true}) \ \wedge\ \mathrm{WP}(\mathbf{S}_2, \textsf{true}));\\ & \textsf{remove}(\mathbf{y})\end{aligned}$$

This means that sequencing is the *only* compound construct which is actually essential in the kernel language. All other constructs can be represented as finite sequences of atomic statements. In practice, we have included choice and recursion in the kernel (but not join) since their weakest preconditions are simple and easy to work with, while their representations as sequences are rather cumbersome.

Theorem 4.3 *The sub-language of the kernel language, consisting of assert, guard, add, remove and sequencing, is equivalent to the full kernel language.*

4.3 Angelic Choice

The method used in the previous section for defining new statement constructs depends on the representation theorem, which in turn depends on the existence of a state transformation which correctly interprets the semantics of the new construct. A construct for which this method fails is *angelic choice* (also called *favourable choice*) which is a duel to the nondeterministic choice (or unfavourable choice), so called because the choice is made so as to satisfy the required postcondition, if this is possible. The weakest precondition of the angelic choice $\mathbf{S}_1 \oplus \mathbf{S}_2$ is defined as:

$$\mathrm{WP}(\mathbf{S}_1 \oplus \mathbf{S}_2, \mathbf{R}) =_{\mathrm{DF}} \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \vee \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$$

For example, $(x := 1 \oplus x := 2)$ assigns either 1 or 2 to x depending on which value will be required later on in the program. For instance: $(x := 1 \oplus x := 2)$; $\{x = 1\}$ is equivalent to $x := 1$ while $(x := 1 \oplus x := 2)$; $\{x = 2\}$ is equivalent to $x := 2$.

If we try to use the representation theorem to define a statement equivalent to $(x := 1 \oplus x := 2)$ (where 1 and 2 are distinct) then the result reduces to **abort**. This is because there is no state transformation which captures the semantics of $(x := 1 \oplus x := 2)$ since there is *no* state transformation whose weakest precondition is $\mathrm{WP}(x := 1, \mathbf{R}) \vee \mathrm{WP}(x := 2, \mathbf{R})$ so the representation theorem is not valid for this construct. Such extensions to the language require a more complicated semantic model.

5 Reverse Engineering Using Transformations

In [War89, War91b, War92, War99] the Wide Spectrum Language and transformation theory are further developed to include:

1. **Unbounded 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 [Knu74] and more recently in [Tay84];
2. **Action systems:** An *action* is a parameterless procedure acting on global variables (cf [Ars82, Ars79]). 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*. There may sometimes be a special action Z, execution of which causes termination of the whole action system even if there are unfinished recursive calls. An occurrence of a statement **call** X within the action body refers to a call of another action;
3. **Procedures and functions:** These may be mutually recursive and include parameters and local variables.

In this section we will give an example of a powerful transformation which is useful for both forward and reverse engineering. The transformation proves the equivalence of certain recursive and iterative programs, expressed in a very general manner. In the forward direction, we use it in the algorithm derivations of [War92, War96] and in the reverse direction we use it in [War94a, War96].

5.1 A Recursion Removal/Introduction Theorem

Theorem 5.1 Suppose we have a recursive procedure whose body is an action system in the following form (where a **call** Z in the action system will terminate only the current invocation of the procedure):

proc $F(x) \ \equiv$
  actions A_1 :
  $A_1 \ \equiv \mathbf{S}_1$**.**
  $\ldots A_i \ \equiv \mathbf{S}_i$**.**
  $\ldots B_j \ \equiv \mathbf{S}_{j0};\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\ F(g_{jn_j}(x));\ \mathbf{S}_{jn_j}$**.**
  $\ldots$ **endactions.**

where $\mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j}$ preserve the value of x and no $\mathbf{S}$ contains a call to F (i.e. all the calls to F are listed explicitly in the B_j actions) and only the statements $\mathbf{S}_i$ and $\mathbf{S}_{jn_j}$ may contain action calls. There are $M + N$ actions in total: $A_1, \ldots, A_M, B_1, \ldots, B_N$.

We claim that this is equivalent to the following iterative procedure which uses a new local stack L and a new local variable m:

proc $F'(x) \ \equiv$
  var $\langle L := \langle\rangle, m := 0\rangle$:
    actions A_1 :
    $A_1 \ \equiv \mathbf{S}_1[$**call** $\hat{F}/$**call** $Z]$**.**
    $\ldots A_i \ \equiv \mathbf{S}_i[$**call** $\hat{F}/$**call** $Z]$**.**
    $\ldots B_j \ \equiv \mathbf{S}_{j0};\ L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j,1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;$
        call $\hat{F}$**.**
    $\ldots \hat{F} \ \equiv$ **if** $L = \langle\rangle$ **then call** Z
        else $\langle m, x\rangle \leftarrow L;$
          if $m = 0 \rightarrow$ **call** A_1
          $\square \ldots \square\ m = \langle j, k\rangle \rightarrow \mathbf{S}_{jk};$ **call** $\hat{F}$
          $\ldots$ **fi fi. endactions end.**

The proof is rather involved and too long to include here. It relies on applying various transformations which have been proved using weakest preconditions, together with multiple applications of Lemma 3.11. By unfolding some calls to $\hat{F}$ and pruning, we get the following, slightly more efficient, version:

proc $F'(x) \ \equiv$
  var $\langle L := \langle\rangle, m := 0\rangle$:
    actions A_1 :
    $A_1 \ \equiv \mathbf{S}_1[$**call** $\hat{F}/$**call** $Z]$**.**
    $\ldots A_i \ \equiv \mathbf{S}_i[$**call** $\hat{F}/$**call** $Z]$**.**
    $\ldots B_j \ \equiv \mathbf{S}_{j0};\ L := \langle\langle\langle j,1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;$
        $x := g_{j1}(x);$ **call** A_1**.**
    $\ldots \hat{F} \ \equiv$ **if** $L = \langle\rangle$ **then call** Z
        else $\langle m, x\rangle \leftarrow L;$
          if $m = 0 \rightarrow$ **call** A_1
          $\square \ldots \square\ m = \langle j, k\rangle \rightarrow \mathbf{S}_{jk};$ **call** $\hat{F}$
          $\ldots$ **fi fi. endactions end.**

5.2 An Example of Inverse Engineering

Consider the following program which takes four positive integer arrays l, r, c and m and a positive integer x and modifies some of the elements of m, where m is assumed to be initially an array of zeros:

var $\langle L := \langle\rangle, d := 0\rangle$:
  do do if $x \neq 0$
      then if $m[l[x]] = 0$
          then $L \xleftarrow{\text{push}} \langle 1, x\rangle;\ x := l[x]$
          else exit fi

else do if $L = \langle\rangle$ **then exit**(3) **fi**;
$\langle d, x\rangle \stackrel{\text{pop}}{\longleftarrow} L$;
if $d = 0 \rightarrow$ **exit**(2)
$\square\ d = 1 \rightarrow$ **exit**
$\square\ d = 2 \rightarrow$ **skip fi od fi**;
do $m[x] := 1$;
if $m[c[x]] = 0 \ \wedge\ m[r[x]] = 1$
then $L \stackrel{\text{push}}{\longleftarrow} \langle 2, x\rangle;\ x := c[x];$ **exit**(2)
elsif $m[c[x]] = 1 \ \wedge\ m[r[x]] = 0$
then $L \stackrel{\text{push}}{\longleftarrow} \langle 2, x\rangle;\ x := r[x];$ **exit**(2)
elsif $m[c[x]] = 0 \ \wedge\ m[r[x]] = 0$
then $L \stackrel{\text{push}}{\longleftarrow} \langle 2, x\rangle;\ L \stackrel{\text{push}}{\longleftarrow} \langle 0, r[x]\rangle;\ x := c[x];$ **exit**(2)
else do if $L = \langle\rangle$ **then exit**(4) **fi**;
$\langle d, x\rangle \stackrel{\text{pop}}{\longleftarrow} L$;
if $d = 0 \rightarrow$ **exit**(3)
$\square\ d = 1 \rightarrow$ **exit**
$\square\ d = 2 \rightarrow$ **skip fi od fi od od od end**

Despite its small size, this program has a fairly complex control structure, with a quadruple-nested loop and exits which terminate from the middle of one, two, three and four nested loops. Finding suitable invariants for all the loops seems to be a difficult task, while finding a variant function is impossible! For suppose $x \neq 0$, $m[l[x]] = 0$ and $l[x] = x$ initially. With this initial state, it is easy to see that the program will never terminate. Therefore there is *no* variant function which works over the whole initial state space. To determine the conditions under which the program terminates, basically involves determining the behaviour of the entire program: not a helpful requirement for the first stage of a reverse engineering task!

There are certain features which suggest that the recursion removal theorem might be usefully applied: in particular the presence of a local array which is used as a stack and which starts empty and finishes empty. One problem is that there are two places in the program where L is tested and an element popped off. However, if we restructure the program as an action system, then there are some powerful transformations which can be used to merge the two actions which test L and convert the program into the right structure for the recursion introduction theorem.

The first step therefore is to restructure the program as an action system. This basically involves implementing the loops as action calls (i.e. **goto**s)—an unusual step in a reverse engineering process!

var $\langle L := \langle\rangle, d := 0\rangle$:
proc $F(x)\ \equiv$
actions A_1 :
$A_1\ \equiv$ **if** $x \neq 0$ **then if** $m[l[x]] = 0$ **then call** B_1
else call A_2 **fi**
else call $\hat{F}_1$ **fi.**
$A_2\ \equiv m[x] := 1$;
if $m[c[x]] = 0 \ \wedge\ m[r[x]] = 1$
then call B_2
elsif $m[c[x]] = 1 \ \wedge\ m[r[x]] = 0$
then call B_3
elsif $m[c[x]] = 0 \ \wedge\ m[r[x]] = 0$
then call B_4
else call $\hat{F}_2$ **fi.**
$B_1\ \equiv L \stackrel{\text{push}}{\longleftarrow} \langle 1, x\rangle;\ x := l[x];$ **call** A_1**.**

$B_2 \equiv L \xleftarrow{\text{push}} \langle 2, x\rangle;\ x := c[x];$ **call** A_1**.**
$B_3 \equiv L \xleftarrow{\text{push}} \langle 2, x\rangle;\ x := r[x];$ **call** A_1**.**
$B_4 \equiv L \xleftarrow{\text{push}} \langle 2, x\rangle;\ L \xleftarrow{\text{push}} \langle 0, r[x]\rangle;\ x := c[x];$ **call** A_1**.**
$\hat{F}_1 \equiv$ **if** $L = \langle\rangle$ **then call** Z **fi**;
$\langle d, x\rangle \xleftarrow{\text{pop}} L;$
if $d = 0 \rightarrow$ **call** A_1
$\square\ d = 1 \rightarrow$ **call** A_2
$\square\ d = 2 \rightarrow$ **call** $\hat{F}_1$ **fi.**
$\hat{F}_2 \equiv$ **if** $L = \langle\rangle$ **then call** Z **fi**;
$\langle d, x\rangle \xleftarrow{\text{pop}} L;$
if $d = 0 \rightarrow$ **call** A_1
$\square\ d = 1 \rightarrow$ **call** A_2
$\square\ d = 2 \rightarrow$ **call** $\hat{F}_2$ **fi. endactions. end**

The actions $\hat{F}_1$ and $\hat{F}_2$ are identical (apart from calls to $\hat{F}_1$ and $\hat{F}_2$) so they can be merged using a transformation in [War89]. The result will be in the right form for Theorem 5.1. (Incidentally, the resulting action call graph is irreducible, but this causes no difficulties for our methods). Applying Theorem 5.1 gives:

proc $F(x) \equiv$
actions A_1 :
$A_1 \equiv$ **if** $x \neq 0$ **then if** $m[l[x]] = 0$ **then call** B_1
else call A_2 **fi**
else call Z **fi.**
$A_2 \equiv m[x] := 1;$
if $m[c[x]] = 0 \wedge m[r[x]] = 1$
then call B_2
elsif $m[c[x]] = 1 \wedge m[r[x]] = 0$
then call $\mathbf{B}_3$
elsif $m[c[x]] = 0 \wedge m[r[x]] = 0$
then call $\mathbf{B}_4$
else call Z **fi.**
$B_1 \equiv F(l[x]);$ **call** A_2**.**
$B_2 \equiv F(c[x]);$ **call** Z**.**
$B_3 \equiv F(r[x]);$ **call** Z**.**
$B_4 \equiv F(c[x]);\ F(r[x]);$ **call** Z**. endactions.**

Restructuring to remove the action system we get:

proc $F(x) \equiv$
if $x \neq 0$
then if $m[l[x]] = 0$ **then** $F(l[x])$ **fi**;
$m[x] := 1;$
if $m[c[x]] = 0 \vee m[r[x]] = 1$
then $F(c[x])$
elsif $m[c[x]] = 1 \vee m[r[x]] = 0$
then $F(r[x])$
elsif $m[c[x]] = 0 \vee m[r[x]] = 0$
then $F(c[x]);\ F(r[x])$ **fi fi.**

5.3 Change Data Representation

The next stage is a simple change in the data representation. We want to replace the low-level data structures (integer arrays) by equivalent high level structures (sets and functions). Let the set $\mathcal{N}$ of *nodes* represent the domain of $F()$ and of the arrays l, c, r and m. The array $m[]$ consists of 0s and 1s and is therefore equivalent to a subset of $\mathcal{N}$. We define the set M of *marked nodes* as follows:

$$M =_{\text{DF}} \{ x \in \mathcal{N} \mid m[x] = 1 \}$$

For each $x \in \mathcal{N}$, either $x = 0$, or there are three other elements of $\mathcal{N}$, recorded in the arrays $l[x]$, $c[x]$ and $r[x]$. We therefore define a function $D : \mathcal{N} \to \mathcal{N}^*$ as follows:

$$D(x) =_{\text{DF}} \begin{cases} \langle\rangle & \text{if } x = 0 \\ \langle l[x], c[x], r[c]\rangle & \text{otherwise} \end{cases}$$

With this data representation our program becomes:

proc $F(x) \equiv$
  if $D(x) \neq \langle\rangle$
    then if $D(x)[1] \notin M$ **then** $F(D(x)[1])$ **fi**;
      $M := M \cup \{x\}$;
      if $D(x)[2] \notin M \ \wedge \ D(x)[3] \in M$
        then $F(D(x)[2])$
      elsif $D(x)[2] \in M \ \wedge \ D(x)[3] \notin M$
          then $F(D(x)[3])$
      elsif $D(x)[2] \notin M \ \wedge \ D(x)[3] \notin M$
          then $F(D(x)[2])$; $F(D(x)[3])$ **fi fi.**

We can simplify the **if** statement by using an iteration over a sequence (see [PrW94] for the formal definition):

proc $F(x) \equiv$
  if $D(x) \neq \langle\rangle$
    then if $D(x)[1] \notin M$ **then** $F(D(x)[1])$ **fi**;
      $M := M \cup \{x\}$;
      for $y \xleftarrow{\text{pop}} D(x)[2\,..] \setminus M$ **do** $F(y)$ **od fi.**

where for a sequence X and set M, the expression $X \setminus M$ denotes the subsequence of elements of X which are not in M.

This version of the program can be generalised for *any* function $D : \mathcal{N} \to \mathcal{N}^*$, so from now on we will ignore the "trinary" nature of the original D function. This step is an "abstraction" in the sense that our original program will implement a special case of the specification we derive.

With the recursive and abstract version of the program it is clear that the effect of a call to $F(x)$ is to add certain nodes to the set M. Since all the recursive calls to $F(x)$ ensure that $x \notin M$, we will assume that this is the case for external calls also. Hence we can assume that the assertion $x \notin M$ holds at the beginning of the body of F. The arguments for the recursive calls are all elements of $D(x)$, so all the elements added to M will be reached by zero or more applications of D. For any set $X \subseteq \mathcal{N}$ we define $R(X)$ to be the set of nodes *reachable* from X via zero or more applications of D. This is called the *Transitive Closure* of D:

$$R(X) =_{\text{DF}} \bigcup_{n<\omega} R^n(X)$$

where

$$R^0(X) =_{\text{DF}} X \quad \text{and} \quad R^{n+1}(X) =_{\text{DF}} \bigcup \{ \mathsf{set}(D(y)) \mid y \in R^n(X) \}$$

We are also interested in the nodes reachable via unmarked nodes. We define $D_M(x) =_{\text{DF}} D(x) \setminus M$ which is the sequence $D(x)$ with elements of M deleted. We extend D_M to its transitive closure R_M in the same way as for D and R.

5.4 Abstraction Assumptions

To simplify the abstraction process we make two assumptions. The first is that all unmarked reachable nodes are reachable via unmarked nodes, i.e. $M \cup R(X) = M \cup R_M(X)$ initially for all $X \subseteq \mathcal{N}$. Since $M = \varnothing$ initially for our original program, this assumption is in fact a further generalisation of that program. Our second assumption is that no unmarked node is reachable from its first daughter node, i.e. $\forall x \in \mathcal{N} \setminus M.\ x \notin R(D(x)[1])$. This is an essential assumption since if $x \in R(D(x)[1])$ and $x \notin M$, then $x \in R_M(D(x)[1])$ and it is easy to see that $F(x)$ will not terminate.

In [War96] we prove the following *Reachability Theorem*:

Theorem 5.2 Let M and X be sets of nodes such that $M \cup R(X) = M \cup R_M(X)$ and let $x \in X \setminus M$. Let A and B be any subsets of $R_M(\{x\})$ such that $R_M(\{x\}) \setminus A \subseteq R_{M\cup A}(B)$. Then:

$$M \cup R_M(X) \;=\; M \cup A \cup R_{M\cup A}((X \setminus \{x\}) \cup B) \;=\; M \cup A \cup R((X \setminus \{x\}) \cup B)$$

Two obvious choices for A are $\{x\}$ and $R_M(\{x\})$. In the former case, a suitable choice for B is $D(x) \setminus (M \cup \{x\})$ and in the latter case, the only choice for B is $\varnothing$. So we have two corollaries:

Corollary 5.3 If $M \cup R(X) = M \cup R_M(X)$ and $x \in X \setminus M$ then:

$$M \cup R_M(X) \;=\; M \cup \{x\} \cup R_{M\cup\{x\}}(X') \;=\; M \cup \{x\} \cup R(X')$$

where $X' = (X \setminus \{x\}) \cup (D(x) \setminus (M \cup \{x\}))$.

Corollary 5.4 If $M \cup R(X) = M \cup R_M(X)$ and $x \in X \setminus M$ then:

$$M \cup R_M(X) = M \cup R(\{x\}) \cup R_{M\cup R(\{x\})}(X \setminus \{x\}) = M \cup R(\{x\}) \cup R(X \setminus \{x\})$$

5.5 The Specification

We claim that $F(x)$ is a refinement of $\mathsf{SPEC}(\{x\})$ where for $X \subseteq \mathcal{N}$:

$$\mathsf{SPEC}(X) \;=_{\text{DF}}\; I(X);\ M := M \cup R(X) \quad \text{where} \quad I(X) = \{M \cup R(X) = M \cup R_M(X)\}$$

To prove the claim, we will use the recursive implementation theorem Theorem 3.21. First, we replace the recursive calls in $F(x)$ by copies of the specification and add an assertion (from the abstraction assumptions):

$$\begin{array}{l}
\mathbf{S} = I(\{x\}); \\
\quad \textsf{if } D(x) \neq \langle\rangle \\
\quad\quad \textsf{then if } D(x)[1] \notin M \textsf{ then } \mathsf{SPEC}(D(x)[1]) \textsf{ fi}; \\
\quad\quad\quad\quad M := M \cup \{x\}; \\
\quad\quad\quad\quad \textsf{for } y \xleftarrow{\text{pop}} D(x)[2\,..] \setminus M \textsf{ do } \mathsf{SPEC}(y) \textsf{ od} \\
\quad\quad \textsf{else } M := M \cup \{x\} \textsf{ fi}
\end{array}$$

This expands to:

$$\begin{array}{l}
\mathbf{S} \;\approx\; I(\{x\}); \\
\quad \textsf{if } D(x) \neq \langle\rangle \\
\quad\quad \textsf{then if } D(x)[1] \notin M \textsf{ then } I(\{D(x)[1]\});\ M := M \cup R(\{D(x)[1]\}) \textsf{ fi}; \\
\quad\quad\quad\quad M := M \cup \{x\}; \\
\quad\quad\quad\quad \textsf{for } y \xleftarrow{\text{pop}} D(x)[2\,..] \setminus M \textsf{ do} \\
\quad\quad\quad\quad\quad I(\{y\});\ M := M \cup R(\{y\}) \textsf{ od} \\
\quad\quad \textsf{else } M := M \cup \{x\} \textsf{ fi}
\end{array}$$

If $D(x)[1] \notin M$ then $D(x)[1] \notin (M \cup \{x\})$ since our abstraction assumption $x \notin R(D(x)[1])$ implies $x \neq D(x)[1]$, so adding x to M does not affect the test. So the assignment $M := M \cup \{x\}$ can be moved back past the preceding **if** statement:

$$\begin{aligned}\mathbf{S} \;\approx\; & I(\{x\});\\ & \textbf{if } D(x) \neq \langle\rangle\\ & \quad \textbf{then } M := M \cup \{x\};\\ & \qquad \textbf{if } D(x)[1] \notin M \setminus \{x\} \textbf{ then } I(\{D(x)[1]\});\ M := M \cup R(\{D(x)[1]\}) \textbf{ fi};\\ & \qquad \textbf{for } y \xleftarrow{\text{pop}} D(x)[2\,..] \setminus M \textbf{ do}\\ & \qquad\quad I(\{y\});\ M := M \cup R(\{y\}) \textbf{ od}\\ & \quad \textbf{else } M := M \cup \{x\} \textbf{ fi}\end{aligned}$$

Now we roll the **if** statement into the **for** loop and factor $M := M \cup \{x\}$ out of the outer **if** statement. The test $D(x) \neq \langle\rangle$ then becomes redundant since **for** $y \xleftarrow{\text{pop}} D(x)$ **do** $\ldots$ **od** is equivalent to **skip** when $D(x) = \langle\rangle$.

$$\begin{aligned}\mathbf{S} \;\approx\; & I(\{x\});\\ & M := M \cup \{x\};\\ & \textbf{for } y \xleftarrow{\text{pop}} D(x) \setminus M \textbf{ do}\\ & \quad I(\{y\});\ M := M \cup R(\{y\}) \textbf{ od}\end{aligned}$$

By Corollary 5.4 and the general induction rule for iteration we can prove that for any $X \subseteq \mathcal{N}$ such that $M \cup R(X) = M \cup R_M(X)$:

$$\begin{aligned}& \textbf{for } y \in X \setminus M \textbf{ do} && \approx\; M := M \cup R_M(X)\\ & \quad I(\{y\});\ M := M \cup R(\{y\}) \textbf{ od} &&\end{aligned}$$

So we have:

$$\mathbf{S} \;\approx\; I(\{x\});\ M := M \cup \{x\};\ M := M \cup R_M(D(x)) \;\approx\; I(\{x\});\ M := M \cup \{x\} \cup R_M(D(x))$$

So by Corollary 5.3:

$$\mathbf{S} \;\approx\; \mathsf{SPEC}(\{x\})$$

Finally, note that before each copy of $\mathsf{SPEC}(\{x\})$ in $\mathbf{S}$, *either* M has been increased (and hence the finite set $\mathcal{N} \setminus M$ reduced) from its initial value, *or* M remains the same, but $R(\{x\})$ has been reduced (the abstraction assumption that $x \notin R(D(x)[1])$ shows that $R(D(x)[1]) \subset R(\{x\})$. So we can apply the recursive implementation theorem (Theorem 3.21) in reverse to prove:

$\mathsf{SPEC}(\{x\}) \;\approx$

$$\begin{aligned}& \textbf{begin}\\ & \quad F(x)\\ & \textbf{where}\\ & \textbf{proc } F(x) \;\equiv\\ & \quad \textbf{if } D(x) \neq \langle\rangle\\ & \qquad \textbf{then if } D(x)[1] \notin M \textbf{ then } F(D(x)[1]) \textbf{ fi};\\ & \qquad\qquad M := M \cup \{x\};\\ & \qquad\qquad \textbf{for } y \xleftarrow{\text{pop}} D(x)[2\,..] \setminus M \textbf{ do } F(y) \textbf{ od fi. end}\end{aligned}$$

So our original program is a correct implementation of $\mathsf{SPEC}(\{x\})$.

6 Conclusion

In this paper we have presented the theoretical foundation for a Wide Spectrum language which encompasses both specifications and low-level program constructs. The language we have developed includes general specifications expressed using first order logic and set theory and the usual iterative programming constructs. The advantages of the combination of set-theoretical semantics together with infinitary-logic based proof rules are illustrated in the proof of the theorem which provides a specification equivalent for any given program (Theorem 3.5): the proof of this and other theorems makes use of both proof techniques. We contend that a judicious combination of the techniques and constructions derived from these alternative viewpoints gives an economical and intuitive approach to program transformation theory. Our definition of the wide spectrum language also makes possible the induction rule for recursion (Lemma 3.11) which assists in the proofs of many transformations of recursive programs, and the implementation theorem (Theorem 3.21) which crosses the "abstraction gap" between a recursively defined specification and a recursive program which implements the specification. Infinitary logic is also used in the proof of the termination theorem for recursive procedures (for use with recursive procedures and loops not derived directly from a specification using Theorem 3.21).

7 Applications of the Theory

In [War96] we derive a data-intensive algorithm, (the Schorr-Waite Graph Marking Algorithm [ScW67]), from a formal specification by means of program transformations. This is a challenging example, due to the intermingled data and control flow, and the use of one data structure for two different purposes. The algorithm has been used as a "test bed" example for several program verification techniques applied to complex data structures ([Gri79, Kow79, Mor82, Roe78, Top79, YeD77]), all these verifications start with a statement of the algorithm and give no indications as to how it could be developed. In [War96] we start with a formal specification, and a vague idea of the technique to be used, and use formal transformations to derive the algorithm from the specification.

In [War90] we derive an efficient implementation of a Quicksort algorithm with "median of three partitioning" [Hoa62, Sed88]. In [War91] the language and transformations are used as the basis for a component repository, which enables reuse of specifications and development methods, as well as program code. In [War93] we use program transformations to transform a small but complex program into an equivalent, readable high-level specification. In [YoW93] we apply inverse engineering to a simple real-time program to extract its specification and determine the timing constraints for correct operation. In [WaB93] we describe a practical program transformation system for reverse engineering which is based on this program transformation theory and has been used successfully with a number of IBM Assembler modules.

The FermaT project at Software Migrations Ltd. is a complete redevelopment of the prototype transformation system to build an industrial strength CASE tool for reverse engineering, software maintenance and program understanding. For this implementation, we decided to extend WSL to add domain-specific constructs, creating *a language for writing program transformations*. This was called $\mathcal{META}$WSL. The extensions include an abstract data type for representing programs as tree structures and constructs for pattern matching, pattern filling and iterating over components of a program structure. The "transformation engine" of FermaT is implemented entirely in $\mathcal{META}$WSL. The implementation of $\mathcal{META}$WSL involves a parser for $\mathcal{META}$WSL (written in WSL), a WSL to Scheme translator (written in $\mathcal{META}$WSL and bootstrapped into Scheme using a prototype of the transformation system), a small support library of Scheme macros and functions, and a WSL runtime library (for the high-level $\mathcal{META}$WSL constructs such as **ifmatch**, **foreach**, **fill** etc.). The Scheme code [ADH98] is translated to C using the Hobbit compiler [Tam95] to generate highly portable C code which runs on Solaris, Linux, Windows 95, Windows NT and many other systems.

One aim of using WSL as the implementation language was so that the tool could be used to maintain its own source code: and this has already proved possible, with transformations being applied to simplify the source code for other transformations! Another aim was to test our theories on language oriented programming (see [War94b]): we expected to see a reduction in the total amount of source code required to implement a more efficient, more powerful and more rugged system. We also anticipated noticeable improvements in maintainability and portability. These expectations have been fulfilled, and we are achieving a high degree of functionality from a small total amount of easily maintainable code: the current system consists of around 30,000 lines of $\mathcal{METAWSL}$ and 1,000 lines of Scheme, while the previous version required over 100,000 lines of LISP. See [War94]. The 30,000 lines of $\mathcal{META}$WSL translates into 114,000 lines of Scheme which compiles to 125,000 lines of C code.

The $\mathcal{META}$WSL language encapsulates much of the expertise developed over the last twelve years of research in program transformation theory and transformation systems. As a result, this expertise is readily available to the programmers, some of whom have only recently joined the project. Working in $\mathcal{META}$WSL, it takes only a small amount of training before new programmers become effective at implementing transformations and enhancing the functionality of existing transformations.

Acknowledgements

The research described in this paper has been partly funded by Alvey project SE-088, partly through a DTI/SERC IEATP grant "From Assembler to Z using Formal Transformations", partly by funding from IBM UK Ltd. and partly by EPSRC[1] project "A Proof Theory for Program Refinement and Equivalence: Extensions".

I would like to thank Keith Bennett for many helpful comments and suggestions for this paper.

References

[ADH98] 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⟩.

[Ars82] J. Arsac, "Transformation of Recursive Procedures," in *Tools and Notations for Program Construction*, D. Neel, ed., Cambridge University Press, Cambridge, 1982, 211–265.

[Ars79] J. Arsac, "Syntactic Source to Source Transforms and Program Manipulation," *Comm. ACM* 22 (Jan., 1979), 43–54.

[Bac80] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, Amsterdam, 1980.

[Bac88] R. J. R. Back, "A Calculus of Refinements for Program Derivations," *Acta Informatica* 25 (1988), 593–624.

[BaW90] R. J. R. Back & J. von Wright, "Refinement Concepts Formalised in Higher-Order Logic," *Formal Aspects of Computing* 2 (1990), 247–272.

[BB85] F. L. Bauer, R. Berghammer & et. al., "The Munich Project CIP – The Wide Spectrum Language CIP-L," in *Lecture Notes in Computer Science 183*, Springer-Verlag, New York, 1985.

[BMP89] 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).

[BaT87] F. L. Bauer & The CIP System Group, *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, Lect. Notes in Comp. Sci. #292, Springer-Verlag, New York–Heidelberg–Berlin, 1987.

[BaW82] F. L. Bauer & H. Wossner, *Algorithmic Language and Program Development*, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

[1] The UK Engineering and Physical Sciences Research Council

[Bir87] R. Bird, "A Calculus of Functions for Program Derivation," Oxford University, Technical Monograph PRG-64, 1987.

[Bul90] T. Bull, "An Introduction to the WSL Program Transformer," *Conference on Software Maintenance 26th–29th November 1990, San Diego* (Nov., 1990).

[DaP90] B. A. Davey & H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, Cambridge, 1990.

[Dij76] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Eng68] E. Engeler, *Formal Languages: Automata and Structures*, Markham, Chicago, 1968.

[Gri79] D. Gries, "The Schorr-Waite Graph Marking Algorithm," *Acta Inform.* 11 (1979), 223–232.

[Hay87] I. J. Hayes, *Specification Case Studies*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Hoa62] C. A. R. Hoare, "Quicksort," *Comput. J.* 5 (1962), 10–15.

[HHJ87] 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.

[Jon86] C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Kar64] C. R. Karp, *Languages with Expressions of Infinite Length*, North-Holland, Amsterdam, 1964.

[Knu74] D. E. Knuth, "Structured Programming with the GOTO Statement," *Comput. Surveys* 6 (1974), 261–301.

[Kow79] R. Kowalski, "Algorithm = Logic + Control," *Comm. ACM* 22 (July, 1979), 424–436.

[Maj77] M. E. Majester, "Limits of the 'Algebraic' Specification of Abstract Data Types," *SIGPLAN Notices* 12 (Oct., 1977), 37–42.

[McN89] M. A. McMorran & J. E. Nicholls, "Z User Manual," IBM UK Laboratories Ltd., TR12.274, Hursley Park, July, 1989.

[Mor88] C. C. Morgan, "The Specification Statement," *Trans. Programming Lang. and Syst.* 10 (1988), 403–419.

[Mor94] C. C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1994, Second Edition.

[MoR87] C. C. Morgan & K. Robinson, "Specification Statements and Refinements," *IBM J. Res. Develop.* 31 (1987).

[MRG88] C. C. Morgan, K. Robinson & Paul Gardiner, "On the Refinement Calculus," Oxford University, Technical Monograph PRG-70, Oct., 1988.

[MoV93] C. C. Morgan & T. Vickers, *On the Refinement Calculus*, Springer-Verlag, New York–Heidelberg–Berlin, 1993.

[Mor82] J. H. Morris, "A Proof of the Schorr-Waite Algorithm," in *Theoretical Foundations of Programming Methodology* Int. Summer School, Marktoberdorf 1981, M. Broy & G. Schmidt, eds., Dordrecht: Reidel, 1982.

[Par84] H. Partsch, "The CIP Transformation System," in *Program Transformation and Programming Environments* Report on a Workshop directed by F. L. Bauer and H. Remus, P. Pepper, ed., Springer-Verlag, New York–Heidelberg–Berlin, 1984, 305–323.

[PrW94] 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⟩.

[Roe78] W. P. de Roever, "On Backtracking and Greatest Fixpoints," in *Formal Description of Programming Constructs*, E. J. Neuhold, ed., North-Holland, Amsterdam, 1978, 621–636.

[ScW67] H. Schorr & W. M. Waite, "An Efficient Machine-Independant Procedure for Garbage Collection in Various List Structures," *Comm. ACM* (Aug., 1967).

[Sed88] R. Sedgewick, *Algorithms*, Addison Wesley, Reading, MA, 1988.

[Sen90] C. T. Sennett, "Using Refinement to Convince: Lessons Learned from a Case Study," *Refinement Workshop, 8th–11th January, Hursley Park, Winchester* (Jan., 1990).

[Tam95] T. Tammet, "Lambda lifting as an optimization for compiling Scheme to C," Chalmers University of Technology, Department of Computer Sciences, Goteborg, Sweden, 1995, ⟨ftp://ftp.cs.chalmers.se/pub/users/tammet/hobbit.ps⟩.

[Tay84] D. Taylor, "An Alternative to Current Looping Syntax," *SIGPLAN Notices* 19 (Dec., 1984), 48–53.

[Ten76] R. D. Tennet, "The Denotational Semantics of Programming Languages," *Comm. ACM* 19 (Aug., 1976), 437–453.

[Top79] R. W. Topor, "The Correctness of the Schorr-Waite List Marking Algorithm," *Acta Inform.* 11 (1979), 211–221.

[War89] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

[War90] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/sorting-t.ps.gz⟩.

[War91a] M. Ward, "A Recursion Removal Theorem—Proof and Applications," Durham University, Technical Report, 1991, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/rec-proof-t.ps.gz⟩.

[War91b] M. Ward, "Specifications and Programs in a Wide Spectrum Language," Submitted to J. Assoc. Comput. Mach., 1991.

[War92] 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⟩.

[War94a] 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⟩.

[War94b] M. Ward, "Language Oriented Programming," *Software—Concepts and Tools* 15 (1994), 147–161, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/middle-out-t.ps.gz⟩.

[War96] M. Ward, "Program Analysis by Formal Transformation," *Comput. J.* 39 (1996), ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/topsort-t.ps.gz⟩.

[War99] M. Ward, "Recursion Removal/Introduction by Formal Transformation: An Aid to Program Development and Program Comprehension," *Comput. J.* 42 (1999), 650–673.

[War91] M. Ward, "Using Formal Transformations to Construct a Component Repository," in *Software Reuse: the European Approach*, Springer-Verlag, New York–Heidelberg–Berlin, Feb., 1991, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/reuse.ps.gz⟩.

[War93] 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⟩.

[War95] M. Ward, "A Definition of Abstraction," *J. Software Maintenance: Research and Practice* 7 (Nov., 1995), 443–450, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/abstraction-t.ps.gz⟩.

[War94] M. Ward, "Specifications from Source Code—Alchemists' Dream or Practical Reality?," *4th Reengineering Forum, September 19-21, 1994, Victoria, Canada* (Sept., 1994).

[War96] 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⟩.

[WaB93] M. Ward & K. H. Bennett, “A Practical Program Transformation System For Reverse Engineering,” *Working Conference on Reverse Engineering, May 21–23, 1993*, Baltimore MA (1993), ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/icse.ps.gz⟩.

[WaB95] M. Ward & K. H. Bennett, “Formal Methods to Aid the Evolution of Software,” *International Journal of Software Engineering and Knowledge Engineering* 5 (1995), 25–47, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/evolution-t.ps.gz⟩.

[WCM89] M. Ward, F. W. Calliss & M. Munro, “The Maintainer’s Assistant,” *Conference on Software Maintenance 16th–19th October 1989, Miami Florida* (1989), ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/MA-89.ps.gz⟩.

[YeD77] L. Yelowitz & A. G. Duncan, “Abstractions, Instantiations and Proofs of Marking Algorithms,” *SIGPLAN Notices* 12 (1977), 13–21, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/prog-spec.ps.gz⟩.

[YoW93] 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⟩.