Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations

M. P. Ward
Software Technology Research Laboratory, De Montfort University, Leicester
Martin.Ward@durham.ac.uk
http://www.cse.dmu.ac.uk/∼mward/

Abstract

Software reengineering has been described as being "about as easy as reconstructing a pig from a sausage" [11]. But the development of program transformation theory, as embodied in the FermaT transformation system, has made this miraculous feat into a practical possibility. This paper describes the theory behind the FermaT system and describes a recent migration project in which over 544,000 lines of assembler "sausage" (part of a large embedded system) were transformed into efficient and maintainable structured C code.

1 Introduction

In recent years program transformation technology has matured into a practical solution for many software reengineering and migration tasks.

In the past many systems were implemented in assembler code for various reasons: the most common probably being performance. With recent improvements in processor performance and compiler technology, raw performance is less of an issue and the limitations of assembler language become more important. Implementing a function point in assembler requires nearly three times as many lines of code, and costs nearly three times as much in comparison to C or COBOL [15]. Assembler is harder and more expensive to maintain, and it can be difficult to carry out extensive enhancements to a legacy assembler system. These days, business agility (the ability to respond quickly to new business opportunities) is frequently more important than raw performance: so many users are looking for ways to reengineer their legacy systems (not just assembler systems) to improve programmer productivity and system flexibility.

The FermaT transformation system uses formal proven program transformations, which preserve or refine the semantics of a program while changing its form. These transformations are applied to restructure and simplify legacy systems and to extract higher-level representations. By using an appropriate sequence of transformations, the extracted representation is guaranteed to be equivalent to the original code logic.

This paper describes one application of FermaT: migrating from assembler to a high level language. But FermaT and the program transformation theory can also be applied to other reengineering tasks (for example, where the target language is similar to or the same as the source language) and to systems development (refining from specifications to implementations).

1.1 History of FermaT

FermaT has its roots in the author's research into program transformation theory [36]. In the early 1990's a prototype transformation system, called the "Maintainer's Assistant", was developed at Durham University and implemented in LISP [43,46]. It included a large number of transformations, but was very much an "academic prototype" whose aim was to test the ideas rather than be a practical tool. In particular, little attention was paid to the time and space efficiency of the

implementation. Despite these drawbacks, the tool proved to be highly successful and capable of reengineering moderately sized assembler modules into equivalent high-level language programs. In the Maintainer's Assistant, programs were represented as LISP structures and the transformations were written in LISP.

For the next version of the tool, called GREET (Generic Reverse Engineering Tools) we decided to extend WSL (the Wide Spectrum Language on which the FermaT transformations operate) to add domain-specific constructs, creating *a language for writing program transformations*. 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 extended language is called $\mathcal{META}$WSL and all the transformations in GREET were written in $\mathcal{META}$WSL and translated to Lisp via the Concerto case tool builder. All the transformations from the Maintainer's Assistant were reimplemented in $\mathcal{META}$WSL, typically with enhancements, and many new transformations were added.

For the latest version (called FermaT), all the remaining parts of the system were reimplemented in $\mathcal{META}$WSL and a $\mathcal{META}$WSL to Scheme [2] translator was implemented (also in $\mathcal{META}$WSL). This was used to bootstrap the whole system to a Scheme implementation in a few weeks work. The FermaT system is implemented almost entirely in $\mathcal{META}$WSL.

This paper gives a brief introduction to the theory behind FermaT and describes a recent migration project in which over 544,000 lines of assembler "sausage" (part of a large embedded system) were transformed into efficient and maintainable structured C code.

1.2 Outline of the Paper

Section 2 discusses the importance and advantages of a formal approach to program transformations and why this is especially important when transformations are used in the context of reverse engineering, migration, or reengineering.

Section 3 defines the kernel level of WSL and outlines the methods for proving the correctness of a transformation or refinement. The details are in Appendix A.

Section 4 describes the extensions of the WSL kernel which form the language levels. It also briefly introduces the methods used to prove the correctness of program transformations and describes some simple restructuring transformations which are used extensively in assembler reengineering.

Section 5 outlines the historical development of the transformation theory.

Section 6 describes the application of FermaT to a major migration project: the automated migration of over half a million lines of assembler to structured and maintainable C code. The final migration step involved the automated application of nearly one and a half million transformations: this illustrates the need for complete automation, combined with the high reliability provided by formal methods.

Section 7 concludes with a discussion of the advantages of automated reengineering.

Finally, Appendix A gives detailed definitions and theorems for the results which were informally discussed in Section 3. Some of the ideas have been published elsewhere, the aim in this part of paper is to provide a self-contained, concise, readable, and up-to-date presentation of FermaT.

2 Why Use Formal Methods?

In the development of methods for program analysis and manipulation it is important to start from a rigorous mathematical foundation. Without such a foundation, it is all too easy to assume that a particular transformation is valid, and come to rely upon it, only to discover that there are certain special cases where the transformation fails.

A reliable foundation is especially important for an automated transformation system. In reengineering a single module, FermaT applies many thousands of individual transformations. It is therefore essential that we have a very high degree of confidence in the correctness of each transformation.

The following transformation was found in an article on program manipulation published in Communications of the ACM (See [4] Section 2.3.4):

do $\mathbf{S}_1$ **od** is equivalent to **do** $\mathbf{S}_2$ **od**
if and only if
do $\mathbf{S}_1$ **od** is equivalent to **do** $\mathbf{S}_1$; $\mathbf{S}_2$ **od**

Here, $\mathbf{S}_1$ and $\mathbf{S}_2$ are any statements and the **do** ... **od** loops are unbounded or infinite loops which can only be terminated by executing an **exit** statement within the loop body. The statement **exit**(n) will terminate n enclosing **do** ... **od** loops.

The reverse implication is easily seen to be false: simply take $\mathbf{S}_2$ to be **skip**, then for any statement $\mathbf{S}_1$:

do $\mathbf{S}_1$ **od** is equivalent to **do** $\mathbf{S}_1$; **skip od**

but:

do $\mathbf{S}_1$ **od** is not necessarily equivalent to **do skip od**

The "if and only if" may be a typo in the original paper because the forward implication looks more reasonable, and quite a useful transformation: it suggests that if we have two loops which implement the same program, then we can generate another program by combining the two loops. But consider these two programs:

do if $x \leqslant 0$ **then exit fi**;
  if even(x) **then** $x := x - 2$ **else** $x := x + 1$ **fi od**

and

do if $x \leqslant 0$ **then exit fi**;
  $x := x - 1$ **od**

Both programs will terminate with $x = 0$ if they are started in a state with the integer $x \geqslant 0$, otherwise both loops will terminate immediately.

The combined program is:

do if $x \leqslant 0$ **then exit fi**;
  if even(x) **then** $x := x - 2$ **else** $x := x + 1$ **fi**;
  if $x \leqslant 0$ **then exit fi**;
  $x := x - 1$ **od**

If $x = 1$ initially, then $x = 1$ at the end of the loop body. So this loop never terminates.

The point of this example is that it is very easy to invent a plausible new "transformation", and even base further research on it, before discovering that it is not valid. In sections 4.4 and 4.5 of [4] the author derives several further transformations from this one, some of which are valid and some are invalid.

This underlines the importance of a sound mathematical foundation. A sound foundation makes it possible to *prove* the correctness of a transformation and justifies building further research on top of the set of proven transformations. A sound foundation is also important for practical applications of transformation theory: the success of the migration project described in Section 6 depends on automatically applying thousands of separate transformations to each individual source file. It is therefore essential to have a very high degree of confidence in the correctness of *all* the transformations. It is simply not feasible to check by hand the accuracy of each transformation step: similarly, if each transformation step were to generate "proof obligations", then it would be impossible to discharge them all manually, even with the assistance of a theorem prover [34].

The failure of this proposed transformation raises a number of issues:

- How can we prove the correctness of a proposed transformation?
- Under what conditions is a proposed transformation actually valid? (for example, this particular transformation *is* valid when $\mathbf{S}_1 = \mathbf{S}_2$).
- Having developed and proved the correctness of a catalogue of transformations, can the proven transformations be applied automatically in a transformation system?
- Can the transformation system automatically check the applicability conditions for a transformation before it is applied?
- Given a transformation system which can check applicability conditions and apply transformations, to what extent can the system itself determine the sequence of transformations to be applied?

2.1 Formal Development Methods

Producing a program (or a large part of it) from a specification in a single step is a difficult task to carry out, to survey and to verify [6]. Moreover, programmers tend to underestimate the complexity of given problems and to overestimate their own mental capacity [31] and this exacerbates the situation further.

A solution in which a program is developed incrementally by *stepwise refinement* was proposed by Wirth [44]. However, the problem still remains that each step is done intuitively and must then be validated to determine whether the changes that have been made preserve the correctness of the program with respect to some specification, yet do not introduce unwanted side effects.

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

- The final program is correct (according to the initial specification) *by construction*.
- Transformations can be described by *semantic rules* and can thus be used for a whole class of problems and situations.
- Due to formality, the whole process of program development can be supported by the computer. A significant part of transformational programming involves the use of a large number of small changes to be made to the code. Performing such changes by hand would introduce clerical errors and the situation would be no better than the original *ad hoc* methods. However, such clerical work is ideally suited to automation, allowing the computer itself to carry out the monotonous part of the work, allowing the programmer to concentrate on the actual design decisions.

Development approaches in which each refinement step is first proposed by the developer and then verified correct (also by the developer but with automated assistance in the form of theorem provers) have had some success [21,22,45] but have also encountered difficulties in scaling to large programs. The scaling problems are such that some authors relegate formal methods to the specification stage of software development [13] for all but the most critical systems. These approaches are therefore of very limited application to reverse engineering, program comprehension or reengineering tasks [47].

In a survey of transformational programming [30] R. Paige wrote:

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

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

Xingyuan Zhang et al [48] have developed a formalisation of WSL in the type-theoretical proof assistant Coq. This has been used to mechanically verify the correctness of some simple restructuring transformations [49].

2.2 Formal Reverse Engineering Methods

The approach presented here, in which a large catalogue of proven transformations, together with their correctness conditions, is made available via a semi-automatic transformation system, has been proved capable of scaling up to large software developments and has the further advantage of being applicable in the reverse engineering and reengineering realm. Because the transformations are known to be correct, they can be applied "blindly" to an existing program whose function is not clearly understood in order to restructure and simplify the program into a more understandable form. FermaT is described as "semi-automatic" because the user selects each transformation and point of application, while the system checks the applicability conditions and applies the transformation. It should be noted that some of the transformations are actually meta-transformations which use heuristics to control the transformation process in order to apply a large number of other transformations to the whole target program. Many activities are therefore totally automated: including WSL restructuring and the whole migration process from assembler to C or COBOL (see [38] and Section 6).

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

The long range goal of transformational programming is to improve reliability, productivity, maintenance and analysis of software without sacrificing performance [30].

2.3 Related Work

2.3.1 Refinement

The *Refinement Calculus* approach to program derivation [14,24,26] is superficially similar to our program transformation method. It is based on a wide spectrum language, using Morgan's specification statement [23] and Dijkstra's guarded commands [10]. However, this language has very limited programming constructs: lacking loops with multiple exits, action systems with a "terminating" action, and side-effects. These extensions are essential if transformations are to be used for reverse engineering. The most serious limitation is that the transformations for introducing and manipulating loops require that any loops introduced must be accompanied by suitable invariant conditions and variant functions. This makes the method unsuitable for a practical reengineering method. Morgan remarks (pp 166–167 of [24]) that the whole development history is required for understanding the structure of the program and making safe modifications.

The Z specification notation [33] has recently been used to develop the specification and full refinement proof of a large, industrial scale application [34]. The proofs were carried out by hand but

the proof obligations and many of the steps were mechanically type-checked. The author's discuss the trade-offs they had to make to find the right path between greater mathematical formality and the need to press on and "just do" the proofs in any way they could.

2.3.2 Software Reuse

One of the first approaches to software reuse through domain modeling was Draco, proposed by Neighbors [28,29]. Draco enables analyses and designs to be reused, as well as actual software components. Draco includes many domain languages, refinements of programs between languages and simple pattern-matched transformations within a language.

2.3.3 Reverse Engineering

The AutoSpec program transformation system [12] uses *strongest postconditions* rather than weakest preconditions (see Appendix A). It is restricted to a partial correctness model and is therefore difficult to use with programs which make use of iteration or recursion.

The Design Maintenance System (DMS) [1] uses backwards transformation to achieve reverse engineering, where a series of transformations similar to those used in forward transformation (i.e. refinement), are used in an inverse manner. Bennett [8] also supports the use of transformations in reverse engineering. He suggests that a badly structured program could be transformed into one that is better structured and easier to comprehend.

2.3.4 Reengineering

Burson et al [9] describe an approach to reengineering based on the Refine toolset. They combine object oriented databases, program specification and pattern matching capability with program transformations facilities. (Refine is a programming language for creating, analyzing and transforming Abstract Syntax Trees of a target language.)

For some reengineering applications it may be possible to extract a grammar from the compiler source or online language documentation [19,20], apply automated transformations to the abstract syntax tree, and then regenerate source code in the same, or a similar language.

The "refactoring" phase of Extreme Programming [7] consists of purely syntactic transformations (carried out either manually or with the aid of a refactoring browser) followed by an automated regression test which aims to check the semantic correctness of the transformations.

Architectural modification [17] involves transformations which preserve the language but do data expansion, wrapper introduction or other special purpose modification. While refactoring transformations are pointwise, programmer-triggered design level changes,

Arnold [3] provides a comprehensive compendium of concepts, tools, techniques, case studies and the risks and benefits associated with reengineering.

3 The Kernel Language

The WSL transformation theory is based in *infinitary logic*: an extension of first order logic which allows infinitely long formulae. These infinite formulae are very useful for describing properties of programs: for example, termination of a **while** loop can be defined as "Either the loop terminates immediately, OR it terminates after one iteration OR it terminates after two iterations OR ..." With no (finite) upper bound on the number of iterations, the resulting description is an infinite formula.

3.1 Definition of the Kernel Language

The WSL kernel language consists of four primitive statements and three compound statements. The primitive statements are as follows (where **P** is any infinitary logic formula):

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 does not terminate;
2. **Guard:** $[\mathbf{P}]$ is a guard statement. It always terminates, and enforces $\mathbf{P}$ 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 $\mathbf{P}$ 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 $\mathbf{P}$);
3. **Add variables:** $\mathsf{add}(\boldsymbol{x})$ adds the variables in $\boldsymbol{x}$ to the state space and assigns arbitrary values to them. If the variables are already in the state space, then they still get assigned arbitrary values.
4. **Remove variables:** $\mathsf{remove}(\boldsymbol{x})$ removes the variables in $\boldsymbol{x}$ from the state space if they are present: i.e. it ensures that the variables are no longer in the state space.

Note that a WSL program can be infinitely long since it can include an infinite formula.

The compound statements are as follows; 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:** $(\mathbf{S}_1\ \sqcap\ \mathbf{S}_2)$ chooses one of $\mathbf{S}_1$ or $\mathbf{S}_2$ for execution, the choice being made nondeterministically;
3. **Recursion:** $(\mu X.\mathbf{S}_1)$ where X is a *statement variable* (taken from a suitable set of symbols). The statement $\mathbf{S}_1$ may contain occurrences of X as one or more of its component statements. These represent recursive calls to the procedure whose body is $\mathbf{S}_1$.

The semantics of a kernel statement is a function from states to sets of states, so the $\mathsf{add}(\boldsymbol{x})$ statement maps a state s to the set of all states which include $\boldsymbol{x}$ in the final state space and differ from s only on $\boldsymbol{x}$.

The reader may be wondering how we can implement a normal assignment when the only means of changing the value of a variable is to give it an arbitrary value, or to implement an **if** statement when the only way to specify more than one execution path is via a *nondeterministic* choice. Both of these effects can be achieved with the aid of guard statements to restrict the previous nondeterminacy. For example, the assignment $x := 1$ can be implemented by the sequence $(\mathsf{add}(x);\ [x = 1])$, while conditional statements can be achieved by a combination of nondeterministic choice and guards:

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

$$\textbf{if } \mathbf{B}_1 \rightarrow \mathbf{S}_1 \,\square\, \mathbf{B}_2 \rightarrow \mathbf{S}_2 \textbf{ fi} =_{\mathrm{DF}} \{\mathbf{B}_1 \vee \mathbf{B}_2\};\ \big(([\mathbf{B}_1];\ \mathbf{S}_1)\ \sqcap\ ([\mathbf{B}_2];\ \mathbf{S}_2)\big)$$

The kernel primitives have been described as "the quarks of programming" — rather mysterious objects which cannot be found in isolation (the guard statement cannot be implemented) but which combine to form more familiar objects: combinations which, until recently, were thought to be "atomic" and indivisible.

The kernel is a very simple and mathematically tractable language which contains all the operations needed for a programming and specification language. It is relatively easy to prove the correctness of transformations in the kernel language, but the language is not very expressive for programming. We extend the kernel language into an expressive programming language by defining new constructs in terms of the kernel. This extension is carried out in a series of layers (see Figure 1), with each layer building on the previous language level. (Note: the **loops** level in Figure 1 includes *action systems* (Section 4.2.2) which are used to implement the labels and jumps when translating from assembler to WSL)

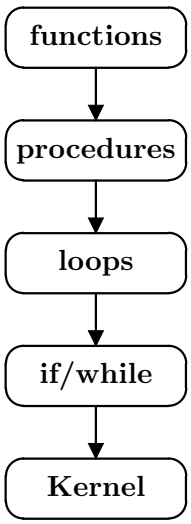

Figure 1: WSL Language Levels

3.2 Proof Methods

In this section we informally outline the methods for proving the correctness of WSL transformations. See Appendix A for the details.

A *state* is a collection of variables (the *state space*) each of which is assigned a value from a given set $\mathcal{H}$ of values. For example, the state $\{x \mapsto 0, y \mapsto 1\}$ has state space $\{x, y\}$ and assigns x the value 0 and y the value 1. The special state, denoted $\perp$, does not assign values to any variables but indicates nontermination or an error condition. A *state predicate* is a set of *proper states*, (states other then $\perp$). For example, if $\mathcal{H}$ is the set $\{0, 1\}$ then the state predicate $\{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$ contains all the states where $x = y$.

A *state transformation* is a function which describes the behaviour of a program. It maps each initial state to the set of possible final states (a nondeterministic program may have more than one possible final state for a given initial state). If $\perp$ is a possible final state, then we define that every other state is also in the set of final states: the program may choose not to terminate on the given initial state, so we do not care what else it might do.

A *statement* is a syntactic object (a collection of symbols structured according to the syntactic rules of infinitary first order logic, and the definition of the kernel language). There may be infinite formulae as components of the statement. If we interpret all the constant symbols, function symbols and relation symbols in the statement as elements of $\mathcal{H}$, functions on $\mathcal{H}$ and relations on $\mathcal{H}$ then we can interpret formulae as state predicates and statements as state transformations. For example, if we interpret $=$ as the equality relation, then the interpretation of $x = y$ on the value set $\{0, 1\}$ is the state predicate $\{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$. The four proper states are s_{00}, s_{01}, s_{10} and s_{11} where s_{ij} is the state $\{x \mapsto i, y \mapsto j\}$. So the interpretation of $x = y$ is $\{s_{00}, s_{11}\}$. The interpretation of the assertion statement $\{x = y\}$ is the function:

$$\{s_{00} \mapsto \{s_{00}\}, s_{01} \mapsto \{s_{00}, s_{01}, s_{10}, s_{11}, \perp\}, s_{10} \mapsto \{s_{00}, s_{01}, s_{10}, s_{11}, \perp\}, s_{11} \mapsto \{s_{11}\}\}$$

The interpretation of the guard statement $[x = y]$ is:

$$\{s_{00} \mapsto \{s_{00}\}, s_{01} \mapsto \{\}, s_{10} \mapsto \{\}, s_{11} \mapsto \{s_{11}\}\}$$

The interpretation of **add**$(\langle x \rangle)$ is:

$$\{s_{00} \mapsto \{s_{00}, s_{10}\}, s_{01} \mapsto \{s_{01}, s_{11}\}, s_{10} \mapsto \{s_{00}, s_{10}\}, s_{11} \mapsto \{s_{01}, s_{11}\}\}$$

The assignment $x := y$ can be implemented as the sequence $\mathbf{add}(\langle x \rangle);\ [x = y]$ which uses the guard to restrict the nondeterminacy of the **add** statement. The interpretation of $x := y$ therefore maps:

$$\{s_{00} \mapsto \{s_{00}\}, s_{01} \mapsto \{s_{11}\}, s_{10} \mapsto \{s_{00}\}, s_{11} \mapsto \{s_{11}\}\}$$

Note: every state transformation also maps $\bot \mapsto \{s_{00}, s_{01}, s_{10}, s_{11}, \bot\}$, but this mapping has been removed for brevity.

Two statements are *equivalent* if their interpretations are identical. A state transformation f_1 is refined by f_2, written $f_1 \leq f_2$, iff for every initial state s we have $f_2(s) \subseteq f_1(s)$. The statement $\{\mathbf{false}\}$ (also written as **abort**) is refined by every other statement, while the statement $[\mathbf{false}]$ refines every other statement.

Given a state transformation f and state predicate e, the *weakest precondition* $\mathrm{wp}(f, e)$ is the state predicate containing all the initial states s such that $f(s) \subseteq e$. This is the weakest condition on the initial state such that if f is started in a state satisfying this condition, then it is guaranteed to terminate in a state satisfying e. The importance of weakest preconditions is that the refinement relation can be characterised using weakest preconditions: $f_1 \leq f_2$ iff $\forall e.\, (\mathrm{wp}(f_1, e) \subseteq \mathrm{wp}(f_2, e))$.

We can define a weakest precondition for statements, where the postcondition is a formula, as a simple formula of infinitary logic. If $\mathbf{S}$ is any statement and $\mathbf{R}$ is any formula, then the formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ has the interpretation $\mathrm{wp}(f, e)$ whenever f is the interpretation of $\mathbf{S}$ and e is the interpretation of $\mathbf{R}$. See Appendix A for the definition of WP.

The Appendix also shows that instead of looking at the WP of statements $\mathbf{S}_1$ and $\mathbf{S}_2$ on *all* postconditions, it is sufficient to check the two special postconditions **true** and $\boldsymbol{x} \neq \boldsymbol{x}'$ (where $\boldsymbol{x}$ is a list of all the variables in the final state space and $\boldsymbol{x}$' is a list of new variables not used elsewhere).

The result is that to prove the validity of a refinement $\mathbf{S}_1 \ \leq\ \mathbf{S}_2$ under a set of assumptions Δ, it is sufficient to prove that the two formulae:

$$\mathrm{WP}(\mathbf{S}_1, \mathbf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{true}) \quad \text{and} \quad \mathrm{WP}(\mathbf{S}_1, \boldsymbol{x} \neq \boldsymbol{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \boldsymbol{x} \neq \boldsymbol{x}')$$

can be proved (or deduced) from the set Δ of assumptions. If this is the case, then we write

$$\Delta \vdash \mathbf{S}_1 \ \leq\ \mathbf{S}_2$$

This is true precisely when every interpretation which interprets every assumption in Δ as true also interprets $\mathbf{S}_2$ as a state transformation which is a refinement of the interpretation of $\mathbf{S}_1$.

This proof technique is *sound* in the sense that if the two implications can be proved then $\mathbf{S}_2$ definitely is a refinement of $\mathbf{S}_1$. It is *complete* in the sense that if $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$ then the two implications can always be proved.

3.3 Specification Statements

A simple combination of kernel statements is used to construct the *specification statement* $\boldsymbol{x} := \boldsymbol{x}'.\mathbf{Q}$ where $\boldsymbol{x}$ is a sequence of variables and $\boldsymbol{x}'$ the corresponding sequence of "primed variables", and $\mathbf{Q}$ is any formula. This assigns new values to the variables in $\boldsymbol{x}$ so that the formula $\mathbf{Q}$ is true where (within $\mathbf{Q}$) $\boldsymbol{x}$ represents the old values and $\boldsymbol{x}'$ represents the new values. If there are no new values for $\boldsymbol{x}$ which satisfy $\mathbf{Q}$ then the statement aborts. For example, the statement:

$$\langle x \rangle := \langle x' \rangle.(x^2 = y)$$

will set x to a square root of y. If $y = 4$ initially then the statement will set x to either 2 or -2 nondeterministically.

The formal definition of $\boldsymbol{x} := \boldsymbol{x}'.\mathbf{Q}$ is:

$$\{\exists \boldsymbol{x}'.\, \mathbf{Q}\};\ \mathbf{add}(\boldsymbol{x}');\ [\mathbf{Q}];\ \mathbf{add}(\boldsymbol{x});\ [\boldsymbol{x} = \boldsymbol{x}'];\ \mathbf{remove}(\boldsymbol{x}')$$

The first assertion ensures that the statement aborts if there are no values for the $\boldsymbol{x}'$ variables which satisfy $\mathbf{Q}$. The next two statements add $\boldsymbol{x}'$ to the state space with arbitrary values and then restrict the values to satisfy $\mathbf{Q}$. The final three statements copy the values from $\boldsymbol{x}'$ to $\boldsymbol{x}$ and then remove $\boldsymbol{x}'$ from the state space. It is assumed that the "primed variables" are a separate set of variables which are not used outside specification statements.

An important property of this specification statement is that it is guaranteed *null-free*: for every input state the set of output states is non-empty. A null program is a program for which the set of output states is empty for one or more initial states. An example is the guard [**false**] which is null for every initial state. Such a program (vacuously) satisfies *any* specification for that input state: since the specification states that every output state must satisfy the given postcondition. A null program is therefore a correct refinement of any specification, but is also not implementable on a machine (since the physical machine must terminate in *some* state if it is guaranteed to terminate). It is therefore important to avoid inadvertently introducing null programs in the refinement process.

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

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

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 Theorem A.12 we show that *any* WSL program can be transformed into a single equivalent specification statement. This shows that the specification statement is sufficiently general to define the specification of any program.

In [23,24,25,27] a different specification statement is described. The Morgan specification statement is written $\boldsymbol{x}$: [Pre, Post] where 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 $\boldsymbol{x}$. In our notation an equivalent statement is {Pre}; **add**($\boldsymbol{x}$); [Post]. The disadvantage of this notation is that it makes the user responsible for ensuring that he never refines a specification into an (unimplementable) null statement. Also, there is no guarantee that a desired specification can be written as a Morgan specification statement: for example, the simple program $x := x + 1$ cannot be specified with a single Morgan specification statement.

We finish this section with a simple transformation which is easily proved using weakest preconditions:

Transformation 1 *Expand Choice*
We can expand a nondeterministic choice operator over surrounding statements:

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

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

In this paper all the transformations we discuss are directly or indirectly relevant to reengineering tasks.

4 Language Extensions

The complete WSL language is developed from the kernel 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 "zero level" language which forms the foundation for all the others.

In Appendix A we describe some of the basic transformations and induction rules which can be derived for the kernel level of WSL. Many transformations for the higher levels of WSL are proved by translating to a lower level and applying a combination of basic transformations and induction rules.

4.1 The if/while Level Language

The if/while level contains assertions, sequencing, recursion, and nondeterministic choice (as in the kernel). The **add**, **remove** and guard statements are replaced by specification statements (see Section 3.3), assignments and local variables. In addition **if** statements, **while** loops, Dijkstra's guarded commands [10] and **for** loops are included. **while** loops are defined in terms of recursion: Let $\mathbf{B}$ be any formula and $\mathbf{S}$ be any statement and let X be a new statement variable. Then:

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

See [36] for a full description of the first level language in terms of the kernel.

For the first level language, all the new constructs are provably null-free (see Section 3.3). As a result, if a program contains no explicit guard statements, then it is guaranteed to be null-free. This is important for two reasons: firstly, null statements are not implementable and secondly some very useful transformations cease to be valid when applied to null statements. One example is assertion moving:

Transformation 2 *Move Assertion*
If $B \wedge \text{WP}(\mathbf{S}, \textbf{true}) \iff \text{WP}(\mathbf{S}, \mathbf{B})$ then $\Delta \vdash \mathbf{S};\ \{\mathbf{B}\} \approx \{\mathbf{B}\};\ \mathbf{S}$

Proof:

$$\begin{aligned} \text{WP}(\mathbf{S};\ \{\mathbf{B}\}, \mathbf{R}) &\iff \text{WP}(\mathbf{S}, \mathbf{B}) \wedge \text{WP}(\mathbf{S}, \mathbf{R}) \\ &\iff \mathbf{B} \wedge \text{WP}(\mathbf{S}, \textbf{true}) \wedge \text{WP}(\mathbf{S}, \mathbf{R}) \end{aligned}$$

by the premise

$$\iff \mathbf{B} \wedge \text{WP}(\mathbf{S}, \mathbf{R})$$

by a property of WP

$$\iff \text{WP}(\{\mathbf{B}\};\ \mathbf{S}, \mathbf{R})$$

■

We might expect to be able to move an assertion past any statement which does not modify any of the variables in the assertion, but this is only guaranteed for statements which are null-free. Consider the statement [**false**]: only assertions equivalent to **skip** can be moved past [**false**] since when [**false**] is at the beginning of the sequence, the whole statement is equivalent to [**false**].

This is another reason for excluding guards from the WSL levels beyond the kernel level.

Transformation 3 *Swap Statements*
If $\mathbf{S}_1$ and $\mathbf{S}_2$ contain no guard statements and no variable modified in $\mathbf{S}_1$ is used in $\mathbf{S}_2$ and no variable modified in $\mathbf{S}_2$ is used in $\mathbf{S}_1$ then:

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

The proof is by an induction on the recursion nesting and structure of $\mathbf{S}_2$ using Transformation 2 as one of the base cases.

Transformation 4 *Splitting a tautology*
If $\mathbf{B}_1 \vee \mathbf{B}_2 \iff \textbf{true}$ then:

$$\Delta \vdash \mathbf{S} \approx \textbf{if } \mathbf{B}_1 \rightarrow \mathbf{S} \ \square\ \mathbf{B}_2 \rightarrow \mathbf{S} \textbf{ fi}$$

Putting $\mathbf{B}_2 = \neg \mathbf{B}_1$ we have:

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

Transformation 5 *Expand Conditional*

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

Transformation 6 *Introduce assertions*

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

Transformation 7 *Introduce assertion*
If the variables in $\boldsymbol{x}$ do not appear free in $\mathbf{Q}$ (i.e. the new value of $\boldsymbol{x}$ does not depend on the old value) then:

$$\Delta \vdash \boldsymbol{x} := \boldsymbol{x}'.(\mathbf{Q}) \ \approx \ \boldsymbol{x} := \boldsymbol{x}'.(\mathbf{Q}); \ \{\mathbf{Q}[\boldsymbol{x}/\boldsymbol{x}']\}$$

In particular, if x does not appear in e, then:

$$\Delta \vdash x := e \ \approx \ x := e; \ \{x = e\}$$

Transformation 8 *Prune conditional*
If $\mathbf{B}' \Rightarrow \mathbf{B}$ then:

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

If $\mathbf{B}' \Rightarrow \neg\mathbf{B}$ then:

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

Transformation 9 *Expand Conditional Backwards*
If both $\mathbf{B}$ and $\neg\mathbf{B}$ are invariant over $\mathbf{S}$ then:

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

Proof: The proof uses the previous transformations:

$$\begin{aligned} &\mathbf{S}; \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi} \\ &\quad \approx \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}; \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi} \\ &\qquad\quad \textbf{else } \mathbf{S}; \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi fi} \end{aligned}$$

by splitting a tautology, Trans 4

$$\begin{aligned} &\approx \ \textbf{if } \mathbf{B} \textbf{ then } \{\mathbf{B}\}; \ \mathbf{S}; \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi} \\ &\qquad\quad \textbf{else } \{\neg\mathbf{B}\}; \ \mathbf{S}; \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi fi} \end{aligned}$$

by introducing assertions, Trans 6

$$\begin{aligned} &\approx \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}; \ \{\mathbf{B}\}; \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi} \\ &\qquad\quad \textbf{else } \mathbf{S}; \ \{\neg\mathbf{B}\}; \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi fi} \end{aligned}$$

by assertion moving, Trans 2

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

by prune conditional, Trans 8

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

by assertion moving, Trans 2

$$\approx \ \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}; \ \mathbf{S}_1 \textbf{ else } \mathbf{S}; \ \mathbf{S}_2 \textbf{ fi}$$

by the converse of introducing assertions.

These restructuring and simplification transformations may appear to be trivial, but they are very useful in the automatic transformation of assembler code. For example, the following 186 assembler code:

```
cmp al,5
jnz foo
bar: . . .
```

is translated to this WSL code:

if ax[1] = 5 **then** zf := 1 **else** zf := 0 **fi**;
if ax[1] < 5 **then** cf := 1 **else** cf := 0 **fi**;
if zf = 0 **then call** foo **fi**;
call bar;

Swap the first two statements (Trans 3) and expand the **if** statement forwards (Trans 5):

if ax[1] < 5 **then** cf := 1 **else** cf := 0 **fi**;
if ax[1] = 5 **then** zf := 1; **if** zf = 0 **then call** foo **fi**
else zf := 0; **if** zf = 0 **then call** foo **fi fi**;
call bar;

Insert assertions (Trans 7), prune the inner conditionals (Trans 8) and remove the assertions again (Trans 7):

if ax[1] < 5 **then** cf := 1 **else** cf := 0 **fi**;
if ax[1] = 5 **then** zf := 1 **else** zf := 0; **call** foo **fi**;
call bar;

If this is the only place where these values of zf and cf are used (which is usually the case for assembler code) then dataflow analysis will show that all the assignments to zf and cf in this section of code are redundant and can be deleted. The result then simplifies to:

if ax[1] < 5 **then call** foo **fi**;
call bar;

This may seem like a lot of work for a simple analysis of two assembler instructions, but the careful approach of (a) translating all side-effects, (b) applying general-purpose transformations, and (c) only deleting code that can be proved to be redundant, applies to all kinds of unstructured sequences of compare and branch instructions. In addition, FermaT carries out all these transformations automatically.

4.2 The Loops Level

The loops level introduces unbounded loops with **exit**s, and action systems.

4.2.1 Exit Statements

WSL includes statements of the form **exit**(n), where n is an integer, (*not* a variable) which occur within loops of the form **do S od** where **S** is a statement. These were described in [18] and [35], while Arsac [4] describes some transformations on these loops. They are "infinite" or "unbounded" loops which can only be terminated by the execution of an **exit**(n) 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 or **if** statement.

More formally, we define the notion of a *simple statement*:

Definition 4.1 A *simple statement* is any if/while level statement apart from: (1) a sequence (2) a deterministic or nondeterministic choice, and (3) a **do** ... **od** loop. The predicate simple(**S**) is true when **S** is a simple statement.

The restriction on the placement of **exit** statements means that a simple statement may not have an **exit**(k) statement as a component unless it occurs within k or more nested **do** ... **od** loops. Note that **do** ... **od** loops and action systems are allowed as components of a simple statement.

These restrictions are motivated by the following concerns:

1. We want to be able to determine all the possible exit points from a loop by a simple *syntactic* analysis of the loop body. Consider the loop:
   do $F(x)$;
   $x := x - 1$;
   if $x \leqslant 0$ **then exit fi od**

   This appears to have one exit (and to terminate with $x \leqslant 0$). But if the body of $F(x)$, or any procedure called by F, is allowed to contain unprotected **exit** statements, then these assumptions will fail.

2. Similarly, we would like to preserve the fact that a **while** loop can only terminate when the terminating condition becomes **false**, a **for** loop will only terminate after the appropriate number of iterations, and so on.

The interpretation of these statements in terms of the if/while level is as follows (but see also the next section for the full interpretation when action systems are also involved):

We add a new integer variable dp to record the current depth of nesting of loops. At the beginning of the program we insert the assertion $\{\mathsf{dp} = 0\}$ and each **exit** statement **exit**(k) is translated as $\mathsf{dp} := \mathsf{dp} - k$ since it changes the depth of "current execution" by moving out of k enclosing loops. To prevent any more statements at the current depth being executed after an **exit** statement has been executed we surround all simple statements (including the **exit**s) by "guards" which are **if** statements which will test dp and only allow the statement to be executed if dp has the correct value. Each unbounded loop **do S od** is translated as:

$$\textbf{if } \mathsf{dp} = n \textbf{ then } \mathsf{dp} := n + 1;\ \textbf{while } \mathsf{dp} = n + 1 \textbf{ do } \mathfrak{G}_{n+1}(\mathbf{S}) \textbf{ od fi}$$

where n is an integer constant representing the depth of the loop ($n = 0$ for top-level statements, $n = 1$ for statements within an outermost loop etc.) and $\mathfrak{G}_n(\mathbf{S})$ is the statement $\mathbf{S}$ with each component statement "guarded" (surrounded by an enclosing **if** statement of the form **if** $\mathsf{dp} = n$ **then** ... **fi**) so that if the depth is changed by an **exit** statement then no more statements in the loop will be executed and the loop will terminate. The important property of a guarded statement is that it will only be executed if dp has the correct value. Thus: $\Delta \vdash \{\mathsf{dp} \neq n\};\ \mathfrak{G}_n(\mathbf{S}) \approx \{\mathsf{dp} \neq n\};$ **skip** by Transformation 8.

So for example, the program:

```
do do last := item[i];
        i := i + 1;
        if i = n + 1 then write(count); exit(2) fi;
        if item[i] ≠ last then write(count); exit(1)
                         else count := count + number[i] fi od;
   count := number[i] od
```

translates to the following:

```
{dp = 0};
dp := 1;
while dp = 1 do
  dp := 2;
  while dp = 2 do
    if dp = 2 then last := item[i] fi;
    if dp = 2 then i := i + 1 fi;
    if dp = 2 then if i = n + 1 then write(count); dp := dp − 2 fi fi;
    if dp = 2
      then if item[i] ≠ last then write(count); dp := dp − 1
                            else count := count + number[i] fi fi od;
  if dp = 1 then count := number[i] fi od
```

Note that the expansion of the program text size in translating back to the if/while level is only relevant in proving the correctness of a transformation involving **exit**s. To prove a transformation, one first translates both sides into if/while level and then uses if/while level transformations to prove their equivalence. (The if/while level transformations were themselves proved correct by translating to the kernel level.) For example:

Transformation 10 *Loop to While*
If $\mathbf{S}$ is a *proper sequence* then:

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

A proper sequence is a statement where every **exit**(k) is contained within at least k enclosing **do** ... **od** loops.

To prove this transformation, we first translate back to if/while level:

$$\begin{aligned}
&\{\mathsf{dp} = 0\};\ \mathfrak{G}_0(\textbf{do if B then exit fi; S od}) \\
&\quad\approx\ \{\mathsf{dp} = 0\};\ \mathsf{dp} := 1; \\
&\qquad \textbf{while } \mathsf{dp} = 1 \textbf{ do} \\
&\qquad\quad \textbf{if B then if } \mathsf{dp} = 1 \textbf{ then } \mathsf{dp} := \mathsf{dp} - 1 \textbf{ fi fi}; \\
&\qquad\quad \mathfrak{G}_1(\mathbf{S}) \textbf{ od}
\end{aligned}$$

Simplify the **if** statements and expand forwards over $\mathfrak{G}_1(\mathbf{S})$:

$$\begin{aligned}
&\quad\approx\ \{\mathsf{dp} = 0\};\ \mathsf{dp} := 1; \\
&\qquad \textbf{while } \mathsf{dp} = 1 \textbf{ do} \\
&\qquad\quad \textbf{if B then } \mathsf{dp} := 0;\ \mathfrak{G}_1(\mathbf{S}) \textbf{ else } \mathfrak{G}_1(\mathbf{S}) \textbf{ fi od}
\end{aligned}$$

Use the fact that $\mathfrak{G}_1(\mathbf{S}) \;\approx\; \textbf{skip}$ when $\mathsf{dp} \neq 1$:

$$\begin{aligned}
&\quad\approx\ \{\mathsf{dp} = 0\};\ \mathsf{dp} := 1; \\
&\qquad \textbf{while } \mathsf{dp} = 1 \textbf{ do} \\
&\qquad\quad \textbf{if B then } \mathsf{dp} := 0 \textbf{ else } \mathfrak{G}_1(\mathbf{S}) \textbf{ fi od}
\end{aligned}$$

Convert to a recursive procedure:

$$\begin{aligned}
&\quad\approx\ \{\mathsf{dp} = 0\};\ \mathsf{dp} := 1; \\
&\qquad (\mu X.\textbf{if } \mathsf{dp} = 1 \\
&\qquad\qquad \textbf{then if B then } \mathsf{dp} := 0 \\
&\qquad\qquad\qquad\qquad \textbf{else } \mathsf{dp} := 0;\ \mathfrak{G}_0(\mathbf{S});\ \mathsf{dp} := 1 \textbf{ fi}; \\
&\qquad\qquad\quad X)
\end{aligned}$$

Expand the inner **if** over X, unfold the first call to X and prune the **if**:

$$\begin{aligned}
&\quad\approx\ \{\mathsf{dp} = 0\};\ \mathsf{dp} := 1; \\
&\qquad (\mu X.\textbf{if } \mathsf{dp} = 1 \\
&\qquad\qquad \textbf{then if B then } \mathsf{dp} := 0 \\
&\qquad\qquad\qquad\qquad \textbf{else } \mathsf{dp} := 0;\ \mathfrak{G}_0(\mathbf{S});\ \mathsf{dp} := 1;\ X \textbf{ fi})
\end{aligned}$$

Expand the procedure over $\mathsf{dp} := 1$ and prune the outer **if**:

$$\begin{aligned}
&\quad\approx\ \{\mathsf{dp} = 0\}; \\
&\qquad (\mu X.\mathsf{dp} := 1; \\
&\qquad\quad \textbf{if B then } \mathsf{dp} := 0 \\
&\qquad\qquad \textbf{else } \mathsf{dp} := 0;\ \mathfrak{G}_0(\mathbf{S});\ X \textbf{ fi})
\end{aligned}$$

Take $\mathsf{dp} := 0$ out of the **if** and merge the assignments:

$$\begin{aligned}
&\quad\approx\ \{\mathsf{dp} = 0\};\ (\mu X.\textbf{if B then skip else } \mathfrak{G}_0(\mathbf{S});\ X \textbf{ fi}) \\
&\quad\approx\ \{\mathsf{dp} = 0\}; \textbf{ while } \neg\mathbf{B} \textbf{ do } \mathfrak{G}_0(\mathbf{S}) \textbf{ od} \\
&\quad\approx\ \{\mathsf{dp} = 0\};\ \mathfrak{G}_0(\textbf{while } \neg\mathbf{B} \textbf{ do S od})
\end{aligned}$$

4.2.2 Action Systems

The recursive statement in the kernel language does not directly allow the definition of mutually recursive procedures (since all calls to a procedure must occur within the procedure body). However we can define a set of mutually recursive procedures by putting them all within a single procedure.

An *Action System* is a set of parameterless mutually recursive procedures together with the name of the first action to be called. There may be a special action Z (with no body): **call** Z results in the immediate termination of the whole action system with execution continuing with the next statement after the action system (if any).

If the execution of any action body must lead to an action call, then the action system is *regular*. A regular action system, no call ever returns and the system can only be terminated by a **call** Z. A program written using labels and jumps translates directly into an action system, provided all the labels appear at the top level (not inside a structure). Labels can be promoted to the top level by introducing extra calls, for example:

A : **if** $\mathbf{B}$ **then** $L : \mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**; $\mathbf{S}_3$

can be translated to the action system:

actions A :
$A \equiv$ **if** $\mathbf{B}$ **then call** L **else call** $L2$ **fi.**
$L \equiv \mathbf{S}_1$; **call** $L3$**.**
$L2 \equiv \mathbf{S}_2$; **call** $L3$**.**
$L3 \equiv \mathbf{S}_3$; **call** Z**. endactions**

In a non-regular system, if the end of the body of an action is reached, then control is passed to the action which called it (or to the statement following the action system) rather than "falling through" to the next label. An action system with no **call** Z is called *recursive*: in a recursive system every action call returns normally.

Certain complications arise when action systems with **call** Z are mixed with **do** ... **od** loops and **exit** statements: for example should we allow an action of the form: $E \equiv$ **exit**(2) with an occurrence of **call** E being equivalent to **exit**(2)? This would make it impossible to determine the terminal values of a statement from the text of the statement alone: it would be necessary to examine the whole of any enclosing action system. Our solution to this problem is to overwrite the value of dp with 0 before each action call, and restore it on return. The "guard" mechanism is extended to cope with the **call** Z terminating action: a new variable act indicates which action body to execute when the main procedure is called. The $\mathfrak{G}_n$ function annotates $\mathbf{S}$ to check that $\mathsf{dp} = n$ and $\mathsf{act} \neq$ "Z" before executing $\mathbf{S}$.

The action system:

actions A_1 :
$A_1 \equiv \mathbf{S}_1$**.**
$A_2 \equiv \mathbf{S}_2$**.**
...
$A_n \equiv \mathbf{S}_n$**. endactions**

(where statements $\mathbf{S}_1, \ldots, \mathbf{S}_n$ are proper sequences) is defined as follows:

var $\langle \mathsf{dp} := 0, \mathsf{act} :=$ "A_1"$\rangle$:
  (μA. **if** $\mathsf{act} =$ "A_1" $\rightarrow \mathsf{act} :=$ "O"; $\mathfrak{G}_0(\mathbf{S}_1)$
    $\square$ $\mathsf{act} =$ "A_2" $\rightarrow \mathsf{act} :=$ "O"; $\mathfrak{G}_0(\mathbf{S}_2)$
      ...
    $\square$ $\mathsf{act} =$ "A_n" $\rightarrow \mathsf{act} :=$ "O"; $\mathfrak{G}_0(\mathbf{S}_n)$ **fi**) **end**

Here act is a new variable which contains the name of the next action to be invoked and $\mathfrak{G}_n(\mathbf{S})$

is defined as follows. For simple statements:

$$\begin{aligned}
\mathfrak{G}_n(\textbf{call } Z) &=_{\text{DF}} \textbf{ if } \mathsf{act} = \text{“}O\text{”} \wedge \mathsf{dp} = n \textbf{ then } \mathsf{dp} := 0;\ \mathsf{act} := \text{“}Z\text{”} \textbf{ fi} \\
\mathfrak{G}_n(\textbf{call } A_i) &=_{\text{DF}} \textbf{ if } \mathsf{act} = \text{“}O\text{”} \wedge \mathsf{dp} = n \\
&\qquad \textbf{then } \mathsf{act} := \text{“}A_i\text{”};\ \mathsf{dp} := 0;\ A; \\
&\qquad\quad \textbf{if } \mathsf{act} = \text{“}O\text{”} \textbf{ then } \mathsf{dp} := n \textbf{ fi fi} \\
\mathfrak{G}_n(\textbf{exit}(k)) &=_{\text{DF}} \textbf{ if } \mathsf{act} = \text{“}O\text{”} \wedge \mathsf{dp} = n \textbf{ then } \mathsf{dp} := \mathsf{dp} - k \textbf{ fi}
\end{aligned}$$

The other simple statements cannot be terminated by **call** or **exit** statements, but they may contain nested action systems or **do** ... **od** loops, so $\mathfrak{G}_n$ still has to be applied to each component. For example:

$\mathfrak{G}_n(\textbf{while B do S od}) =_{\text{DF}} \textbf{ if } \mathsf{act} = \text{“}O\text{”} \wedge \mathsf{dp} = n \textbf{ then while B do } \mathfrak{G}_n(\mathbf{S}) \textbf{ od fi}$

For non-simple statements we define:

$$\begin{aligned}
\mathfrak{G}_n(\mathbf{S}_1;\ \mathbf{S}_2) &=_{\text{DF}} \mathfrak{G}_n(\mathbf{S}_1);\ \mathfrak{G}_n(\mathbf{S}_2) \\
\mathfrak{G}_n(\mathbf{S}_1 \sqcap \mathbf{S}_2) &=_{\text{DF}} \mathfrak{G}_n(\mathbf{S}_1) \sqcap \mathfrak{G}_n(\mathbf{S}_2) \\
\mathfrak{G}_n(\textbf{if } \mathbf{B}_1 \to \mathbf{S}_1 \ \square \ldots \ \square\ \mathbf{B}_m \to \mathbf{S}_m \textbf{ fi}) &=_{\text{DF}} \textbf{ if } \mathsf{act} = \text{“}O\text{”} \wedge \mathsf{dp} = n \\
&\qquad \textbf{then if } \mathbf{B}_1 \to \mathfrak{G}_n(\mathbf{S}_1) \\
&\qquad\quad \square \ldots \\
&\qquad\quad \square\ \mathbf{B}_m \to \mathfrak{G}_n(\mathbf{S}_m) \textbf{ fi fi} \\
\mathfrak{G}_n(\textbf{do S od}) &=_{\text{DF}} \textbf{ if } \mathsf{act} = \text{“}O\text{”} \wedge \mathsf{dp} = n \\
&\qquad \textbf{then } \mathsf{dp} := n + 1; \\
&\qquad\quad \textbf{while } \mathsf{act} = \text{“}O\text{”} \wedge \mathsf{dp} = n + 1 \textbf{ do} \\
&\qquad\qquad \mathfrak{G}_{n+1}(\mathbf{S}) \textbf{ od fi}
\end{aligned}$$

This definition implies that as soon as act is set to “Z” no further statements in the action system will be executed and the current action system will (eventually) terminate as all the “pending” calls to A unwind with no side effects. This ensures the correct operation of the “halting” action. Execution then continues with the statements after the action system (if any). The strings “A_1”, ..., “A_n”, “O” and “Z” represent a suitable set of $n + 2$ distinct constant values.

The following are true for all syntactically correct action systems:

- The procedure A is never called with act equal to “Z” (or in fact any value other than “A_1”, ..., “A_n”);
- The use of a local variable for dp means that $\mathsf{dp} = 0$ whenever A is called;
- Because dp is restored after each call of A, a loop can only be terminated by a suitable **exit** statement within it, or a call of Z either directly or indirectly via a called action. If there is no **call** Z in the system, then any action call can be treated as an ordinary simple statement.

The assignments $\mathsf{act} :=$ “O” at the beginning of each action body allow us to distinguish the following three cases depending on the value of act:

1. Currently executing an action: $\mathsf{act} =$ “O”;
2. About to call an action other than Z: $\mathsf{act} \in \{\text{“}A_1\text{”}, \ldots, \text{“}A_n\text{”}\}$;
3. Have called the terminating action, all outstanding recursive calls in this system are terminated without any statements being executed: $\mathsf{act} =$ “Z”.

The ability to distinguish these cases is convenient for reasoning about action systems and proving the correctness of transformations on action systems.

Definition 4.2 An action is *regular* if every execution of the action leads to an action call. (This is similar to a regular rule in a Post production system [32]).

Definition 4.3 An action system is regular if every action in the system is regular. Any algorithm defined by a flowchart, or a program which contains labels and **goto**s but no procedure calls in non-terminal positions, can be expressed as a regular action system. A regular action system can only be terminated by a call to Z, and no action call in a regular action system can ever return.

Definition 4.4 An action system is *recursive* if it contains no calls to Z (apart from those in nested action systems). An action call in a recursive action system will always return.

Regular action systems have a number of useful properties: for example, code immediately following an action call can be deleted. In translating from assembler to WSL, FermaT ensures that the result is a regular action system and that all subsequent transformations preserve this property.

4.3 The Procedures Level

The procedures level of WSL adds procedures with parameters which are called by value or by value-result. Here the value of the actual parameter is copied into a local variable which replaces the formal parameter in the body of the procedure. For result parameters, the final value of this local variable is copied back into the actual parameter. In this case the actual parameter must be a variable or some other object (eg an array element) which can be assigned a value. Such objects are called "L-values" because they can occur on the left of assignment statements.

A **where** statement contains a main body plus a collection of (possibly mutually recursive) procedures:

begin S **where**
  proc $F_1(x_1 \dots) \equiv \mathbf{S}_1$**.**
  ...
end

Recursive procedures and action systems are similar in several ways, the differences are:

- There is nothing in a **where** statement which corresponds to the Z action: all procedures must terminate normally (and thus a "regular" set of recursive procedures could never terminate);
- Procedure calls can occur anywhere in a program, for example in the body of a **while** loop: action calls cannot occur as components of simple statements.

An action system which does not contain calls to Z can be translated to a **where** clause (the converse is only true provided no procedure call is a component of a simple statement).

4.4 The Functions Level

A **where** clause may also include functions and boolean functions with parameters. In [36] these are defined in terms of their "procedural equivalent" (a procedure which stores the result of the function in a suitable variable) and they are allowed to use local variables and have side effects, using the notation $\lceil \mathbf{S};\ e \rfloor$ for expressions with side effects. For example, the statement $x := \lceil \mathbf{S};\ e \rfloor$ is equivalent to $\mathbf{S};\ x := e$.

5 History of the Theory

Our transformation theory developed in roughly the following stages:

1. Start with a very simple and tractable kernel language;
2. Develop proof techniques based on set theory and mathematical logic, for proving the correctness of transformations in the kernel language;
3. Extend the kernel language by definitional transformations which introduce new constructs (the result is the WSL wide spectrum language);

4. Develop a catalogue of proven WSL transformations: each transformation is proved correct by appealing to already proven transformations, or by translating to the kernel language and applying the proof techniques directly.
5. Tackle some challenging program development and reverse engineering tasks to demonstrate the validity of this approach;
6. Extend WSL with constructs for implementing program transformations (the result is called $\mathcal{META}$WSL);
7. Implement an industrial strength transformation engine in $\mathcal{META}$WSL with translators to and from existing programming languages. This allowed us to test our theories on large scale legacy systems (including systems written in IBM Assembler: see [38,40]).

5.1 Translating from Assembler to WSL

The aim in developing an assembler to WSL translator is to capture the semantics of the assembler program (as far as possible) into WSL without worrying about efficiency, structure or redundancy. Each register and flag is represented as a WSL variable and each assembler instruction is translated to a single action in a huge action system. The "fall through" from one instruction to the next is implemented with an explicit **call** statement.

The translation of each instruction is designed to capture *all* the effects of the instruction: regardless of whether the result is needed or not. For example, if an instruction sets the zero flag (zf), then WSL code to assign to zf is generated even if the next instruction immediately overwrites it.

Demonstrating the correctness of the translator then reduces to the task of demonstrating that each instruction is translated correctly: if this is the case, then the translation of an entire program will be correct *by construction* (due to the *replacement property* of WSL, see Section A.4.) Note that without a formal semantics for assembler it is impossible to *prove* the correctness of the translator: but the organisation of the translator allows the developers to concentrate on correct translation of each instruction without needing to consider combinations of instructions (apart from specific cases such as self-modifying code and jump tables).

5.2 Automated Transformation

All of the transformations in FermaT are implemented in an extension of WSL, called $\mathcal{META}$WSL. This adds new constructs to WSL for pattern matching and pattern filling and new looping constructs for walking over the parse tree of a program and executing the loop body on selected statements, expressions or conditions. Within $\mathcal{META}$WSL the condition @Trans?(name) tests if the given transformation is valid at the current position and the statement @Trans(name, data) will apply the given transformation at the current position, passing data as the additional argument. (For example, data might be the new name to use for a procedure renaming transformation).

A $\mathcal{META}$WSL program which only modifies the current program via @Trans calls after first testing the transformation via @Trans? will then form the implementation of a new transformation which is guaranteed to be correct: this due to the *replacement property* of WSL. Over many years of development of the transformation theory, the various versions of FermaT and case studies with many different systems we have developed a large number of transformations which are known to always "improve" the program whenever they are applied. This improvement is almost always a reduction in either size or complexity (usually both). Occasionally a transformation will be applied which increases the program size or complexity because it is known to make possible further simplifications, but because we know that each step or group of steps is a definite improvement, we can iterate the process until no further improvement is possible and avoid the problem of an infinite loop (such as repeated application of a transformation and its inverse).

The transformations are collected together into "meta transformations" (transformations which primarily operate by invoking other transformations). Ultimately this process led to the development of a single Fix_Assembler transformation which automates the bulk of the migration process. As a result, a directory full of assembler files can be migrated via the `x86toc` script with a single command.

6 The Project

Tenovis offers modular communication solutions focusing on the convergence of telecommunications and the internet. At Tenovis, 200,000 clients throughout Europe are serviced by some 6,000 employees. In 2002, Tenovis generated revenue of about 950 million euros.

One of Tenovis' products is a PBX system (Private Branch eXchange) running on four different hardware platforms and installed in sites spread across 18 countries. The system contains about 800,000 lines of C code, and 544,000 lines of 186 assembler: with the assembler split over 318 source files.

Software Migrations Ltd was tasked with migrating all of the assembler code to high-level, structured, maintainable C code, suitable for porting to a more modern processor and also suitable for implementing a backlog of enhancements.

6.1 Case Study

An initial case study involved migrating a single 3,000 line source file to C. We developed a simple x86 to WSL translator which was limited to translating those instructions actually used in the test file. This simply translated each x86 instruction to a block of WSL code which implements all the effects of the instruction (including setting flags and registers).

The whole source file was translated to a single Action System with one action per block of instructions and with explicit action calls to implement the "fall through" from one block to the next. We then applied a sequence of automatic transformations to restructure and simplify the WSL code. These transformations included control flow analysis (to restructure the action system into structured loops and conditional statements), dataflow analysis (to eliminate redundant register and flag assignments), constant propagation, and many other operations. The set of transformations was based on the transformations used to process IBM assembler, with only a few modifications. See below for the details.

The resulting structured WSL code was translated to C using an existing WSL to C translator. This translator was developed for IBM assembler to C migration [38,40] but only needed slight modifications to cope with the different register and flag names.

The case study was very successful: some reviewers were quite astonished with the quality of the code samples and commented "Hey, this really looks like C!". This confirms our view that general-purpose FermaT transformations can be applied to a new source language with little difficulty.

6.2 Mini Call Control

The next stage in the project was the migration of a subset of the system, which formed a "mini" call control. This consisted of 67,000 lines of assembler in 41 source files.

The x86 assembler to WSL translator was rewritten to be table driven: the table lists each assembler mnemonic or subroutine followed by the WSL translation for that instruction or subroutine. See Figure 2 for an extract from the translation table.

Note that subroutine implementations (`ladpk` and `tstbt`) are freely intermixed with assembler instruction implementations (`cmp`, `jmp` etc.). When the translator encounters a "`call far ptr`" instruction it checks for the name of the subroutine in the translation table.

```
  ladpk  ->  ax[1] := a[adtn1].tprkn;

# bx contains pointer to a location,
# dx contains a byte offset plus a bitmask.

  tstbt -> IF !XC tstbt(bx, dx) THEN zf := 1 ELSE zf := 0 FI;

  cmp   -> IF $par1$ = $par2$ THEN zf := 1 ELSE zf := 0 FI;
           IF $par1$ < $par2$ THEN cf := 1 ELSE cf := 0 FI;

  jmp   -> CALL $par1$;

  jnz   -> IF zf = 0 THEN CALL $par1$ FI;

  mov   -> $par1$ := $par2$;
```

Figure 2: Part of the `x86.tab` table

An assembler instruction consists of an optional label, a mnemonic, a list of zero or more parameters and an optional comment. The parameters are translated to WSL expressions or lvalues as appropriate and the mnemonic is looked up in the table. The corresponding WSL code is extracted from the table and the "tags" `$par1$` etc. are replaced by the WSL code. A **call** to the next action is appended and the resulting WSL code block forms the action body corresponding to this instruction.

Some additional requirements were identified as a result of the case study:

1. The customer already had C header files for the assembler data structures. These were used by the existing C code, so it was important for the new code to use the same header files. This involved writing a parser for the header files so that FermaT could generate code appropriate to the declared variable types: for example, casting integers to pointers and vice versa where appropriate. The aim was to ensure that the generated C code compiled with no errors or warnings: the lack of warnings about automated type casting ensured that FermaT had correctly understood the types of all variables.
2. The customer also had available a "function parameter table". This listed which registers and flags were inputs and outputs to each assembler subroutine. The aim was to generate C functions which took these registers and flags as parameters. For the most commonly used functions, a separate table listed the required C type for each parameter.
3. The customer wanted to translate certain subroutine calls directly to their implementation in a high level form. For example, the assembler subroutine `ltnrb` takes an index number in the `al` register and puts the address of the record `ram[al]` in the `bx` register. This function could therefore be translated directly to the assignment bx := ram[ax[1]]. Dataflow analysis can then replace subsequent references to bx by the expression ram[ax[1]] and then delete the assignment to bx. As a result, a code sequence such as:

   ```
   mov al,rufnr
   call far ptr ltnrb
   cmp byte ptr [bx+rwter],0
   ```

   will be transformed to the test:

   ```
   if (ram[rufnr].rwter == 0)
   ```

   The dataflow analysis is applied to the whole program and "propagates" the value of each register as far forwards as possible: replacing references to a register by its current value

where available. So the assignment to `al` could be arbitrarily far from the call to `ltnrb`, which in turn could be arbitrarily far from the reference to `bx` in the `cmp` instruction.

A list of these "inlined" functions was added to the translation table.

4. The routines `tstbt`, `setbt` and `resbt` are used to test, set and clear a single bit in a bitfield. The "address" of the bit is passed in `dx` in the form of a mask plus a byte offset. All the bits can be accessed directly in C using a symbolic field name. In this case, the solution is to translate `tstbt` to a function call, but also implement a customer-specific transformation which looks for `tstbt` calls with a bitfield argument (where dataflow analysis has replaced the register by its known value). These can be replaced by the equivalent record field access. For example:

```
mov al,rufnr
call far ptr ltnrb
mov dx,ertbf
call far ptr tstbt
```

will be translated to:
ax[1] := rufnr;
bx := ram[ax[1]];
dx := ertbf;
if !XC tstbt(bx, dx) **then** zf := 1 **else** zf := 0 **fi**;

Subsequent dataflow analysis will transform the WSL to:
ax[1] := rufnr;
bx := ram[rufnr];
dx := ertbf;
if !XC tstbt(ram[rufnr], ertbf) **then** zf := 1 **else** zf := 0 **fi**;

A customer-specific transformation will recognise that ertbf is a field name and turn the WSL condition !XC tstbt(ram[rufnr], ertbf) into ram[rufnr].ertbf If there are no further references to these values of ax[1], bx and dx, then the C code generated will be:

```
if (ram[rufnr].ertbf == 1)
```

5. Stack usage: registers are frequently pushed and popped from the stack, but (apart from two cases for which manual editing was used) the stack is not modified directly. Where possible, the stack operations will be eliminated (by replacing the register by its value, and so avoiding modifying the register in between the push and pop), or the registers will be saved in local variables, or as a last resort the C functions `__Save` and `__Restore` are used to save data on a global stack implemented as a linked list of records.

6. Jump tables are used in this format:

```
move bx,offset soztb
dec al
mov ah,0
add bx,ax
add bx,ax
add bx,ax
add bx,ax
add bx,ax
jmp bx

soztb:
jmp far ptr label1
jmp far ptr label2
jmp far ptr label3
```

...

This sets bx to soztb+5*(al−1) and then branches to that address. So if al = 1 it branches to label1, if al = 2 it branches to label2 etc. This should be translated to a switch/case statement on al

7. Switch/case statements should be used instead of nested if constructs, where possible.
8. For this application, segment addressing can be ignored since all data can be accessed directly in the C.

For this stage of the project the WSL to C translator was completely rewritten in about five man days work, including testing. This translator is implemented in $\mathcal{META}$WSL and so can make use of the powerful constructs for analysing and manipulating WSL code.

Figure 2 shows part of the translation table with the WSL code for various assembler instructions and subroutines. For example, ladpk is not an instruction mnemonic but the name of an assembler subroutine which is translated directly to the corresponding WSL. Symbols such as $par1$ are replaced with WSL translations of the corresponding parts of the assembler instruction.

The array $a[]$ represents the memory of the system, so the WSL expression a[adtn1].tprkn dereferences the pointer adtn1 to get a structure and then extracts the tprkn field from the structure. Array access notation is also used to extract the individual bytes from the registers ax, bx, cx and dx which are 16 bits wide. ax[1] is the low byte (represented as al in the source) and ax[2] is the high byte.

Below is an extract from one of the assembler source files. This is part of the code for the htst_iwl subroutine together with declarations of the external data and subroutines it uses. It calls ladpk and then carries out various tests on al in order to determine which subroutine to call next.

```
        extrn   dsaft           :abs
        extrn   adtn1           :word
        extrn   hrfft           :abs
        extrn   oldgs           :byte

no_pick:
        mov     dx,dsaft
        mov     bx,adtn1
        call    far ptr tstbt
        jnz     htst_irf_ret
        mov     bx,adtn1
        mov     dx,hrfft
        call    far ptr tstbt
        jz      htst_irf
        mov     oldgs,0
        call    far ptr hwal
        jnz     htst_irf_ret
        jmp     htst_irf
htst_irf_ret:
        ret
```

The raw WSL translation of this extract is:

```
no_pick ≡ dx := dsaft;
          bx := adtn1;
          if !XC tstbt(bx, dx)
            then zf := 1 else zf := 0 fi;
          if zf = 0 then call htst_irf_ret fi;
          bx := adtn1;
          dx := hrfft;
          if !XC tstbt(bx, dx)
```

```
            then zf := 1 else zf := 0 fi;
          if zf = 1 then !P htst_irf( var os);
                         call Z fi;
          oldgs := 0;
          hwal_zf := NOT_USED;
          !P hwal( var hwal_zf, os);
          zf := hwal_zf;
          if zf = 0 then call htst_irf_ret fi;
          !P htst_irf( var os);
          call Z;
          call htst_irf_ret end
htst_irf_ret ≡ call Z;
               call Z end
```

Note that the `tstbt` call has been translated to a WSL function call. Dataflow analysis will replace the references to bx and dx by their actual values, after which a specialised transformation will replace the tstbt call by the appropriate field reference. The assignment hwal_zf := NOT_USED is there to tell FermaT that the initial value of hwal_zf is not used by the hwal procedure.

The restructuring process applies over 100 separate WSL transformations to produce the following restructured WSL program:

```
if a[adtn1].dsaft = 0 ∧ a[adtn1].hrfft = 0
  then !P htst_irf( var os)
elsif a[adtn1].dsaft = 0
    then oldgs := 0;
         hwal_zf := NOT_USED;
         !P hwal( var hwal_zf, os);
         if hwal_zf ≠ 0
           then !P htst_irf( var os) fi fi
```

This is then translated to the following C code:

```
void
no_pick()
{
 if ((adtn1->dsaft == 0
        && adtn1->hrfft == 0))
 {
         htst_irf();
 }
 else if (adtn1->dsaft == 0)
 {
         oldgs = 0;
         hwal_zf = hwal();
         if (hwal_zf != 0)
         {
                 htst_irf();
         }
 }
 return;
}
```

The main problem uncovered by the mini call control migration was that FermaT assumed that all external procedures only modified their output parameters. In fact, subroutines save and restore some registers and clobber other registers and there is no consistency or simple rule to determine which registers are modified (other than a global analysis of the whole system). Our solution was

to modify the semantics of the !P external call to clobber all but a small list of variables.

Altogether there were five iterations of the mini call control code with the customer examining the code after each iteration and giving feedback. After fixing the problems mentioned above and making various changes to the style of the generated C code, the code from the fifth iteration was compiled and installed on the hardware where it performed flawlessly.

6.3 Results

The final stage was to migrate the entire system. Only three iterations (out of the planned five) were required to iron out any remaining problems which were not exposed by the mini call control migration.

The final migration process was carried out on a 2.6GHz PC with 512Mb of RAM. All 318 source files were processed successfully in 1 hour, 27 minutes of CPU time (1 hour 30 minutes elapsed time) for an average of 16.5 seconds per file. A total of 1,436,031 transformations were applied, averaging 4,500 transformations per file and 275 transformations per second. The most complex file contained 8,348 lines of source which required 72,393 transformations and took 370 seconds of CPU time but needed less than 42Mb of RAM. The 544,454 lines of assembler source were migrated to 506,672 lines of C code with a further 37,047 lines of header files.

Several bugs were uncovered in the system as a result of the migration process. The C code, while not exactly a strongly-typed language, does impose some typing constraints. As a result, some type errors can be detected at compile time which are undetectable in the assembler: for example, the assembler can load any values into bx and dx before calling tstbt. After transformation, the value in dx is converted to a field name: if this is not a valid field for the record, then the C compiler gives an error.

Tenovis have built a PC-based test environment for the transformed software. The goal is to implement a "soft switch" where the Call Control software on a PC is linked to the peripherals on the PBX — and the system behaves exactly like it did with the embedded assembler call control.

At the time of writing, the migrated code has been manually examined and signed off by Tenovis and is currently undergoing final testing before release.

6.4 Cost Savings

Prior to the project the customer had received a quote for a complete manual translation of the software, after the data structures had been designed, which came to 67 man months.

The effort invested in the project by the customer was as follows (note that there were five iterations of the mini system):

Task	Man Days
C header files	10
Code design rules	3
Tool creation/adaptation	5
Code Review	4 per iteration
Code Compilation	1 per iteration
Testing (last 2 iterations)	20
TOTAL	63

In addition there was a small amount of management time plus a final regression test of the whole system, which is currently in progress.

Software Migrations Ltd. invested 52 man days plus some management time: this included developing the 186 to WSL translator and redeveloping the WSL to C translator.

So the total effort for an automated migration of over half a million lines of assembler was less than 6 1/2 man months: which is less than 10% of the estimate for a manual migration!

It should be noted that much of the effort was expended on the mini call control system: developing header files, developing translators, tuning the results of the transformation process and testing the result. Once the transformation process had been tuned and tested on the 67,000 line mini system, scaling up to the complete 544,000 line system took very little extra effort. So we anticipate that the cost savings would be even greater for larger legacy systems.

7 Conclusion

This paper provides a brief introduction to the transformation theory behind FermaT and some of the methods used to prove the correctness of transformations. We also describe a major migration project: translating over half a million lines of assembler to C, which made extensive use of automated program transformation. This project, and other case studies with IBM 370 Assembler [38, 40], demonstrate that the FermaT technology is a practical solution to automated migration and reengineering for diverse programming languages.

7.1 Advantages of Automated Reengineering

- Scalability: once the transformation rules have been refined on the subsystem to generate exactly the code required by the customer, the same rules can be applied to the whole system very quickly. So the amount of work required to reengineer a large system is less than linear in the size of the system (i.e. a system 10 times larger takes much less than 10 times the effort to migrate). So automated reengineering is particularly suited to large legacy systems.
- Rapid turnaround of the subsystem: new transformations can be developed in a matter of days and the complete subsystem reprocessed in at most a few hours.
- Customisability: due to the rapid turnaround, it is possible to carry out a number of iterations on the subsystem and customise the transformation process to deal with customer-specific macros, coding conventions, programming quirks etc. and to generate high-quality target code which matches the customer's programming conventions.
- Low resource requirement: only a small team of programmers are required to analyse and review the different iterations of the subsystem. Final testing of the migrated system is still a major task: comparable with the final testing stage of any new release of the product. But as such, this can be fitted into the normal product cycle.
- Low impact on ongoing development: there is no need to "freeze" ongoing development during most of the migration process, and development resources are not tied up by the migration process. Once the transformation process has been tuned to satisfaction, the system can be temporarily "frozen", the latest copy of the system can be put through the migration process, and testing of the new system can start in a matter of days.
- Migration from assembler to a HLL allows the possibility of porting to a different machine and operating system (eg, systems can be moved from expensive mainframes onto cheaper Unix or Linux boxes, or from outdated embedded system processors onto more modern and powerful processors).
- Reengineering enables major enhancements to be carried out with greater productivity.
- Migration removes dependence on limited and diminishing resources such as assembler programming talent, or expensive resources such as mainframe CPU cycles.

The FermaT Transformation System is available under the GNU GPL (General Public Licence) and can be downloaded from:

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

Acknowledgements

The work described in this paper has been partly funded by Tenovis, partly by Software Migrations Ltd., and partly by De Montfort University.

References

[1] Ira D. Baxter and & Michael Mehlich, "Reverse Engineering is Reverse Forward Engineering," in *Fourth Working Conference on Reverse Engineering*, IEEE Computer Society, Amsterdam, The Netherlands, Oct., 1997.

[2] 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⟩.

[3] R. S. Arnold, *Software Reengineering*, IEEE Computer Society, 1993.

[4] J. Arsac, "Syntactic Source to Source Transforms and Program Manipulation," *Comm. ACM* 22 #1 (Jan., 1979), 43–54.

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

[6] F. L. Bauer, B. Moller, H. Partsch & P. Pepper, "Formal Construction by Transformation—Computer Aided Intuition Guided Programming," *IEEE Trans. Software Eng.* 15 #2 (Feb., 1989).

[7] K. Beck, *Extreme Programming Explained*, Addison Wesley, Reading, MA, 1999.

[8] Keith H. Bennett, "Do Program Transformations Help Reverse Engineering?," *International Conference on Software Maintenance (ICSM), Washington DC, USA* (Nov., 1998).

[9] Scott Burson, Gordon B. Kotik & Lawrence Z. Markosian, "A Program Transformation Approach to Automating Software Re-engineering," *14th Annual International Computer Software and Applications Conference (COMPSAC)* (1990).

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

[11] A. Eastwood, "It's a hard sell - and hard work too. (software reengineering)," *Computing Canada* 18 #22 (1992), 35.

[12] G. C. Gannod & H. C. Cheng, "Using Informal and Formal Techniques for Reverse Engineering of C Programs," in *Proceedings of the Third Working Conference on Reverse Engineering, 8–10th November*, IEEE Computer Spciety, Monteray, California, 1996, 249–258.

[13] A. Hall, "Seven Myths of Formal Methods," *IEEE Software* 7 #5 (Sept., 1990), 11–19.

[14] 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 #8 (Aug., 1987), 672–686.

[15] Capers Jones, *Programming Languages Table*, Release 8.2, Software Productivity Research, Inc., Mar., 1996.

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

[17] Stephen Klusener, Ralf Lämmel & Chris Verhoef, "Architectural Modifications to Deployed Software," 2002, ⟨http://www.cs.vu.nl/~x/am⟩.

[18] D. E. Knuth, "Structured Programming with the GOTO Statement," *Comput. Surveys* 6 #4 (1974), 261–301.

[19] Ralf Lämmel & Chris Verhoef, "Cracking the 500-Language Problem," *IEEE Software* 18 #6 (2001), 78–88.

[20] Ralf Lämmel & Chris Verhoef, "Semi-automatic Grammar Recovery," *Software—Practice and Experience* 31 #15 (2001), 1395–1438.

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

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

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

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

[25] C. C. Morgan & K. Robinson, "Specification Statements and Refinements," *IBM J. Res. Develop.* 31 #5 (1987).

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

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

[28] J. M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Trans. Software Eng.* SE-10 #5 (Sept., 1984).

[29] James M. Neighbors, "Draco: A Method for Engineering Reusable Software Systems," in *Software Reusability – Concepts and Models*, Ted J. Biggerstaff & Alan J. Perlis, eds. #I, ACM Press, 1989, 295–319.

[30] R. Paige, "Future Directions In Program Transformations," *ACM Computing Surveys* 28A #4 (1996), 170–174.

[31] H. Partsch & R. Steinbrügen, "Program Transformation Systems," *Computing Surveys* 15 #3 (Sept., 1983).

[32] E. L. Post, "Formal Reduction of the General Combinatorial Decision Problem," *Amer. J. Math.* (1943).

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

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

[35] D. Taylor, "An Alternative to Current Looping Syntax," *SIGPLAN Notices* 19 #12 (Dec., 1984), 48–53.

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

[37] 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⟩.

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

[39] M. Ward, "Recursion Removal/Introduction by Formal Transformation: An Aid to Program Development and Program Comprehension," *Comput. J.* 42 #8 (1999), 650–673, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/recursion-t.ps.gz⟩ doi:10.1093/comjnl/42.8.650.

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

[41] M. Ward, "A Definition of Abstraction," *J. Software Maintenance: Research and Practice* 7 #6 (Nov., 1995), 443–450, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/abstraction-t.ps.gz⟩ doi:10.1002/smr.4360070606.

[42] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," *IEEE Trans. Software Eng.* 22 #9 (Sept., 1996), 665–686, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/sw-alg.ps.gz⟩ doi:doi.ieeecomputersociety.org/10.1109/32.541437.

[43] 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⟩.

[44] N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM* 14 #4 (1971), 221–227.

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

[46] H. Yang, "The Supporting Environment for a Reverse Engineering System—The Maintainer's Assistant," *Conference on Software Maintenance, Sorrento, Italy* (Dec., 1991).

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

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

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

A Appendix: The Kernel Language and Proof Methods

In this Appendix we give the detailed definitions and theorems for the results which were informally discussed in Section 3. We also give two important theorems: the *Representation Theorem* which shows how to transform any WSL program into an equivalent specification statement, and the *Recursive Implementation Theorem* which provides a general method for transforming a specification into an equivalent recursive implementation.

A.1 Infinitary Logic

The theoretical work on which FermaT is based originated in research on the development of a language in which proofs of equivalence for program transformations could be achieved as easily as possible for a wide range of constructs.

Expressions and conditions (formulae) in WSL are taken directly infinitary first order logic [16]. The logic adds one extension to ordinary first order logic: if we have the countable sequence of formulae $\mathbf{Q}_0, \mathbf{Q}_1, \ldots$ then the infinite conjunction:

$$\bigwedge_{n<\omega} \mathbf{Q}_n$$

is true precisely when every formula $\mathbf{Q}_n$ is true, and false otherwise. It can be written informally as $\mathbf{Q}_0 \wedge \mathbf{Q}_1 \wedge \cdots \wedge \mathbf{Q}_n \wedge \ldots$. Infinite disjunction is defined in terms of negation and infinite conjunction, by analogy with De Morgan's laws for finite formulae:

$$\bigvee_{n<\omega} \mathbf{Q}_n \;=_{\mathrm{DF}}\; \neg \bigwedge_{n<\omega} \neg \mathbf{Q}_n$$

The use of first order logic in WSL means that statements can include existential and universal quantification over infinite sets, and similar (non-executable) operations.

Infinitary first order logic is used in FermaT both to express the weakest preconditions of programs [10] and to define assertions and guards in the kernel language. 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:

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 of toy programs. By using an infinitary logic we can define the weakest precondition of a loop (for example) as an infinite disjunction: "*Either* the loop terminates immediately and satisfies the postcondition, *or* it terminates after one iteration, *or* after two iterations, *or* ...". Then we can reason about loops and recursion, and prove the correctness of transformations involving loops and recursion, without needing to discover loop invariants. (Note that if invariants are available, the information they provide can be made use of.)

The particular infinitary logic we use is $\mathcal{L}_{\omega_1\omega}$ which allows conjunctions and disjunctions over any *countably* infinite sequences of formulae and quantification over *finite* sets of variables. Hence $\mathcal{L}_{\omega_1\omega}$ may be regarded as the "smallest" infinitary logic.

Back and von Wright [5] describe an implementation of the refinement calculus, based on (finitary) higher-order logic using the refinement rule $\forall \mathbf{R}.\, \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ where the quantification is over all predicates (boolean state functions). However, the completeness theorem fails for all higher-order logics. Karp [16] 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 single postcondition $\boldsymbol{x} \neq \boldsymbol{x}'$ is sufficient. Thus we can be confident that the proof method is *complete*: in other words if $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ then there exists a proof of $\mathrm{WP}(\mathbf{S}_1, \boldsymbol{x} \neq \boldsymbol{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \boldsymbol{x} \neq \boldsymbol{x}')$. Basing our transformation theory on any other logic would not provide the two different proof methods we require.

A.2 States, State Spaces and State Predicates

Program refinements and transformations are defined in terms of a program and its initial and final *state spaces*. A state space is a finite set of variables which defines the domain of the initial or final states of the program. The state spaces must be consistent with the constructs appearing in the program: for example, the final state space for a program which ends with the statement $\mathsf{remove}(\boldsymbol{y})$ must not include the variables in the list $\boldsymbol{y}$. For a statement $\mathbf{S}$ with initial state space V and final state space W, the trinary relation $\mathbf{S} : V \to W$ is true whenever V and W are consistent with $\mathbf{S}$.

We will suppose that variables take on values from a set $\mathcal{H}$ which will not be further analysed. A *state* on V and $\mathcal{H}$ is either the special state $\perp$ (which indicates nontermination or error) or a function s from V to $\mathcal{H}$ which gives the value $s(x)$ to the variable $x \in V$. The set of all states on V and $\mathcal{H}$ is denoted $D_{\mathcal{H}}(V)$:

$$D_{\mathcal{H}}(V) \;=_{\mathrm{DF}}\; \mathcal{H}^V \cup \{\perp\}$$

The *proper states* are all the states other than $\perp$. A *state predicate* is a set of proper states (those states which satisfy the predicate), so we can think of $\perp$ as not satisfying any state predicate. The set of all state predicates on V and $\mathcal{H}$ is denoted $E_{\mathcal{H}}(V)$:

$$E_{\mathcal{H}}(V) \;=_{\mathrm{DF}}\; \mathcal{P}(\mathcal{H}^V)$$

i.e. the set of all subsets of $\mathcal{H}^V$.

A state transformation f is a function which maps each initial state s to the set $f(s)$ of possible final states. If $\perp$ is in the set of final states, then we define that every other state is in the set. The initial state $\perp$ always maps to the set of all states (consider a sequential composition of statements: if any statement in the sequence fails to terminate, then so does the whole sequence). More formally, we define:

Definition A.1 *State Transformation:* The set of all *state transformations* from V to W on $\mathcal{H}$ is the set of (total) functions from $D_{\mathcal{H}}(V)$ to $\mathcal{P}(D_{\mathcal{H}}(W))$ which map $\perp$ to $D_{\mathcal{H}}(W)$ and for which the image of any element of $D_{\mathcal{H}}(V)$ which contains $\perp$ also contains every other element of $D_{\mathcal{H}}(W)$. This restriction implies that the image of any element under a state transformation is either $D_{\mathcal{H}}(W)$ or is a state predicate in $E_{\mathcal{H}}(W)$. The set of all state transformations from V to W on $\mathcal{H}$ is denoted $F_{\mathcal{H}}(V, W)$:

$$F_{\mathcal{H}}(V, W) \;=_{\mathrm{DF}}\; \left\{\, f : D_{\mathcal{H}}(V) \to \mathcal{P}(D_{\mathcal{H}}(W)) \;\middle|\; \perp \in f(\perp) \;\wedge\; \forall s \in D_{\mathcal{H}}(V).\, (\perp \in f(s) \Rightarrow f(s) = D_{\mathcal{H}}(W)) \,\right\}$$

an equivalent definition is:

$$F_{\mathcal{H}}(V, W) \;=_{\mathrm{DF}}\; \{\, f : D_{\mathcal{H}}(V) \to (E_{\mathcal{H}}(W) \cup \{D_{\mathcal{H}}(W)\}) \mid f(\perp) = D_{\mathcal{H}}(W) \,\}$$

For each initial state $s \in D_{\mathcal{H}}(V)$, if $\perp \in f(s)$ then the state transformation may not terminate on s and we say f is *undefined* on s. Otherwise, if $\perp \notin f(s)$ and $f(s)$ is non-empty then the state

transformation must terminate on one of the states in $f(s)$. If $f(s) = \varnothing$ we say that the state transformation is *null* on s: the program still terminates even though the set of possible final states is empty.

A state transformation can be thought of as either a specification of a program, or as a (partial) description of the behaviour of a program. If f is a specification, then for each initial state s, $f(s)$ is the allowed set of final states. If $\bot \in f(s)$ then the specification does not restrict the program in any way for initial state s, since every other state is also in $f(s)$.

Similarly, if f is a program description, then $\bot \notin f(s)$ means that the program is guaranteed to terminate in some state in $f(s)$ when started in state s.

Program f *satisfies* specification g precisely when $\forall s.\, (f(s) \subseteq g(s))$.

A program f_2 is a *refinement* of program f_1 if f_2 satisfies every specification satisfied by f_1, i.e. $\forall g.\, (\forall s.\, (f_1(s) \subseteq g(s)) \Rightarrow \forall s.\, (f_2(s) \subseteq g(s)))$. It turns out that refinement and satisfaction, as defined above, are identical relations. If f_2 refines f_1 then f_2 satisfies f_1 (simply put $g = f_1$ in the definition of refinement). Conversely, if f_2 satisfies f_1 then f_2 also satisfies all specifications satisfied by f_1 (by the transitivity of $\subseteq$). So from now on we only talk about refinement, with the understanding that anything we say about refinement applies equally well to satisfaction of specifications.

For the formal definition of refinement we use the shorter of the two equivalent definitions:

Definition A.2 *Refinement:* Given two state transformations $f_1, f_2 \in F_{\mathcal{H}}(V, W)$ we say that f_2 *refines* f_1, or f_1 *is refined by* f_2, and write $f_1 \;\leq\; f_2$ when f_2 is at least as defined and at least as deterministic as f_1. More formally:

$$f_1 \;\leq\; f_2 \iff \forall s \in D_{\mathcal{H}}(V).\, f_2(s) \subseteq f_1(s)$$

Note that if $\bot \in f_2(s)$ then $f_2(s) = D_{\mathcal{H}}(W)$ and $f_1(s)$ can be any set of states.

If we fix on a particular set of values and an interpretation of the symbols of the base logic $\mathcal{L}$ in terms of the set of values, then we can interpret formulae in $\mathcal{L}$ as state predicates and statements of WSL as state transformations. To be precise:

Definition A.3 A *structure* M for $\mathcal{L}$ is a set $\mathcal{H}$ of values together with functions that map the constant symbols, function symbols and relation symbols of $\mathcal{L}$ to elements, functions and relations on $\mathcal{H}$. A structure M for $\mathcal{L}$ defines an *interpretation* of each formula $\mathbf{B}$ as a state predicate $\mathrm{int}_M(\mathbf{B}, V) \in E_{\mathcal{H}}(V)$, consisting of the states which satisfy the formula, and also interprets each statement $\mathbf{S}$ as a state transformation $\mathrm{int}_M(\mathbf{S}, V, W) \in F_{\mathcal{H}}(V, W)$.

For example, if $\mathcal{H} = \{0, 1\}$ and $V = \{x, y\}$, then the state predicate $\mathrm{int}_M(x = y, V)$ is the set of states in which the value given to x equals the value given to y, ie:

$$\{\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 1\}\}$$

A.3 Weakest Preconditions

Dijkstra introduced the concept of weakest preconditions in [10] as a tool for reasoning about programs. For a given program $\mathbf{P}$ and condition $\mathbf{R}$ on the final state space, the weakest precondition $\mathrm{WP}(\mathbf{P}, \mathbf{R})$ is the weakest condition on the initial state such that if $\mathbf{P}$ is started in a state satisfying $\mathrm{WP}(\mathbf{P}, \mathbf{R})$ then it is guaranteed to terminate in a state satisfying $\mathbf{R}$.

For a state transformation f and state predicate e, we define the weakest precondition, $\mathrm{wp}(f, e)$ to be the weakest state predicate such that if s satisfies $\mathrm{wp}(f, e)$ then all elements of $f(s)$ satisfy e. So the weakest precondition is a function $\mathrm{wp} : F_{\mathcal{H}}(V, W) \times E_{\mathcal{H}}(W) \to E_{\mathcal{H}}(V)$ defined as follows:

Definition A.4 *Weakest Precondition of State Transformations:* For any state transformation $f \in F_{\mathcal{H}}(V, W)$ and state condition $e \in E_{\mathcal{H}}(W)$ the *weakest precondition* of f on e is:

$$\mathrm{wp}(f, e) \;=_{\mathrm{DF}}\; \{\, s \in D_{\mathcal{H}}(V) \mid f(s) \subseteq e \,\}$$

Note that since $\bot \in f(\bot)$ for any f, we have $f(\bot) \not\subseteq e$, so $\bot \notin \mathrm{wp}(f, e)$ for any f and e, so $\mathrm{wp}(f, e)$ is indeed in $E_{\mathcal{H}}(V)$.

We define the weakest precondition for statements as a formula of infinitary logic. WP is a function which takes a statement (a syntactic object) and a formula from the infinitary first order logic $\mathcal{L}$ (another syntactic object) and returns another formula in $\mathcal{L}$.

Definition A.5 *For any kernel language statement $\mathbf{S} : V \to W$, 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. $\mathrm{WP}(\mathbf{add}(\boldsymbol{x}), \mathbf{R}) \ =_{\mathrm{DF}} \ \forall \boldsymbol{x}.\, \mathbf{R}$
4. $\mathrm{WP}(\mathbf{remove}(\boldsymbol{x}), \mathbf{R}) \ =_{\mathrm{DF}} \ \mathbf{R}$
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 $(\mu X.\mathbf{S})^0 = \mathbf{abort} = \{\mathbf{false}\}$ 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 $\mathbf{T}$).

Note that the weakest precondition for **remove** is identical to the postcondition: the effect of a $\mathbf{remove}(\boldsymbol{x})$ is to ensure that the variables in $\boldsymbol{x}$ do not appear in W and hence do not appear free in $\mathbf{R}$, but otherwise it has no effect on the state.

With the recursive statement we see the advantage of using infinitary logic: the weakest precondition for this statement is defined as a countably infinite conjunction of weakest preconditions of statements with one fewer recursive constructs (and hence, ultimately, in terms of weakest preconditions of statements with no recursion).

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

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

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

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

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

For the **if** statement:

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

Similarly, for the Dijkstra guarded command:

$$\begin{aligned} &\mathrm{WP}(\mathbf{if}\ \mathbf{B}_1 \ \to\ \mathbf{S}_1 \,\square\, \mathbf{B}_2 \ \to\ \mathbf{S}_2\ \mathbf{fi}, \mathbf{R}) \\ &\quad \iff (\mathbf{B}_1 \vee \mathbf{B}_2) \wedge (\mathbf{B}_1 \Rightarrow \mathrm{WP}(\mathbf{S}_1, \mathbf{R})) \wedge (\mathbf{B}_2 \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})) \end{aligned}$$

There is a fundamental relationship between wp and WP:

Theorem A.6 *Let* $\mathbf{S} : V \to W$ *be a statement with no free statement variables, and* $\mathbf{R}$ *be a formula of* $\mathcal{L}$ *whose free variables are in* W*. Then for any structure* M *for* $\mathcal{L}$ *we have:*

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

The importance of weakest preconditions is that refinement can be characterised using them::

Theorem A.7 *For any state transformations* $f_1, f_2 \in F_{\mathcal{H}}(V,W)$*:*

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

The next theorem shows that instead of quantifying over *all* postconditions it is sufficient to check the wp for two carefully selected postconditions. This result will be extremely important in what follows.

Theorem A.8 *Let* f_1, f_2 *be any state transformations in* $F_{\mathcal{H}}(V,W)$ *and let* $\boldsymbol{x}$ *be any list of all the variables in* W *which are assigned anywhere in* f_1 *or* f_2*. Let* $\boldsymbol{x}'$ *be a list of new variables, of the same length as* $\boldsymbol{x}$*. We may assume, without loss of generality, that* $\boldsymbol{x}' \subseteq V$ *and* $\boldsymbol{x}' \subseteq W$*. Let* e_x *be the state condition in* $E_{\mathcal{H}}(W)$ *defined by:*

$$e_x = \left\{\; s \in E_{\mathcal{H}}(W) \mid\; s \neq \bot \;\wedge\; \forall x \in \boldsymbol{x}.\, s(x) \neq s(x') \;\right\}$$

so e *is the interpretation of the formula* $\boldsymbol{x} \neq \boldsymbol{x}'$*. Let* $e_{true} = D_{\mathcal{H}}(W) - \{\bot\}$ *which is the interpretation of the formula* **true***. Then:*

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

Putting all these results together, we see that given a set Δ of assumptions (expressed as a set of formulae with no free variables), in order to prove that statement $\mathbf{S}_2$ refines $\mathbf{S}_1$, we need to prove that:

1. For any interpretations f_1 of $\mathbf{S}_1$ and f_2 of $\mathbf{S}_2$ which are consistent with Δ we have $f_1 \;\leq\; f_2$. (An interpretation consistent with Δ is a structure in which all the formulae in Δ are interpreted as true. This is called a *model* for Δ).

   By Theorem A.7 it is sufficient to prove:

2. For any interpretation consistent with Δ and any state predicate e we have:

   $$\mathrm{wp}(f_1,e) \subseteq \mathrm{wp}(f_2,e)$$

   By Theorem A.8 it is sufficient to prove:

   $$\mathrm{wp}(f_1,e_{\mathrm{true}}) \subseteq \mathrm{wp}(f_2,e_{\mathrm{true}}) \quad \text{and} \quad \mathrm{wp}(f_1,e_x) \subseteq \mathrm{wp}(f_2,e_x)$$

   for the two special state predicated e_{true} and e_x mentioned in A.8.

   Finally, by Theorem A.6 it is sufficient to prove:

3. The two formulae:

   $$\mathrm{WP}(\mathbf{S}_1,\mathbf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2,\mathbf{true}) \quad \text{and} \quad \mathrm{WP}(\mathbf{S}_1,\boldsymbol{x} \neq \boldsymbol{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2,\boldsymbol{x} \neq \boldsymbol{x}')$$

   can be proved (or deduced) from the set Δ of assumptions.

We have reduced the task of proving the correctness of a refinement to that of proving the validity of two formulae of infinitary logic. If the two formulae can be proved, then the refinement is valid and we write $\Delta \vdash \mathbf{S}_1 \;\leq\; \mathbf{S}_2$.

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

Back and von Wright [5] note that the refinement relation can be characterised using weakest preconditions in *higher order logic* (where quantification over Boolean predicates is allowed). Under their formalism, the program $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$ if the formula $\forall \mathbf{R}.\, \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ is true in finitary higher order logic. 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 $\mathbf{R}$. Proof rules justified by an appeal to WP in *finitary* logic cannot justifiably be 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.

By using a countable infinitary logic and the special postcondition $\boldsymbol{x} \neq \boldsymbol{x}'$ we avoid both of these problems.

A.4 Induction Rules

The weakest precondition approach to proving the correctness of a refinement does have one potential drawback: the formulae we are working with can be infinitely long! However, these formulae do have a very regular structure, which means that it is possible to prove various properties of the formulae by induction. One example is the induction rule for recursion which can be used to show that a particular statement is a valid refinement of a recursive procedure:

Lemma A.9 *The Induction Rule for Recursion:*

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

The condition $\Delta \vdash (\mu X.\mathbf{S})^n \ \leq \ \mathbf{S}'$ is usually proved by induction on n. The base case $n = 0$ is trivial since $(\mu X.\mathbf{S})^0 = \mathbf{abort}$ and **abort** is refined by any statement. For the induction step, we assume that $\Delta \vdash (\mu X.\mathbf{S})^n \ \leq \ \mathbf{S}'$ and use this to prove that $\Delta \vdash (\mu X.\mathbf{S})^{n+1} \ \leq \ \mathbf{S}'$:

$$(\mu X.\mathbf{S})^{n+1} = \mathbf{S}[(\mu X.\mathbf{S})^n / X] \ \leq \ \mathbf{S}[\mathbf{S}'/X]$$

by the induction hypothesis and the replacement property (see below).

Lemma A.10 *The replacement property:* Any component of a statement can be replaced by a refinement of the component to give a refinement of the whole statement.

Proof: The proof is by induction on depth of nesting of recursive statements and induction on the structure of $\mathbf{S}$. Each induction step is proved from the weakest preconditions. For the recursive statement $\mathrm{WP}((\mu X.\mathbf{S}), \mathbf{R}) = \bigvee_{n<\omega} \mathrm{WP}((\mu X.\mathbf{S})^n, \mathbf{R})$ by definition, and each $(\mu X.\mathbf{S})^n$ has a lower recursion nesting than $(\mu X.\mathbf{S})$ so the induction hypothesis applies even though $(\mu X.\mathbf{S})^n$ may be syntactically larger than $(\mu X.\mathbf{S})$.

The replacement property is essential for any notion of refinement, it might seem to be an obvious property, but it does not hold for some popular languages. For example: in C the statements:

```
x = 2*x + 1;
```

and

```
x = 2*x; x = x + 1;
```

are equivalent, but the statements:

```
if (y = 0)
  x = 2*x + 1;
```

and

```
if (y = 0)
  x = 2*x; x = x + 1;
```

are *not* equivalent (since the scope of an `if` statement is a single statement or block).

It is the replacement property of WSL which makes our approach to migration possible: the transformation system applies thousands of transformations in turn to components of the program. Provided each transformation is valid and implemented correctly, the replacement property ensures that the resulting transformed program is still equivalent to the initial program.

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 A.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 $\Delta \vdash \mathbf{S} \leq \mathbf{S}'$. (A statement has *bounded nondeterminacy* if every component specification statement has a finite set of final states for each initial state on which it terminates.)

A corollary of this lemma is that if $\Delta \vdash \mathbf{S}_1^n \approx \mathbf{S}_2^n$ for all $n < \omega$ then $\Delta \vdash \mathbf{S}_1 \approx \mathbf{S}_2$. Because this lemma is so useful, we will assume that all statements have bounded nondeterminacy in the rest of this paper.

The following is one of many transformations which can be proved from the general induction rule. If a statement $\mathbf{S}_1$ appears outside a recursive procedure and also immediately before each recursive call, then we can move it to the start of the procedure body.

Transformation 11 *Expand Recursion*
If $\mathbf{S}_1$ has no free occurrences of X then:

$$\Delta \vdash \mathbf{S}_1;\ (\mu X.\mathbf{S}[\mathbf{S}_1;\ X/X]) \approx (\mu X.(\mathbf{S}_1;\ \mathbf{S}))$$

We will prove that $\mathbf{S}_1;\ (\mu X.\mathbf{S}[\mathbf{S}_1;\ X/X])^n \approx (\mu X.(\mathbf{S}_1;\ \mathbf{S}))^n$ for all n by induction.

$$\begin{aligned}
\mathbf{S}_1;\ (\mu X.\mathbf{S}[\mathbf{S}_1;\ X/X])^{n+1} &\approx \mathbf{S}_1;\ \mathbf{S}[\mathbf{S}_1;\ X/X][(\mu X.\mathbf{S}[\mathbf{S}_1;\ X/X])^n/X] \\
&\approx \mathbf{S}_1;\ \mathbf{S}[\mathbf{S}_1;\ (\mu X.\mathbf{S}[\mathbf{S}_1;\ X/X])^n/X] \\
&\approx \mathbf{S}_1;\ \mathbf{S}[(\mu X.(\mathbf{S}_1;\ \mathbf{S}))^n/X]
\end{aligned}$$

by the induction hypothesis.

$$\begin{aligned}
&\approx (\mathbf{S}_1;\ \mathbf{S})[(\mu X.(\mathbf{S}_1;\ \mathbf{S}))^n/X]
\end{aligned}$$

Since there are no free occurrences of X in $\mathbf{S}_1$

$$\begin{aligned}
&\approx (\mu X.(\mathbf{S}_1;\ \mathbf{S}))^{n+1}
\end{aligned}$$

Transformation 12 *Fold/Unfold:*
For any $\mathbf{S}\colon V \to V$:

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

The Induction rule is used to prove some general recursion removal and recursion introduction theorems in [39].

A.5 The Representation Theorem

The next theorem shows that *any* WSL statement can be transformed directly into a single specification statement, with a guard if necessary.

Theorem A.12 *The Representation Theorem*
Let $\mathbf{S}\colon V \to V$, be any kernel language statement and let $\boldsymbol{x}$ be a list of all the variables in V. Then for any countable set Δ of sentences:

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

For a general statement $\mathbf{S}\colon V \to W$ it is sufficient to add a single remove statement:

Corollary A.13 Let $\mathbf{S}\colon V \to W$, be any kernel language statement and let $\boldsymbol{x}$ be a list of all the variables in W. 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 $\boldsymbol{y}$ be a list of the variables removed by $\mathbf{S}$, so $\boldsymbol{x} \cap \boldsymbol{y} = \varnothing$ and $\boldsymbol{x} \cup \boldsymbol{y} = V$. Then for any countable set Δ of sentences:

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

This theorem shows that the specification statement is sufficiently powerful to specify *any* computer program we may choose to develop. It would also appear to solve all reverse engineering problems at a stroke, and therefore be a great aid to software maintenance and reverse engineering. But 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 [41] 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.

The theorem is of considerable theoretical value however, firstly in showing the power of the specification statement: in particular it tells us that a single specification statement is certainly sufficiently expressive for writing the specification of *any* computer program whatsoever (including languages with infinitary predicates: though these have yet to be implemented).

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

Corollary A.14 *For any statement* $\mathbf{S}$*:*

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

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

Proof: Convert $\mathbf{S}$ to its specification equivalent using Theorem A.12, take the weakest precondition for $\mathbf{R}$ and simplify the result. ■

The point of this corollary is that it expresses the weakest precondition of a statement for *any* postcondition as a simple formula containing a single occurrence of postcondition itself plus some weakest preconditions of fixed formulae.

A.6 Recursive Implementation Theorem

Our next result is an important theorem on the recursive implementation of statements. It provides a general method for transforming a specification into an equivalent recursive statement.

These transformations can be used to implement recursive specifications as recursive procedures, to introduce recursion into an abstract program to get a "more concrete" program (i.e. closer to a programming language implementation), and to transform a given recursive procedure into a different form. The theorem is used in the algorithm derivations of [37,42] and [36].

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

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

The expression $\boldsymbol{t}$ need not be integer valued: any set Γ which has a well-founded order $\preccurlyeq$ is suitable. To prove that the value of $\boldsymbol{t}$ is reduced it is sufficient to prove that if $\boldsymbol{t} \preccurlyeq t_0$ initially, then the assertion $\{\boldsymbol{t} \prec t_0\}$ 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 A.15 *The Recursive Implementation Theorem*
If $\preccurlyeq$ is a well-founded partial order on some set Γ and $\boldsymbol{t}$ is a term giving values in Γ and t_0 is a variable which does not occur in $\mathbf{S}$ then if

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

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