$\mathcal{META}$WSL and Meta-Transformations in the FermaT Transformation System

Martin Ward and Hussein Zedan
Software Technology Research Laboratory, De Montfort University, Leicester, UK
mward@dmu.ac.uk
http://www.cse.dmu.ac.uk/∼mward/

Abstract

A program transformation is an operation which can be applied to any program (satisfying the transformations applicability conditions) and returns a semantically equivalent program. In the FermaT transformation system program transformations are carried out in a wide spectrum language, called WSL, and the transformations themselves are written in an extension of WSL called $\mathcal{META}$WSL which was specifically designed to be a domain-specific language for writing program transformations. *As a result, FermaT is capable of transforming its own source code via* meta-transformations. *This paper introduces $\mathcal{META}$WSL and describes some applications of meta-transformations in the FermaT system.*

1 Introduction

The FermaT transformation system, based on research carried out over more than twenty years at Durham University, Software Migrations Ltd. and De Montfort University, is an industrial-strength formal transformation engine with many applications in program comprehension and language migration. FermaT 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.

The first prototype transformation system, called the "Maintainer's Assistant", was written in LISP [4,26,27]. 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 reverse-engineering moderately sized assembler modules into equivalent high-level language programs.

For the next version of the tool (i.e. FermaT itself) we decided to extend WSL to add domain-specific constructs, creating *a language for writing program transformations*. This was called $\mathcal{META}$WSL. The extensions include an abstract data type for representing programs as tree structures and constructs for pattern matching, pattern filling and iterating over components of a program structure. The "transformation engine" of FermaT is implemented entirely in $\mathcal{META}$WSL.

The implementation of $\mathcal{META}$WSL involves a translator from $\mathcal{META}$WSL to Scheme, a small Scheme runtime library (for the main abstract data types) and a WSL runtime library (for the high-level $\mathcal{META}$WSL constructs such as **ifmatch**, **foreach**, **fill** etc.). One aim was so that the tool could be used to maintain its own source code: in fact we wanted to develop meta-transformations which could be applied to the source code of other transformations in order to improve efficiency or implement new constructs. Another aim was to test our theories on language oriented programming [15]: we expected to see a reduction in the total amount of source code required to implement a more efficient, more powerful and more rugged system. We also anticipated noticeable improvements in maintainability and portability. These expectations have been fulfilled, and we are achieving a high degree of functionality from a small total amount of easily maintainable code: the first $\mathcal{META}$WSL implementation of FermaT consisted of around 16,000 lines of $\mathcal{META}$WSL and Scheme code, while the previous version required over 100,000 lines of LISP.

2 Theoretical Foundation

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.

Over the last sixteen years we have been developing the

WSL language, in parallel with the development of a transformation theory and proof methods. In this time the language has been extended from a simple and tractable kernel language [12,16] to a complete and powerful programming language. At the "low-level" end of the language there exists automatic translators from IBM Assembler, Intel x86 Assembler, TPF Assembler, a proprietory 16 bit assembler and PLC code into WSL, and from a subset of WSL into C, COBOL and Jovial. At the "high-level" end it is possible to write abstract specifications, similar to **Z** and VDM.

Expressions and conditions (formulae) in WSL are taken directly from first order logic: in fact, an infinitary first order logic is used (see [6] for details), which allows countably infinite disjunctions and conjunctions, but this is not essential for understanding this paper. This use of first order logic means that statements in WSL can include existential and universal quantification over infinite sets, and similar (non-executable) operations.

The language includes constructs for loops with multiple exits, action systems, side-effects etc. and the transformation theory includes a large catalogue of proven transformations for manipulating these constructs, most of which are implemented in FermaT [15,24,25].

In [17,23] program transformations are used to derive a variety of efficient algorithms from abstract specifications. In [17,22,23] the same transformations are used in the reverse direction: using transformations to derive a concise abstract representation of the specification for several challenging programs.

3 The Kernel Language

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

1. **Assertion:** $\{\mathbf{P}\}$ is an assertion statement which acts as a partial **skip** statement. If the formula **P** is true then the statement terminates immediately without changing any variables, otherwise it aborts (we treat abnormal termination and non-termination as equivalent, so a program which aborts is equivalent to one which never terminates);
2. **Guard:** $[\mathbf{Q}]$ is a guard statement. It always terminates, and enforces **Q** to be true at this point in the program *without changing the values of any variables*. It has the effect of restricting previous nondeterminism to those cases which will cause **Q** to be true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore all the final states will satisfy any desired condition (including **Q**);
3. **Add variables:** $\mathsf{add}(\boldsymbol{x})$ adds the variables in $\boldsymbol{x}$ to the state space (if they are not already present) and assigns arbitrary values to them;
4. **Remove variables:** $\mathsf{remove}(\boldsymbol{y})$ removes the variables in $\boldsymbol{y}$ from the state space (if they are present).

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

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 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 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 $(\mathsf{add}(x);\ [x = 1])$. The **if** statement

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

can be implemented by a nondeterministic choice with guarded arms:

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

For the guarded command [5] such as:

$$\textbf{if } \mathbf{B}_1\ \rightarrow\ \mathbf{S}_1 \,\square\, \mathbf{B}_2\ \rightarrow\ \mathbf{S}_2 \textbf{ fi}$$

we need an assertion to correctly implement the case where both conditions are false:

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

Note the statement will abort if neither $\mathbf{B}_1$ nor $\mathbf{B}_2$ is true.

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.

3.1 Specification Statements

We define the notation $\boldsymbol{x} := \boldsymbol{x}'.\mathbf{Q}$ 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. The formal definition is

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

An important property of this specification statement is that it is guaranteed null-free.

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

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

This says "assign a new value A' to A which is a sorted array and a permutation of the original value of A", it precisely describes *what* we want our sorting program to do without saying *how* it is to be achieved. In other words, it is not biased towards a particular sorting algorithm. In [13] we take this specification as our starting point for the "derivation by formal transformation" of several efficient sorting algorithms.

In [21] we prove 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.

Morgan *et al* [7,8,9,10] use a different specification statement $\boldsymbol{x}\colon [\mathsf{Pre}, \mathsf{Post}]$ 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 $(\{\mathsf{Pre}\};\ (\mathsf{add}(\boldsymbol{x});\ [\mathsf{Post}]))$. 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.

3.2 Weakest Preconditions

The semantics of the kernel language and the definition of refinement and equivalence in terms of these semantics have been discussed in previous papers [18,21,23] so will not be covered here. A fundamental property of the semantics of WSL is that in order to prove a refinement or equivalence relation between two programs, it is sufficient to prove an implication or equivalence between the corresponding weakest preconditions.

If $\mathbf{S}$ is any kernel language statement and V and W are finite non-empty sets of variables, then the condition $\mathbf{S}\colon V \to W$ is true when V and W are syntactically valid initial and final state spaces for $\mathbf{S}$. For example, if V is the set $\{x\}$ and $\mathbf{S}$ is $\mathsf{add}(y)$ then the only valid set W such that $\mathbf{S}\colon V \to W$ is $\{x, y\}$. A kernel language program might have *no* valid initial and final state spaces: consider the statement: $(\mathsf{add}(x) \sqcap \mathsf{remove}(x))$ for example.

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

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

where $(\mu X.\mathbf{S})^0 = \mathbf{abort}$ and
$(\mu X.\mathbf{S})^{n+1} = \mathbf{S}[(\mu X.\mathbf{S})^n / X]$ which is $\mathbf{S}$ with all occurrences of X replaced by $(\mu X.\mathbf{S})^n$. (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 $\mathbf{T}''$).

Here we see the advantage of using infinitary logic: the weakest precondition of a recursive 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 with no recursion).

The fundamental property of the weakest precondition is that the refinement relation (and therefore, program equivalence in general) can be completely encapsulated by two formulae involving weakest preconditions. Let $\mathbf{S}$ and $\mathbf{S}'$ be any statements and let $\boldsymbol{x}$ be a sequence of all the variables assigned to in either $\mathbf{S}$ or $\mathbf{S}'$. Let Δ be any countable set of sentences (formulae with no free variables). Then $\mathbf{S}$ is refined by $\mathbf{S}'$ under the assumptions Δ if and only if the formulae:

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

and

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

are provable from Δ. If this is the case, then we write $\Delta \vdash \mathbf{S} \leq \mathbf{S}'$. See [21] for the proof of this theorem.

Back and von Wright [2] show that the refinement relation can be characterised using weakest preconditions in *higher order logic* (where quantification over formulae 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 $\mathbf{R}$. Can proof rules justified by an appeal to WP in *finitary* logic be justifiably applied to arbitrary programs, for which the appropriate *finite* $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ may not exist? This problem does not occur with infinitary logic, since $\mathrm{WP}(\mathbf{S}, \mathbf{R})$ has a simple definition for all programs $\mathbf{S}$ and all (infinitary logic) formulae $\mathbf{R}$;
2. Second order logic is *incomplete* in the sense that not all true statements are provable. So even if the refinement is true, there is no guarantee that the refinement can be proved.

By using a countable infinitary first order logic we solve both of these problems and the problem of proving the correctness of a refinement of transformation, under a given set of initial conditions, reduces to proving two formulae of first order logic. Because infinitary first order logic is complete (see [6]) we know that if the refinement is valid then proofs of the formulae do exist.

4 $\mathcal{METAL}$WSL: Extensions to WSL

A transformation is a function which maps a WSL program to an equivalent WSL program. WSL programs are represented as abstract syntax trees: therefore we can express a transformation as an operation on a syntax tree. Similarly, we can express the applicability condition of a transformation as a function on syntax trees. (This, obviously, only applies to computable transformations with computable applicability conditions). By extending the WSL language to provide suitable constructs for accessing and manipulating WSL syntax trees we are able to express our transformations in this extension of WSL, called $\mathcal{METAL}$WSL. Since $\mathcal{METAL}$WSL is an extension to WSL, the WSL transformations can also be applied to $\mathcal{METAL}$WSL code (with some small modifications), in addition further $\mathcal{METAL}$WSL specific transformations are possible.

Ty recap: a program transformation can be implemented as a piece of $\mathcal{METAL}$WSL code which in turn can be the source program for applying a transformation (including itself: a transformation can be applied to its own source code). The result will be a *different* implementation of the *same* program transformation.

This "reflexivity" in the system has several advantages:

We can prove the correctness of the *implementation* of a transformation by transforming a specification of the transformation into an implementation of the transformation, using proven transformations.

We can use transformations to improve the efficiency of the transformation system by transforming the source code into a more efficient implementation.

4.1 FermaT Implementation

The first step in implementing the new transformation system was to develop a $\mathcal{METAL}$WSL to Scheme translator (itself written in $\mathcal{METAL}$WSL). The rest of the system was also implemented in $\mathcal{METAL}$WSL and "bootstrapped" into Scheme by using the old LISP implementation of WSL.

The transformation system uses transformations as part of the translation process from WSL to Scheme. Firstly, the WSL to Scheme translator only needs to be implemented on a subset of $\mathcal{METAL}$WSL code: constructs which cannot be directly translated are transformed into equivalent code implementing the constructs in the subset of WSL. In addition, the $\mathcal{METAL}$WSL code implementing a transformation can be transformed into a more efficient but more complex version (see Section 5).

Scheme was chosen as the target of the $\mathcal{METAL}$WSL translator because it is a small, well-defined language (the entire formal language description is less than 50 pages of text) which has an ISO and ANSI standard and which has good facilities for manipulating trees and lists. The FermaT source code consists of about 1,200 lines of Scheme support code and about 46,000 lines of $\mathcal{METAL}$WSL code. The tree-manipulation and pattern matching features of $\mathcal{METAL}$WSL mean that it is possible to implement many of the transformations with only a few lines each of $\mathcal{METAL}$WSL: this greatly improves the productivity, maintainability and reliability of the system [15].

4.2 Abstract Syntax Trees

WSL syntax trees are manipulated via an abstract data type which stores the tree internally and records the "current position" in the syntax tree. A "position" in the tree is represented as a list of integers $\langle p_1, p_2, \ldots, p_n \rangle$ where we take the p_1th node of the root, the p_2th node of that node, and so on. $\mathcal{METAL}$WSL procedures `@Up`, `@Down`,

@Left, @Right, @Goto(P), @To_Last, @To and @Down_To are used to move around the tree. The @Program function returns the whole tree, while @Item returns the current item and @Posn returns the current position. The functions @GT(I), @ST(I), @V(I) and @Cs(I) return the generic type, specific type, value and list of components for the node I. For example, if I is of type T_Variable (an ordinary variable), then I has no components and the value of I is the name of the variable. In this case, @GT(I) = T_Expression and @ST(I) = T_Variable.

The notation $I\hat{}n$ returns the nth component of node I, while $I\hat{}\hat{}L$ returns the subcomponent at the relative position given by L, so, for example, @I = @Program^^@Posn.

The editing procedures are:

@Delete, @Clever_Delete, @Cut,
@Paste_Over, @Paste_Before, @Paste_After,
@Splice_Over, @Splice_Before, @Splice_After

These are applied to the current position in a tree.

@Delete deletes the current item without checking the syntactic correctness of the result. @Clever_Delete also deletes the current item but ensures that the resulting program is syntactically correct. For example, if the whole body of a **while** loop is deleted, then @Clever_Delete will replace the loop by an assertion of the negation of the loop condition: this is semantically equivalent to a **while** loop with no body. @Cut is the same as delete but stores the deleted item in a buffer which can be accessed via the @Buffer function.

The Paste procedures take a single WSL item as parameter, while the Splice procedures take a list of items.

The creation function @Make(t, v, L) returns a new item with specific type s, value v and components L (where L is a list of items). This can be inserted in the tree using the edit operations.

4.3 The foreach Construct

As an example of a high-level construct in $\mathcal{META}$WSL we will consider two variants of the **foreach** construct. A **foreach** is used to iterate over all those components of the currently selected item which satisfy certain conditions, and apply various editing operations to them. Within the body of the **foreach** it appears as if the current item is the whole program. The construct takes care of all the details, when for example, components are deleted, expanded or otherwise edited. Consider the following procedure (which is the implementation of a transformation taken from the FermaT system):

proc @Delete_All_Skips_Code(Data) $\equiv$
  foreach Statement **do**
    if @ST(@I) = `Skip` **then** @Delete **fi od.**

The purpose of this transformation is to delete all occurrences of **skip** statements in the currently selected item. Since **skip** has no effect, the transformation is clearly valid. However, there are various syntactic considerations as shown by the following examples:

Before	After
while B do skip od	$\{\neg$**B**$\}$
if B then skip else $x := 0$ **fi**	**if** $\neg$**B then** $x := 0$ **fi**
do skip od	**abort**
var $x := 0$: **if B then skip fi end**; $y := 0$	$y := 0$

All of these cases are handled by the **foreach** construct: for example, if the body of the **foreach** loop is executed on the body of a **while** loop and deletes the whole body, then the **foreach** will replace the **while** loop by the corresponding assertion.

Another variant of the **foreach** construct iterates over all the *simple terminal statements*. These are the components of a statement which when executed will cause termination of the statement. See [12,14,16] for a detailed definition. In the following program, the two simple terminal statements are boxed:

```
last := " "; line := " "; i := 1;
line := item[i] ++ " " ++ number[i];
if i ⩾ n
  then do do last := item[i];
            i := i + 1;
            if i = n + 1 then write(line); exit(2) fi;
            if item[i] ≠ last
              then write(line); exit(1);
                   if i = j then exit(2) fi
              else line :=line ++ ", "
                             ++ number[i] fi od;
          line := item[i] ++ " " ++ number[i] od
  else skip fi
```

Note that the second occurrence of **exit**(2) is *not* considered a terminal statement because it is not reachable. This is because it occurs as part of a statement which follows an **exit** statement.

An example of a transformation which involves finding all simple terminal statements is *absorb right*. Suppose the previous program is followed by the statement:

if item$[i]$ = error **then exit fi**

This statement can be "absorbed" into all the terminal positions of the preceding statement to give the following equiv-

alent version:

```
last := " "; line := " "; i := 1;
line := item[i] ++ " " ++ number[i];
if i >= n
  then do do last := item[i];
            i := i + 1;
            if i = n + 1
              then write(line);
                   if item[i] = error
                     then exit(3) else exit(2) fi fi;
            if item[i] ≠ last
              then write(line); exit(1);
                   if i = j then exit(2) fi
               else line :=line ++ ", "
                              ++ number[i] fi od;
         line := item[i] ++ " " ++ number[i] od
  else if item[i] = error then exit fi fi
```

Note that the absorbed statement has to be "incremented" (by having its **exit** statements increased in value, and an **else** clause added with an appropriate **exit**) when it is inserted into one or more loops. The unreachable **exit**(2) has not been modified.

If the selected statement is part of an *action system*, then further complexities arise. Some action calls may not return (because the action or a subsequent one called the special action Z which causes immediate termination of the whole action system). In a *regular* action system, no action calls will return and nothing must be absorbed after an action call. In a *hybrid* action system, this is only true for a **call** Z.

All these complexities and special cases are handled automatically by the high-level features of $\mathcal{META}$WSL: the "**foreach** STS" structure finds all the simple terminal statements within a selected statement, the @Increment function deals with moving statements to a lower depth, and the @Gen_Improper? function tests whether the insertion is actually required or not for each case.

The transformation can therefore be implemented in the following few lines of code (which again are taken directly from an early version the FermaT system):

```
proc @Absorb_Right_Code(Data) ≡
  @Right; @Cut; @Left;
  foreach STS do
    if Depth = 0
         ∨ @ST(@I) = Exit ∧ @V(@I) = Depth
      then if @ST(@I) = Exit ∧ Depth > 0
             then @Splice_Over(
                    @Increment(
                      @Buffer, AS_Type, Depth, 0))
           elsif @Gen_Improper?(@I, AS_Type)
             then skip
           elsif @ST(@I) = Skip ∨ @ST(@I) = Exit
             then @Paste_Over(@Buffer)
             else @Paste_After(@Buffer) fi fi od.
```

4.4 The ateach Construct

The construct:

ateach *type* **do**
S od

is similar to a **foreach** but with two main differences:

1. **ateach** processes the tree in top-down fashion (processing the root before any of its components) while **foreach** processes the components first;
2. **ateach** merely moves to the appropriate nodes: it does not create a new temporary program for each processed node, as **foreach** does. As a result, it is possible to move away from the current node within an **ateach** body: though this should be done with care. For example, the loop:
   ateach Statement **do if** @Left? **then** @Left **fi od**
   is unlikely to terminate.

4.5 The ifmatch and fill Constructs

The **ifmatch** construct is used to match the current node against a WSL program schema, while the **fill** construct creates a node by filling in the schema variables in a WSL schema.

Within **ifmatch** constructs *pattern variables* are allowed:

1. ~?x matches any item and puts the matched result into variable x;
2. ~*x matches a sequence of zero or more items and puts the result into x;
3. ~=(e) matches the current item against the value of the expression e.

Within a **fill** construct,:

1. ~?x pastes in the current value of x at this position;
2. ~*x splices the list of items in x over the pattern variable;
3. ~=(e) pastes in the value of expression e at this position.

For example the transformation to reverse the arms of an **if** statement can be written as:

```
ifmatch Statement if ~?B then ~?S1 else ~?S2 fi
  then @Paste_Over(fill Statement
         if ~=(@Not(B))
           then ~?S2
            else ~?S1 fi endfill) endmatch
```

5 Meta-Transformations

The fact that the transformations in FermaT operate on $\mathcal{META}$WSL and are also implemented in $\mathcal{META}$WSL means that we can apply transformations to the source code of other transformation. In this section we discuss some examples of these *meta-transformations* in the FermaT system.

Our first example is the way in which the expression and condition simplifier is implemented in FermaT.

5.1 Expression/Condition Simplifier

The ability to simplify expressions and conditions is important for any program transformation system. An obvious solution is to use an existing theorem prover, and in fact some versions of the Maintainer's Assistant used the Boyer-Moore theorem prover `nqthm` [3]. But that was something of an "overkill" for the purpose and imposed a large overhead in both memory and CPU time. The simplifier is invoked very frequently, mostly on fairly simple expressions such as $x = 0$ or $\neg(x \neq 0)$, and in most cases there is little work that needs to be done. This is because in the vast majority of cases the simplifier is invoked on an expression or condition for which little simplification is possible.

For the industrial strength FermaT transformation system the requirements for an expression and condition simplifier were:

1. Efficient execution: especially on small expressions. This implies a short "start up time";
2. Easily extendible. It would be impossible to attempt to simplify *all* possible expressions which are capable of simplification. For example, we now know that the integer formula $n > 2 \ \wedge \ x^n + y^n = z^n$ can be simplified to **false**: but it took a lot of work to get this particular result! Since we must be content with a less-than-complete implementation, it is important to be able to add new simplification rules as and when necessary;
3. Easy to prove correct. Clearly a faulty simplifier will generate faulty transformations and incorrect code. If the simplifier is to be easily extended, then it is important that we can prove the correctness of the extended simplifier equally easily.

In order to meet requirement (2) the heart of the simplifier is table-driven, consisting of a set of pattern match and replacement operations. For example, the condition $x + y \leqslant z + y$ can be simplified to $x \leqslant z$ whatever expressions are represented by x, y and z. This pattern match and replacement can be coded as a simple **ifmatch** and **fill** in $\mathcal{META}$WSL:

ifmatch Condition $^\sim?x + {}^\sim?y \leqslant {}^\sim?z + {}^\sim=(y)$
  then @Paste_Over(**fill** Condition $^\sim?x \leqslant {}^\sim?z$ **endfill**)
endmatch

To reduce the number of patterns required, the simplifier first normalises the expression as follows:

1. Push down all "negate" operators to the lowest level, using De Morgan's laws, so that $\neg(x = 1 \ \vee \ x = 2)$ becomes $x \neq 1 \ \wedge \ x \neq 2$ for example;
2. Flatten nested associative operators, such as $(x+y)+z$;
3. Evaluate any operators with constant operands;
4. Sort the operands of all commutative operators and merge duplicated operands;
5. Expand factors (where this does not make the expression "too large");

The next step is to check each pattern in the list. To make it easy to add new patterns and to prove that the simplifier is correct, the database of patterns and replacements is implemented as a **foreach** Expression followed by a **foreach** Condition, each of which contains a list of **ifmatch** constructs:

foreach Expression **do**
  ifmatch Expression $(-(-{}^\sim?x))$
    then @Paste_Over(x) **endmatch**;
  ifmatch Expression $1/(1/{}^\sim?x)$
    then @Paste_Over(x) **endmatch**;
  ...
od

If any of the patterns matched, then we repeat from step (2) until the result converges (or we have had "too many" iterations). Finally we factorise expressions where possible and apply some final cosmetic cleanup rules: for example, people usually write $2 * x$ (putting the number first in a multiplication operation), but $x + 2$ (putting the number *last* in an addition).

Clearly, the most expensive step in the process is the pattern matching step and as described above, this step is implemented very inefficiently. For example, if we discover that the operator at the root of the expression is a $+$ with two components, then there is no need to test most of the **ifmatch** patterns. In one version of the Maintainers Assistant the patterns were implemented as a huge nested **if** structure which tested the type of the root and number of components, and then tested the types of each component. This solved the efficiency problem at the expense of being very hard to read, understand and maintain: for example, if a new pattern was added to the wrong place in the structure, then it would simply never get triggered!

The solution we developed for FermaT is to implement a meta-transformation which took as input a sequence of

ifmatch constructs and transformed them into the equivalent nested **if** clause. As a result, new patterns can be added anywhere in the list and the correctness of the list can be proved simply by proving the correctness of each individual pattern. This meta-transformation transforms the source code of the Simplify transformation from a simple but inefficient form to an equivalent convoluted but more efficient implementation.

The resulting improvement in maintainability and productivity is quite dramatic. The source file for the simplifier's patterns (`maths2-1.src`) is only 19,465 bytes and is very easy to maintain because it largely consists of a list of simple **ifmatch** and **fill** constructs. The transformed WSL file (`maths2-2.wsl`) is 85,786 bytes of complex, nested tests. This compiles into a 274,032 byte Scheme file which expands to 884,943 bytes when all the Scheme macros are expanded. The Scheme is then compiled into a 1,089,448 byte C file plus associated 108,098 byte header file. So in this case the final C code is over 60 times larger than the WSL source file!

5.2 WSL to Scheme Translator

Another area where meta-transformations are used extensively is in the $\mathcal{META}$WSL to Scheme translator. Most of the features of WSL are fairly easy to translate into equivalent Scheme code, but some features cause difficulties. In particular, the multi-level **do** ... **od** loops with **exit** statements can only be directly implemented using Scheme's "call with current continuation" procedure [1]. The procedure (call/cc *proc*) packages up the current continuation as an "escape procedure" and passes it as an argument to *proc*. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created. This mechanism can be used to implement exception handling, coroutines and a wide variety of advanced control structures. Unfortunately, this very generality makes it difficult for the Hobbit Scheme compiler [11] to compile Scheme code efficiently to C in the presence of call/cc: even if this particular use of call/cc could be implemented as a simple `goto`. Ordinary **while** loops on the other hand *can* be compiled efficiently.

The solution is to use FermaT's powerful restructuring transformations to transform all the **do** ... **od** loops in the FermaT source code into simple **while** loops.

Transforming a loop **do S od** to a **while** loop is simple if the loop satisfies two conditions:

1. **do S od** is a *proper sequence*, i.e. every terminal statement has terminal value zero. In other words, every **exit**(n) statement must be within n or more nested **do** ... **od** loops. This is easy to arrange by processing the loops from the top down (since the whole program must be a proper sequence);
2. The body of the loop, **S**, must be *reducible*, i.e. replacing any terminal statement **exit**(n) with terminal value one in **S** by **exit**$(n-1)$ should give a terminal statement with terminal value zero.

Condition (2) can be arranged by repeated applications of the Absorb_Right transformation (Section 4.3). For example, consider the following loop:

do if $\mathbf{B}_1$
  then if $\mathbf{B}_2$ **then exit fi**;
    $\mathbf{S}_1$
  else $\mathbf{S}_2$ **fi**;
$\mathbf{S}_3$ **od**

The body of the loop is not reducible since if we replaced the **exit** by **exit**(0) (which is equivalent to **skip**) it would no longer be a terminal statement in the loop body. First we apply Absorb_Right to the statement **if** $\mathbf{B}_2$ ... to give:

do if $\mathbf{B}_1$
  then if $\mathbf{B}_2$ **then exit else** $\mathbf{S}_1$ **fi**
  else $\mathbf{S}_2$ **fi**;
$\mathbf{S}_3$ **od**

Then apply Absorb_Right to the outer **if** statement:

do if $\mathbf{B}_1$
  then if $\mathbf{B}_2$ **then exit else** $\mathbf{S}_1$; $\mathbf{S}_3$ **fi**
  else $\mathbf{S}_2$; $\mathbf{S}_3$ **fi od**

Note that there are now two copies of the statement $\mathbf{S}_3$. In the general case, repeated application of Absorb_Right can lead to an exponential growth in the program size. However, this growth can be avoided by creating procedures from the copied statements and copying the procedure calls. Another transformation Make_Proc carries out this operation:

begin
  do if $\mathbf{B}_1$
    then if $\mathbf{B}_2$ **then exit else** $\mathbf{S}_1$; $F_3()$ **fi**
    else $\mathbf{S}_2$; $F_3()$ **fi od**
where
proc $F_3() \equiv \mathbf{S}_3$**.**
end

Finally, the transformation Floop_To_While converts the loop to a **while** loop by introducing a flag:

begin
  flag $:= 1$;
  while flag $= 1$ **do**
    if $\mathbf{B}_1$
      then if $\mathbf{B}_2$ **then** flag $:= 0$ **else** $\mathbf{S}_1$; $F_3()$ **fi**
      else $\mathbf{S}_2$; $F_3()$ **fi od**
where

proc $F_3() \equiv \mathbf{S}_3$.
end

The resulting program, when translated to Scheme, can be efficiently compiled to C code. Eliminating the flag variable is left as an exercise for the optimising compiler.

The WSL to Scheme translator source code includes many **do** ... **od** loops, as does the source code for Floop_To_While and Reduce_Loop (the latter transformation applies Absorb_Right and Make_Proc to make a loop body reducible). So these transformations are *applied to their own source code* on a regular basis, as part of the FermaT build process.

6 Applications of FermaT

The FermaT transformation system forms the core of the FERMAT Migration and Comprehension Workbench. Other papers [19,20] describe case studies using FermaT: one is a detailed description of using program transformations to extract a formal a specification from an IBM 370 Assembler module. The other case study is a "mass migration" exercise where we took a random selection of 1,925 different Assembler modules, containing just over one million lines of source code, from many different commercial systems, and migrated them all to compilable C code.

Currently, FERMAT is being used in a commercial project to translate over half a million lines of x86 assembler code (the core of a telephone switching system) into high-level, efficient and maintainable C code.

These migration projects involve the automated application of thousands of separate transformations to each source file: the correctness of each transformation and the efficiency of implementation are therefore paramount to the success of the project.

7 Conclusions and Future Work

These is a lot of further scope for using transformations as a way of "compiling" very high level programs down to source code (not just for program transformation systems). Already within FermaT we can freely use high level constructs such as **ifmatch** and **foreach** (and **do** ... **od** loops) without worrying about efficiency: since the transformations can automatically process these constructs into more efficient, but less comprehensible, low-level code. In the future we want to extend this process so that the transformations themselves are written in an abstract "specification" style, possibly with "hints" to guide the transformation process. The source code can then be derived automatically from the specifications via proven transformations. Further efficiency improvements can be gained by eliminating the intermediate Scheme stage: we transform the high-level WSL code into an equivalent very low level WSL code which in turn can be translated directly into low-level C code.

The FermaT transformation system is available under the GNU GPL (General Public License) from the author's web sites:

http://www.dur.ac.uk/∼dcs0mpw/fermat.html
http://www.cse.dmu.ac.uk/∼mward/fermat.html

Acknowledgements

The research described in this paper has been partly funded by EPSRC (The UK Engineering and Physical Sciences Research Council) project "A Proof Theory for Program Refinement and Equivalence: Extensions", partly by Software Migrations Ltd., and partly by De Montfort University.

References

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

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

[3] R. S. Boyer & J. S. Moore, *A Computational Logic Handbook.*, Academic Press, Boston, 1997, Second Edition.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] 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.dur.ac.uk/martin.ward/martin/papers/wcre2000.ps.gz⟩.

[21] M. Ward, "A Reverse Engineering Theorem: Deriving the Specification of Any Given Program," to appear, 2003.

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

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

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

[25] M. Ward & K. H. Bennett, "Formal Methods for Legacy Systems," *J. Software Maintenance: Research and Practice* 7 (May, 1995), 203–219, ⟨http://www.dur.ac.uk/martin.ward/martin/papers/legacy-t.ps.gz⟩.

[26] M. Ward, F. W. Calliss & M. Munro, "The Maintainer's Assistant," *Conference on Software Maintenance 16th–19th October 1989, Miami Florida* (1989), ⟨http://www.dur.ac.uk/martin.ward/martin/papers/MA-89.ps.gz⟩.

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