Reverse Engineering through Formal Transformation: Knuths 'Polynomial Addition' Algorithm

M. P. WARD

Computer Science Department, Science Laboratories, South Road, Durham DH1 3LE, UK

In this paper we will take a detailed look at a larger example of program analysis by transformation. We will be considering Algorithm 2.3.3.A from Knuth's 'Fundamental Algorithms' Knuth (1968) (p. 357) which is an algorithm for the addition of polynomials represented using four-directional links. Knuth (1974) describes this as having "a complicated structure with excessively unrestrained goto statements" and goes on to say "I hope someday to see the algorithm cleaned up without loss of its efficiency". Our aim is to manipulate the program, using semantics-preserving operations, into an equivalent, high-level specification. The transformations are carried out in the WSL language, a 'wide spectrum language' which includes both low-level program operations and high level specifications, and which has been specifically designed to be easy to transform.

Received April 8 1994, revised August 12 1994

1. INTRODUCTION

There has been much research in recent years on the formal development of programs by refining a specification to an executable program via a sequence of intermediate stages, where each stage is proved to be equivalent to the previous one, and hence the final program is a correct implementation of the specification. However, there has been very little work on applying program transformations to reverse-engineering and program understanding. This may be because of the considerable technical difficulties involved: in particular, a refinement method has total control over the structure and organization of the final program, while a reverse-engineering method has to cope with any code that gets thrown at it: including unstructured ('spaghetti') code, poor documentation, misuse of data structures, programming 'tricks' and undiscovered errors. 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 'toy' programs. A different approach to reverse engineering is therefore required: the approach presented in this paper does not require the use of loop invariants to deal with arbitrary loops, (although if invariants are available, they can provide useful information).

There are several distinct advantages to a transformational approach to program development and reverse engineering:

- The final developed program, or derived specification, is correct *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, and reverse engineering, can be supported by the computer. The computer can check the correctness conditions for each step, apply the transformation, store different versions, attach comments and documentation to code, preserve the links between code and specifications, etc.
- Provided the set of transformations is sufficiently powerful, and is capable of dealing with all the low-level constructs in the language, then it becomes possible to use program transformations as a means of restructuring and reverse-engineering existing source code (which has not been developed in accordance with any particular formal method).
- The user does not have to fully understand the code before starting to transform it: the program can be transformed into a more understandable form before it is analysed. This (partial) understanding is then used as a guide in deciding what to do next. Thus transformations provide a powerful program understanding tool.

Our aim in this paper is to demonstrate that our program transformation theory, based on weakest preconditions and infinitary logic, and described in

Ward (1989), Ward (1993a,b) can form the basis for a method for reverse engineering programs with complex data structures and control flow. This transformation theory is used for forward engineering (transforming a high-level abstract specification into an efficient implementation) in Ward (1992b) and Priestley and Ward (1993).

The reverse-engineering method is a heuristic method based on the selection and application of formal transformations, with tool support to check correctness conditions, apply the transformations and store the results. No reverse engineering process can be totally automated, for fundamental theoretical reasons, but as we gain more experience with this approach, we are finding that more and more of the process is capable of being automated.

In Ward (1993a,b) we present a simple example of program analysis by transformation. The paper describes a formal method for reverse engineering existing code which uses program transformations to restructure the code and extract high-level specifications. By a 'specification' we mean a sufficiently precise definition of the input–output behaviour of the program. A 'sufficiently precise' description is one which can be expressed in first order logic and set theory: this includes **Z**, VDM (Jones, 1986) and all other formal specification languages. We did not consider timing constraints in that paper: although the method has been extended to model time as an extra output of a program Younger and Ward (1993).

In this paper we treat a much more challenging example than the one in Ward (1993a,b): a program which exhibits a high degree of both control flow complexity and data representation complexity. The program is Algorithm 2.3.3.A from Knuth (1968) (p. 357) which is an algorithm for the addition of polynomials in several variables. The polynomials are represented in a tree structure using four-directional links. Knuth describes this as having "a complicated structure with excessively unrestrained **goto** statements" Knuth (1974) and goes on to say "I hope someday to see the algorithm cleaned up without loss of its efficiency".

1.1. Transformation methods

The *Refinement Calculus* approach to program derivation Hoare *et al.* (1987), Morgan (1990), Morgan *et al.* (1988) is superficially similar to our program transformation method. It is based on a wide spectrum language, using Morgan's specification statement Morgan (1988) and Dijkstra's guarded commands Dijkstra (1976). However, this language has very limited programming constructs: lacking loops with multiple exits, action systems with a 'terminating' action and side-effects. These extensions are essential if transformations are to be used for reverse engineering. The most serious limitation is that the transformations for introducing and manipulating loops require that any loops introduced must be accompanied by suitable invariant conditions and variant functions. This makes the method unsuitable for a practical reverse-engineering method.

A second approach to transformational development, which is generally favoured in the **Z** community and elsewhere, is to allow the user to select the next refinement step (e.g. introducing a loop) at each stage in the process, rather than selecting a transformation to be applied to the current step. Each step will therefore carry with it a set of *proof obligations*, which are theorems which must be proved for the refinement step to be valid. Systems such as mural (Jones *et al.*, (1991), RAISE (Neilson *et al.*, 1989) and the B-tool (Abrial *et al.*, 1991) take this approach. These systems thus have a much greater emphasis on proofs, rather than the selection and application of transformation rules. Discharging these proof obligations can often involve a lot of tedious work and much effort is being exerted to apply automatic theorem provers to aid with the simpler proofs. However, Sennett (1990) indicates that for 'real' sized programs it is impractical to discharge much more than a tiny fraction of the proof obligations. He presents a case study of the development of a simple algorithm, for which the implementation of one function gave rise to over one hundred theorems which required proofs. Larger programs will require many more proofs. In practice, since few if any of these proofs will be rigorously carried out, what claims to be a formal method for program development turns out to be a formal method for program specification, together with an *informal* development method. For this approach to be used as a reverse-engineering method, it would be necessary to discover suitable loop invariants for each of the loops in the given program, and this is very difficult in general, especially for programs which have not been developed according to some structured programming method.

The well known Munich project CIP (Computer-aided Intuition-guided Programming) (Bauer *et al.*, 1989; Bauer and The CIP Language Group, 1985, 1987) uses a wide-spectrum language based on algebraic specifications and an applicative kernel language. They provide a large library of transformations, and an engine for performing transformations and discharging proof obligations. The kernel is a simple applicative language which uses only function calls and the conditions (if...then) statement. This language is provided with a set of 'axiomatic transformations' consisting of: α-, β-and η-reduction of the Lambda calculus (Church, 1951), the definition of the **if**-statement and some error axioms. Two programs are considered 'equivalent' if one can be reduced to the other by a sequence of axiomatic transformations. The core language is extended until it resembles a functional programming language. Imperative constructs (variables, assignment, procedures, **while**-loops, etc.) are introduced by defining them in terms of this 'applicative core' and giving further axioms which enable the new constructs to be reduced to those already defined. Similar methods are used in Broy *et al.* (1979), Pepper (1979), Wossner *et al.* (1979) and Bauer and Wossner (1982). However, this approach does have some problems with the numbers of axioms required and the difficulty of determining the exact correctness conditions of transformations. These problems are greatly exacerbated when imperative constructs are added to the system.

Problems with purely algebraic specification methods have been noted by Majester (1977). She presents an abstract data type with a simple constructive definition, but which requires several infinite sets of axioms to define algebraically. In addition, it is important for any algebraic specification to be *consistent*, and the usual method of proving consistency is to exhibit a model of the axioms. Since every algebraic specification requires a model, while not every model can be specified algebraically, there seems to be some advantages in rejecting algebraic specifications and working directly with models.

1.2. Our approach

In developing a model based theory of semantic equivalence, we use the popular approach of defining a core 'kernel' language with denotational semantics, and permitting definitional extensions in terms of the basic constructs. In contrast to other work (e.g. Bauer *et al.*, 1989; Bird, 1988; Partsch, 1984) we do not use a purely applicative kernel; instead, the concept of state is included, using a *specification statement* which also allows specifications expressed in first order logic as part of the language, thus providing a genuine wide spectrum language.

Fundamental to our approach is the use of infinitary first order logic (see Karp, 1964) both to express the weakest preconditions of programs Dijkstra (1976) and to define assertions and guards in the kernel language. Engeler (1968) was the first to use infinitary logic to describe properties of programs; Back (1980) used such a logic to express the weakest precondition of a program as a logical formula. His kernel language was limited to simple iterative programs. We use a different kernel language which includes recursion and guards, so that Back's language is a subset of ours. We show that the introduction of infinitary logic as part of the language (rather than just the metalanguage of weakest preconditions), together with a combination of proof methods using both denotational semantics and weakest preconditions, is a powerful theoretical tool which allows us to prove some general transformations and representation theorems (Ward, 1993a,b).

Over the last eight years we have been developing a wide spectrum language (called WSL), in parallel with the development of a transformation theory and proof methods, together with methods for program development and inverse engineering. Recently an interactive program transformation system (called FermaT) has been developed which is designed to automate much of the process of transforming code into specifications and specifications into code. This process can never be completely automated — there are many ways of writing the specification of a program, several of which may be useful for different purposes. So the tool must work interactively with the tedious checking and manipulation carried out automatically, while the maintainer provides high-level 'guidance' to the transformation process. In the course of the development of the prototype, we have been able to capture much of the knowledge and expertise that we have developed through manual experiments, and case studies with earlier versions of the tool, and incorporate this knowledge within the tool itself. For example, restructuring a regular action system (a collection of **gotos** and labels) can now be handled completely automatically through a single transformation.

Any practical program transformation system for reverse engineering has to meet the following requirements:

1. It has to be able to cope with all the usual programming constructs: loops with exits from the middle, **gotos**, recursion, etc.
2. Techniques are needed for dealing with variable aliasing, side-effects and pointers.
3. It cannot be assumed that the code was developed (or maintained) according to a particular programming method: real code ('warts and all') must be acceptable to the system; in particular, significant restructuring may be required before the real reverse engineering can take place. It is important that this restructuring can be carried out automatically or semi-automatically by the transformation system.
4. It should be based on a formal language and formal transformation theory, so that it is possible to *prove* that all the transformations used are semantic-preserving. This allows a high degree of confidence to be placed in the results.
5. The formal language should ideally be a wide spectrum language which can cope with both low-level constructs such as **gotos** and high-level constructs, including non-executable specifications expressed in first order logic and set theory.
6. Translators are required from the source language(s) to the formal language: many large software systems are written in a combination of different languages.
7. It must be possible to apply transformations without needing to understand the program first; this is so that transformations can be used as a program understanding and reverse engineering tool.
8. It must be possible to extract a module, or smaller component, from the system for analysis and transformation, with the transformations guaranteed to preserve all the interactions of that component with the rest of the system. This allows the maintainer to concentrate on 'maintenance hot

spots' in the system, without having to process the entire source code (which may amount to millions of lines).

9. An extensive catalogue of proven transformations is required, with mechanically checkable correctness conditions and some means of composing transformations to develop new ones.
10. An interactive interface which pretty-prints each version on the display will allow the user to instantly see the structure of the program from the indentation structure.
11. The correctness of the transformation system itself must be well-established, since all results depend on the transformations being implemented correctly.
12. A method for reverse engineering by program transformation needs to be developed alongside the transformation system.

1.3. The FermaT project

The WSL language and transformation theory forms the basis of the FermaT project (Bull, 1990; Ward *et al.*, 1989) at Durham University and Durham Systems Engineering Ltd which aims to develop an industrial strength program transformation tool for software maintenance, reverse engineering and migration between programming languages (e.g. Assembler to COBOL). The tool consists of a structure editor, a browser and pretty-printer, a transformation engine and library of proven transformations, and a collection of translators for various source languages.

The initial prototype tool was developed as part of an Alvey project at the University of Durham (Ward *et al.*, 1989). This work on applying program transformation theory to software maintenance formed the basis for a joint research project between the University of Durham, CSM Ltd and IBM UK Ltd., whose aim was to develop a tool to interactively transform assembly code into high-level language code and **Z** specifications. A prototype translator has been completed and tested on sample sections of up to 80 000 lines of assembler code, taken from very large commercial assembler systems. One particular module had been repeatedly modified over a period of many years until the control flow structure had become highly convoluted. Using the prototype translator and ReForm tool we were able to transform this into a hierarchy of (single-entry, single-exit) subroutines resulting in a module which was slightly shorter and considerably easier to read and maintain. The transformed version was hand-translated back into Assembler which (after fixing a single mis-translated instruction) "worked first time". See Ward and Bennett (1993, 1994) for a description of this work and the methods used.

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 METAWSL. 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 METAWSL. The implementation of METAWSL involves a translator from METAWSL to LISP, a small LISP runtime library (for the main abstract data types) and a WSL runtime library (for the high-level METAWSL constructs such as **ifmatch**, **foreach**, **fill**, etc.). One aim was so that the tool could be used to maintain its own source code: and this has already proved possible, with transformations being applied to simplify the source code for other transformations! Another aim was to test our theories on language oriented programming (Ward, 1994): we expected to see a reduction in the total amount of source code required to implement a more efficient, more powerful and more rugged system. We also anticipated noticeable improvements in maintainability and portability. These expectations have been fulfilled and we are achieving a high degree of functionality from a small total amount of easily maintainable code: the current prototype consists of around 16 000 lines of METAWSL and LISP code, while the previous version required over 100 000 lines of LISP.

The tool is designed to be interactive because the reverse-engineering process can never be completely automated — there are many ways of writing the specification of a program, several of which may be useful for different purposes. So the tool must work interactively, with the tedious checking and manipulation carried out automatically, while the maintainer provides high-level 'guidance' to the transformation process. In the course of the development of the prototype, we have been able to capture much of the knowledge and expertise that we have developed through manual experiments and case studies with earlier versions of the tool and incorporate this knowledge within the tool itself. For example, restructuring a regular action system (a collection of **gotos** and labels) can now be handled completely automatically through a single transformation. See Ward (1994) for more details.

FermaT can also be used as a software development system (but this is not the focus of this paper): starting with a high level specification expressed in set-theory and logic notation (similar to **Z** or **VDM**; Jones, 1986), the user can successively transform it into an efficient, executable program. See Priestly and Ward (1993) and Ward (1992b) for examples of program development in WSL using formal transformations. Within FermaT, transformations are themselves coded in an extension of WSL called METAWSL: in fact, much of the code for the prototype is written in WSL and this makes it possible to use the system to maintain its own code:

2. THE LANGUAGE WSL

WSL is the 'wide spectrum language' used in our program transformation work, which includes low-level programming constructs and high-level abstract specifications within a single language. By working within a single formal language we are able to prove that a program correctly implements a specification or that a specification correctly captures the behaviour of a program, by means of formal transformations in the language. We do not have to develop transformations between the 'programming' and 'specification' languages. An added advantage is that different parts of the program can be expressed at different levels of abstraction, if required.

A *program transformation* is an operation which modifies a program into a different form which has the same external behaviour (it is equivalent under a precisely defined denotational semantics). Since both programs and specifications are part of the same language, transformations can be used to demonstrate that a given program is a correct implementation of a given specification. We write $\mathbf{S}_1 \approx \mathbf{S}_2$ if statements $\mathbf{S}_1$ and $\mathbf{S}_2$ are semantically equivalent.

A *refinement* is an operation which modifies a program to make its behaviour more defined and/or more deterministic. Typically, the author of a specification will allow some latitude to the implementor, by restricting the initial states for which the specification is defined, or by defining a non-deterministic behaviour (e.g. the program is specified to calculate a root of an equation, but is allowed to choose which of several roots it returns). In this case, a typical implementation will be a *refinement* of the specification rather than a strict equivalence. The opposite of refinement is *abstraction*: we say that a specification is an abstraction of a program which implements it. See Morgan (1990), Morgan *et al.* (1988) and Back (1980) for a description of refinement. We write $\mathbf{S}_1 \leqslant \mathbf{S}_2$ if $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$, or if $\mathbf{S}_1$ is an abstraction of $\mathbf{S}_2$.

2.1. Syntax and semantics

The syntax and semantics of WSL are described in Priestly and Ward (1993) and Ward (1989, 1993a,b), and so will not be discussed in detail here. Note that we do not distinguish between arrays and sequences, both the 'array notations' and 'sequence notations' can be used on the same objects. For example if a is the sequence $\langle a_1, a_2, \ldots, a_n \rangle$ then:

- $l(a)$ denotes the length of the sequence a;
- $a[i]$ is the ith element a_i;
- $a[i..j]$ denotes the *subsequence* $\langle a_i, a_{i+1}, \ldots, a_j \rangle$;
- last (a) denotes the element $a[l(a)]$;
- butlast (a) denotes the subsequence $a[1..l(a) - 1]$;
- reverse (a) denotes the sequence $\langle a_n, \ldots, a_2, a_1 \rangle$;
- set (a) denotes the set of elements in the sequence, i.e. $\{a_1, a_2, \ldots a_n\}$;
- The statement $x \xleftarrow{\text{pop}} a$ sets x to a_1 and a to $\langle a_2, a_3, \ldots, a_n \rangle$;
- The statement $a \xleftarrow{\text{push}} x$ sets a to $\langle x, a_1, a_2, \ldots, a_n \rangle$;
- The statement $x \xleftarrow{\text{last}} a$ sets x to a_n and a to $\langle a_1, a_2, \ldots a_{n-1} \rangle$.

The concatenation of two sequences is written $a \mathbin{+\!\!+} b$.

Most of the constructs in WSL, for example **<u>if</u>** statements, **<u>while</u>** loops, procedures and functions, are common to many programming languages. However, there, are some features relating to the 'specification level' of the language which are unusual.

Expressions and conditions (formulae) in WSL are taken directly from first order logic: in fact, an infinitary first order logic (see Karp, 1964, for details), which allows countably infinite disjunctions and conjunctions, but this is not essential for this paper. This means that statements in WSL can include existential and universal quantification over infinite sets, and similar (non-executable) operations.

An example of a non-executable operation is the nondeterministic assignment statement (or specification statement) $\langle x_1, \ldots, x_n \rangle := \langle x'_1, \ldots, x'_n \rangle.\mathbf{Q}$ which assigns new values to the variables $x_1, \ldots, x_n$. In the formula $\mathbf{Q}$, x_i represent the old values and x'_i represent the new values. The new values are chosen so that $\mathbf{Q}$ will be true, then they are assigned to the variables. If there are several sets of values which satisfy $\mathbf{Q}$ then one set is chosen nondeterministically. If there are no values which satisfy $\mathbf{Q}$ then the statement does not terminate. For example, the assignment $\langle x \rangle := \langle x' \rangle.(x = 2.x')$ halves x if it is even and aborts if x is odd. If the sequence contains one variable then the sequence brackets may be omitted, for example: $x := x'.(x = 2.x')$. The assignment $x := x'.(y = 0)$ assigns an arbitrary value to x if $y = 0$ initially, and aborts if $y \neq 0$ initially: it does not change the value of y. Another example is the statement $x := x'.(x' \in B)$ which picks an arbitrary element of the set B and assigns it to x (without changing B). The statement aborts if B is empty, while if B is a singleton set, then there is only one possible final value for x.

The *simple assignment* $\langle x_1, \ldots, x_n \rangle := \langle e_1, \ldots, e_n \rangle$ assigns the values of the expressions e_i to the variables x_i. The assignments are carried out simultaneously, so for example $\langle x, y \rangle := \langle y, x \rangle$ swaps the values of x and y. The single assignment $\langle x \rangle := \langle e \rangle$ can be abbreviated to $x := e$.

The *local variable statement* **<u>var</u>** x : **S** **<u>end</u>** introduces a new local variable x whose initial value is arbitrary and which only exists while the statement **S** is executed. If x also exists as a *global* variable, then its value is saved and restored at the end of the block. A collection of local variables can be introduced and initialized using the notation **<u>var</u>** $\langle x_1 := e_1, \ldots, x_n := e_n \rangle$: **S** **<u>end</u>**.

An *action* is a parameterless procedure acting on global variables (*cf.* Arsac, 1982a,b). It is written in the form $A \equiv$ S. where A is a statement variable (the name of the action) and S is a statement (the action body). A set of mutually recursive actions is called an *action system*. There may sometimes be a special action Z, execution of which causes termination of the whole action system even if there are unfinished recursive calls. An occurrence of a statement **call** X within the action body is a call of another action.

An action system is written as follows, with the first action to be executed named at the beginning. In this example, the system starts by calling A_1:

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

For example, this action system is equivalent to the while loop **while B do S od**:

actions A:
$A \equiv$
  if ¬**B then call** Z **fi**;
  S; **call** A. **endactions**

With this action system, each action call must lead to another action call, so the system can only terminate by calling the Z action (which causes immediate termination). Such action systems are called *regular*.

For a given set X, the *non-deterministic iteration* over X is written **for** $i \in X$ **do S od**. This executes the body **S** once for each element in X, with i taking on the value of each element. It is equivalent to the following:

var $\langle i := 0, X' := X \rangle$:
while $X' \neq \emptyset$ **do**
  $i := i'.(i' \in X'); X' := X' \setminus \{i\}$;
  S od end

For a *sequence* X, the iteration over the elements of X is written **for** $x \overset{\text{pop}}{\longleftarrow} X$ **do S od**. The elements are taken in their order in the sequence, so the loop is deterministic. The loop is equivalent to:

var $\langle i := 0, X' := X \rangle$:
while $X' \neq \emptyset$ **do**
  $i \overset{\text{pop}}{\longleftarrow} X'$;
  S od end

3. EXAMPLE TRANSFORMATIONS

In this section we give some examples of the transformations to be used later in the paper.

3.1. Loop inversion

The first example is a simple restructuring transformation. Suppose statement $\mathbf{S}_1$ is a *proper sequence*, i.e. it cannot cause termination of an enclosing loop. then is $\mathbf{S}_1$ appears at the beginning of a loop body, we can take it out of the loop provided we insert a second copy of $\mathbf{S}_1$ at the *end* of the loop. In other words, the statement **do** $\mathbf{S}_1$; $\mathbf{S}_2$ **od** is equivalent to $\mathbf{S}_1$; **do** $\mathbf{S}_2$; $\mathbf{S}_1$ **od**.

This transformation is useful in both directions, for example we may convert a loop with an exit in the middle to a **while** loop:

do $\mathbf{S}_1$; **if B then exit fi**; $\mathbf{S}_2$ **od**
$\approx$
$\mathbf{S}_1$; **while** ¬**B do** $\mathbf{S}_2$; $\mathbf{S}_1$ **od**

when $\mathbf{S}_1$ and $\mathbf{S}_2$ are both proper sequences. Or we may use it in the reverse direction to reduce the size of a program by merging two copies of $\mathbf{S}_1$.

3.2. Loop unrolling

The simplest loop unrolling transformation is the following:

while B do S od
$\approx$
if B then S; **while B do S od fi**

This simply unrolls the first step of the loop. The next transformation unrolls a step of the loop *within* the loop body. For *any* condition **Q**:

while B do S od
$\approx$
while B do S; **if B** $\wedge$ **Q then S fi od**

This can be useful when the body **S** is able to be simplified when condition **Q** is true. An extension of this transformation is to unroll an arbitrary number of iterations into the loop body:

while B do S od
$\approx$
while B do S; **while B** $\wedge$ **Q do S od od**

As an example of the effect of several unrolling operations, consider the following program schema:

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

where executing $\mathbf{S}_1$ makes $\mathbf{B}_2$ true and $\mathbf{B}_1$ false (i.e. $\{\mathbf{B}_1\}$; $\mathbf{S}_1 \leqslant \{\mathbf{B}_1\}$; $\mathbf{S}_1$; $\{\mathbf{B}_2 \wedge \neg \mathbf{B}_1\}$), and $\mathbf{S}_2$ is the only statement which can affect condition $\mathbf{B}_1$. If we selectively unroll after $\mathbf{S}_2$, then **B** will still be true, $\mathbf{B}_1$ will be false, and $\mathbf{B}_2$ will be true. So we can prune the inserted **if** statement to get:

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

Since $\mathbf{S}_1$ does not affect **B**, we can selectively unroll the entire loop after $\mathbf{S}_1$ under the condition $\mathbf{B} \wedge \mathbf{B}_1$ (which reduces to $\mathbf{B}_1$ since **B** is true initially and not affected by $\mathbf{S}_1$):

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

Convert the **elsf** to **else** **if**, take out $\mathbf{S}_3$, and roll up one step of the inner **while** loop to get:

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

3.3. General recursion removal

Our next transformation is a general transformation from a recursive procedure into an equivalent iterative procedure, using a stack. It can also be applied in reverse, to turn an iterative program into an equivalent recursive procedure (which may well be easier to understand). The theorem was presented in Ward (1992a), and the proof may be found in Ward (1991).

Suppose we have a recursive procedure whose body is a regular action system in the following form:

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

where the statements $\mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j}$ preserve the value of x and no **S** contains a call to F (i.e. all the calls to F are listed explicitly in the B_j actions) and the statements $\mathbf{S}_{j0}, \mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j-1}$ contain no action calls. There are $M+N$ actions in total: $A_1, \ldots, A_M$, $B_1, \ldots, B_N$. Note that since the action system is regular, it can only be terminated by executing **call** Z, which will terminate the current invocation of the procedure.

The aim is to remove the recursion by introducing a local stack K which records 'postponed' operations: When a recursive call is required we 'postpone' it by pushing the pair $\langle 0, e\rangle$ onto K (where e is the parameter required for the recursive call). Execution of the statements $\mathbf{S}_{jk}$ also has to be postponed (since they occur between recursive calls), we record the postponement of $\mathbf{S}_{jk}$ by pushing $\langle\langle j,k\rangle, x\rangle$ onto K. Where the procedure body would normally terminate (by calling Z) we instead call a new action $\hat{F}$ which pops the top item off K and carries out the postponed operation. If we call $\hat{F}$ with the stack empty then all postponed operations have been completed and the procedure terminates by calling Z.

THEOREM 3.1. The procedure $F(x)$ above is equivalent to the following iterative procedure which uses a new local stack K and a new local variable m:

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

By unfolding the calls to $\hat{F}$ in B_j we can avoid pushing and popping $\langle 0, g_{j1}(x)\rangle$ onto K and instead, call A_1 directly. So we have the corollary:

COROLLARY 3.2. $F(x)$ is equivalent to:

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

Note that *any* procedure $F(x)$ can be restructured into the form of Theorem 3.1; in fact there may be several different ways of structuring $F(x)$ which meet these criteria. The simplest such restructuring is to put each recursive call into its own B action (with no other statements apart from a call to the next action). Since it is always applicable, this is the method used by most compilers. See Ward (1992a) for further applications of the theorem.

3.4. Tail recursion

A simple case of tail recursion is the following:

```
proc F(x) ≡ if B1 then S1; F(y)
                  else S2 fi.
```

where S_1 and S_2 may both call $F()$. The terminal call can be implemented with a **while** loop as follows:

```
proc F(x) ≡ while B1 do S1; x := y od; S2.
```

A slightly more complicated example:

```
proc F(x) ≡ if B1 then if B2 then S1; F(y)
                             else S2 fi
                  else S3 fi.
```

is equivalent to:

```
proc F(x) ≡ while B1 ∧ B2 do S1; x := y od;
            if B1 then S2 else S3 fi.
```

4. POLYNOMIAL ADDITION

A polynomial P in several variables may be expressed as:

$$P = \sum_{0 \leqslant j \leqslant n} g_j x^{e_j} \qquad (1)$$

where x is a variable (the primary variable), $n > 0$, $0 = e_0 < e_1 < \cdots < e_n$ are non-negative integers and for each $0 \leqslant j \leqslant n$, g_j (the coefficient of the jth term) is either a number or a polynomial whose primary variable is alphabetically less than x. Each polynomial has a constant term (which may have coefficient zero) and one or more other terms (which must have non-zero coefficients).

This definition lends itself to a tree structure, Knuth uses nodes with four links each to implement the tree structure, we will represent these nodes using the following six arrays:
For each integer i:

- $D[i]$ is either Λ (for a constant polynomial) or points down the tree to the constant term of a circularly-linked list of terms.
- $C[i]$ If $D[i] = \Lambda$ then $C[i]$ is a number (the value of the coefficient), otherwise it is a symbol (the variable of the polynomial).
- $E[i]$ is the value of the exponent for this term.
- $L[i]$ points to the previous term in the circular list.
- $R[i]$ points to the next term in the circular list.
- $U[i]$ points up the tree, from each term of a polynomial to the polynomial itself.

The 'next term' is either the term with the next largest exponent, or the term with zero exponent. The algorithm assumes that there is a 'sufficiently large' number of free nodes available on the stack avail.

The root node P of a polynomial stores the following values:

- $C[P]$ is either the constant value (for a constant polynominal) or the primary variable.
- $E[P]$ is zero.
- $L[P]$ points to P.
- $R[P]$ points to P.
- $U[P]$ is Λ: an otherwise unused pointer value.
- $D[P]$ is either Λ (for a constant polynomial) or points to the constant term of a circular list of terms.

If $D[P] \neq \Lambda$ then $D[P]$ is the first term of a list, $E[D[P]] = 0$, the term $L[D[P]]$ has the largest exponent (which must be greater than zero), the last term P' in the list (with the lowest exponent) can be recognised by the fact that $E[L[P']] = 0$.

4.1. Knuth's algorithm

Knuth (1968) includes an algorithm for adding polynomials represented as tree structures with four-way linked nodes. The algorithm is written in an informal notation, using labels and **gotos**. We have translated the algorithm into WSL, using an action system with one action for each label.

```
ADD ≡
  if D[P] = Λ
    then while D[Q] ≠ Λ do Q := D[Q] od;
      call A3
    else if D[Q] = Λ ∨ C[Q] < C[P] then call A2
    elsf C[Q] = C[P]
      then P := D[P]; Q := D[Q]; call ADD
      else Q := D[Q]; call ADD fi fi.
A2 ≡
       pop
  r ←── avail; s := D[Q];
  if s ≠ Λ then do U[s] := r; s := R[s];
                   if E[s] = 0 then exit fi od fi;
  U[r] := Q; D[r] := D[Q]; L[r] := r;
  R[r] := r; C[r] := C[Q]; E[r] := 0;
  C[Q] := C[P]; D[Q] := r; call ADD.
A3 ≡
  {E[Q] ≠ 0 ⇒ (E[P] = E[Q] ∧ C[U[P]] = C[U[Q]])};
  C[Q] := C[Q] + C[P];
  if C[Q] = 0 ∧ E[Q] ≠ 0 then call A8 fi;
  if E[Q] = 0 then call A7 fi;
  call A4.
A4 ≡
  P := L[P];
  if E[P] = 0
    then call A6
    else do Q := L[Q];
            if E[Q] ⩽ E[P] then exit fi od;
      if E[Q] = E[P] then call ADD fi fi;
    call A5.
```

$A_5 \equiv$
  $r \xleftarrow{\text{pop}}$ avail;
  $U[r] := U[Q];\ D[r] := \Lambda;\ L[r] := Q;$
  $R[r] := R[Q];\ L[R[r]] := r;\ R[Q] := r;$
  $E[r] := E[P];\ C[r] : 0;\ Q := r;$
  call ADD.
$A_6 \equiv$
  $P := U[P];$ **call** A_7.
$A_7 \equiv$
  if $U[P] = \Lambda$
    then **call** A_{11}
    else **while** $C[U][Q]] \neq C[U[P]]$ **do**
        $Q := U[Q]$ **od**;
      call A_4 **fi**.
$A_8 \equiv$
  $\{E[P] = E[Q] \wedge C[U[P]] = C[U[Q]]\};$
  $r := Q;\ Q := R[r];\ s := L[r];\ R[s] := Q;$
  $L[Q] := s;$ avail $\xleftarrow{\text{push}} r$:
  if $E[L[P]] = 0 \wedge Q = s$ **then** **call** A_9
        else **call** A_4 **fi**.
$A_9 \equiv$
  $r := Q;\ Q := U[Q]; D[Q] := D[r];$
  $C[Q] := C[r];$ avail $\xleftarrow{\text{push}} r;$
  $s := D[Q];$
  if $s \neq \Lambda$
    then **do** $U[s] := Q;\ s := R[s];$
        if $E[s] = 0$ **then** **exit** **fi** **od** **fi**;
  call A_{10}.
$A_{10} \equiv$
  if $D[Q] = \Lambda\ \wedge C[Q] = 0 \wedge E[Q] \neq 0$
    then $P := U[P];$ **call** A_8
    else **call** A_6 **fi**.
$A_{11} \equiv$
  while $U[Q] \neq \Lambda$ **do** $Q := U[Q]$ **od**;
  call Z.

See Figure 1 for the call graph of this program.

The two assertions have been taken from the comments Knuth makes about the algorithm. We will prove that they are valid later on, because this will be much easier with the recursive version of the program.

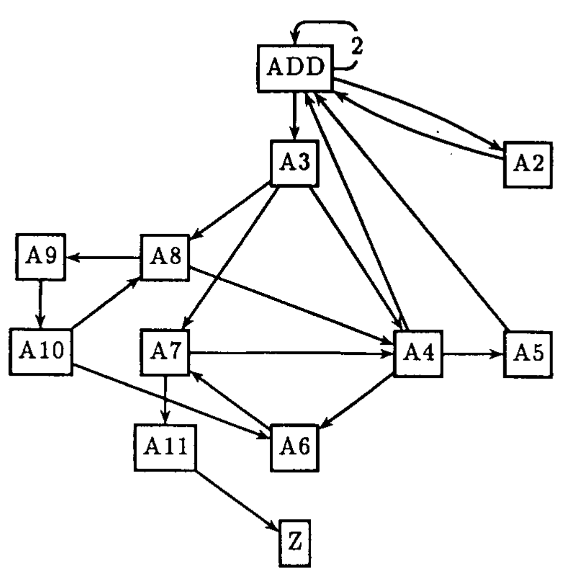

FIGURE 8. The call graph of Knuth's polynomial addition algorithm.

5. ANALYSIS BY TRANSFORMATION

We will now show how such an algorithm can be analysed by applying a sequence of transformation steps which first transform it into a structured form and then derive a mathematical specification of the algorithm. Since each of the transformation steps has been proven to preserve the semantics of a program, the correctness of the specification so derived is guaranteed.

The program exhibits both control flow complexity and data representation complexity, with the control flow directed by the data structures. With the aid of program transformations it is possible to 'factor out' these two complexities, dealing first with the control flow and then changing the data representation. Both control and data restructuring can be carried out using only local information, it is not until near the end of the analysis (when much of the complexity has been eliminated, and the program is greatly reduced in size) that we need to determine the 'big picture' of how the various components fit together. This feature of the transformational approach is essential in scaling up to large programs, where it is only possible in practice to examine a small part of the program at a time.

5.1. Restructuring

The first step in analysing the program involves simple restructuring. We begin by looking for procedures and variables which can be 'localized'. In this case there are a number of blocks of code which can be extracted out as procedures, some of which use local variables. The names for the procedures are taken from the comments in the original program: This reduces the size of the main body of tangled 'spaghetti code' in preparation for the restructuring,

begin
actions ADD:
ADD $\equiv$
  if $D[P] = \Lambda$
    then **while** $D[Q] \neq \wedge$ **do** $Q := D[Q]$ **od**; **call** A_3
    else **if** $(D[Q] = \Lambda) \vee (C[Q] < C[P])$ **then** **call** A_2
      elsf $C[Q] = C[P]$
        then $P := D[P], Q := D[Q];$ **call** ADD
        else $Q := D[Q];$ **call** ADD **fi** **fi**.
$A_2 \equiv$
  Insert_Below_Q; **call** ADD.
$A_3 \equiv$
  $\{E[Q] \neq 0 \Rightarrow (E[P] = E[Q] \wedge C[U[P]] = C[U[Q]])\};$
  $C[Q] := C[Q] + C[P];$
  if $(C[Q] = 0) \wedge (E[Q] \neq 0)$ **then** **call** A_8 **fi**;
  if $E[Q] = 0$ **then** **call** A_7 **fi**;
  call A_4.

```
A4 ≡
  P := L[P];
  if E[P] = 0
    then call A6
    else Move_Left_Q:
      if E[P] = E[Q] then call ADD fi fi;
  call A5.
A5 ≡
  Insert_to_Right; call ADD.
A6 ≡
  P := U[P]; call A7.
A7 ≡
  if U[P] = Λ then call A11
              else Move_Up_Q; call A4 fi.
A8 ≡
  {E[P] = E[Q] ∧ C[U[P]] = C[U][Q]]};
  Delete_Zero_Term;
  if (E[L[P]] = 0) ∧ (Q = L[Q])
    then call A9
    else call A4 fi.
A9 ≡
  Delete_Const_Poly; call A10.
A10 ≡
  if ((D[Q] = Λ) ∧ (C[Q] = 0)) ∧ (E[Q] ≠ 0)
    then P := U[P]; call A8
    else call A6 fi.
A11 ≡
  while U[Q] ≠ Λ do Q := U[Q] od;
  call Z. endactions
where
proc Insert_Below_Q ≡
  r ←pop avail; s := D[Q];
  if s ≠ Λ
    then do U[s] := r; r := R[s];
            if E[s] = 0 then exit fi od fi;
  ⟨U[r] := Q, D[r] := D[Q], L[r] := r, R[r]; = r⟩;
  ⟨C[r] := C[Q], E[r] := 0⟩;
  ⟨C[Q] := C[P], D[Q] := r⟩.

proc Move_Left_Q ≡
  do Q := L[Q]; if E[Q] ⩽ E[P] then exit fi od.

proc Insert_to_Right ≡
  r ←pop avail;
  ⟨U[r] := U[Q], D[r] := ∧, L[r] := Q, R[r] := R[Q]⟩;
  L[R[r]] := r; R[Q] := r;
  ⟨E[r] := E[P], C[r] := 0⟩;
  Q := r.

proc Move_Up_Q ≡
  while C[U[Q]] ≠ C[U[P]] do Q := U[Q] od.

proc Delete_Zero_Term ≡
  r := Q;
  ⟨Q := R[r], s := L[r]⟩;
  R[s] := Q; L[Q] := s;
  avail ←push r.

proc Delet_Const_Poly ≡
  r := Q; Q := U[Q];
  ⟨D[Q] := D[r], C[Q] := C[r]⟩;
  avail ←push r; s := D[Q];
  if s ≠ Λ
    then do U[s] := Q; s := R[s];
            if E[s] = 0 then exit fi od fi.
```

The next stage is to restructure the 'spaghetti' of labels and jumps by unfolding action calls, introducing loops, re-arranging **if** statements, merging action calls, and so on. In the Maintainer's Assistant this whole process has been automated in a single transformation `Collapse_Action_System` which follows heuristics we have developed over a long period of time: selecting the sequence of transformations required to restructure a program. The result of this single transformation is as follows:

```
do do if D[P] = Λ
        then while D[Q] ≠ Λ do Q := D[Q] od;
         {E[Q] ≠ 0
           ⇒ (E[P] = E[Q] ∧ C[U[P]] = C[UQ]])};
         C[Q] := C[Q] + C[P];
         if (U[P] = Λ  ∧ E[Q] = 0)
             ∧(C[Q] ≠ 0 ∨ E[Q] = 0)
           then while U[Q] ≠ Λ do Q := U[Q] od;
             exit (2)
         elsf (C[Q] ≠ 0) ∨ (E[Q] = 0)
           then if E[Q] = 0 then Move_Up_Q fi;
             exit fi;
         do {E[P] = E[Q] ∧ C[U][P]] = C[U[Q]]};
           Delete_Zero_Term;
           fi (E(L[P]] ≠ 0) ∨ (Q ≠ L[Q])
               then exit fi;
             Delete_Const_Poly;
             P := U[P];
             if (U[P] = Λ)
                 ∧(C[Q] ≠ 0 ∨ D[Q] ≠ Λ ∨ E[Q] = 0)
               then while U[Q] ≠ Λ do
                          Q := U[Q] od;
                 exit (2)
             elsf (C[Q] ≠ 0) ∨ (D[Q] ≠ Λ)
                     ∨(E[Q] = 0)
               then Move_Up_Q; exit fi od
       else (D[Q] = Λ) ∨ (C[Q] < C[P])
         then Insert_Below_Q
       elsf C[Q] = C[P] then P := D[P], Q := D[Q]
                         else Q := D[Q] fi od;
   do P := L[P];
      if E[P] ≠ 0
        then Move_Left_Q;
          if E[P] ≠ E[Q] then Insert_To_Right fi;
          exit fi;
      P := U[P];
      if U[P] = Λ
        then while U[Q] ≠ ∧ do Q := U[Q] od;
          exit (2) fi;
      Move_Up_Q od od
```

As can be seen above, most of the restructuring has been carried out by this single transformation. There is some potential for further simplification transformations, taking code out of loops and **if** statements and so on:

```
do while D[P] ≠ Λ do
    if D[Q] = Λ ∨ C[Q] < C[P] then Insert_Below_Q
    elsif C[Q] = C[P] then P := D[P], Q := D[Q]
                     else Q := D[Q] fi od;
  while D[Q] ≠ Λ do Q := D[Q] od;
  {E[Q] ≠ 0 ⇒ (E[P] = E[Q] ∧ C[U[P]] = C[U[Q]])};
  C[Q] := C[Q] + C[P];
  if C[Q] = 0 ∧ E[Q] ≠ 0
    then do {E[P] = E[Q] ∧ C[U[P]] = C[U[Q]]};
            Delete_Zero_Term;
            if E[L[P]] ≠ 0 ∨ Q ≠ s then exit fi;
            Delete_Const_Poly;
            P := U[P];
            if U[P] = Λ
                ∧(C[Q] ≠ 0 ∨ E[Q] = 0 ∨ D[Q] ≠ ∧)
              then exit (2) fi;
            if D[Q] ≠ Λ ∨ C[Q] ≠ 0 ∨ E[Q] = 0
              then Move_Up_Q; exit fi od
    else if U[P] = Λ then if E[Q] = 0 then exit fi fi;
      if E[Q] = 0 ∧ then Move_Up_Q fi fi;
  do P := L[P];
     if E[P] ≠ 0 then exit fi;
     P := U[P];
     if U[P] = Λ then exit (2) fi;
     Move_Up_Q od;
  Move_Left_Q;
  if E[P] ≠ E[Q] then Insert_to_Right fi od;
while U[Q] ≠ Λ do Q := U[Q] od
```

Turning our attention to the loop **while** $D[P] \neq \Lambda$ **do** ... **od** we see that only one of the arms of the inner **if** statement can affect the value of $D[P]$: for the other two cases, the loop test is redundant. Secondly, the procedure Insert_Below_Q is guaranteed to make $D[Q] \neq \Lambda$ and $C[Q] = C[P]$. The loop can be made more efficient by entire loop unrolling for the case $D[Q] \neq \Lambda \vee C[Q] > C[P]$ followed by loop body unrolling after Insert_Below_Q. The result is:

```
while D[P] ≠ Λ do
  while D[Q] ≠ Λ ∧ C[Q] > C[P] do Q := D[Q] od;
  if D[Q] = Λ ∨ C[Q] < C[P] then Insert_Below_Q fi;
  P := D[P]; Q := D[Q] od
```

On termination of this loop we clearly have $D[P] = \Lambda$. A little later, we test if $U[P] = \Lambda$. The only possibility for both $D[P] = \Lambda$ and $U[P] = \Lambda$ is if the original P polynomial was a constant. It is rather inefficient to repeatedly test for this trivial case, so instead we assume that constant polynomials are treated as a special case, outside the main loop. This allows us to remove the test $U[P] = \Lambda$ from the body of the main loop.

Next we consider the final **do** ... **od** loop:

```
do P := L[P];
  if E[P] ≠ 0 then exit fi;
  P := U[P];
  if U[P] = Λ then exit (2) fi;
  Move_Up_Q od;
```

By pushing the statement $P := L[P]$ into the following **if** statement and then taking it out of the loop, we get the pair of assignments $P := L[P]$; $P := U[P]$ which can be simplified to $P := U[P]$ (since each node in each circular list has the same U value). So the loop simplifies to:

```
do if E[L[P]] ≠ 0 then exit fi;
  P := U[P];
  if U[P] = Λ then exit (2) fi;
  Move_Up_Q od;
P := L[P];
```

Finally, the test $U[P] = \Lambda \wedge (D[Q] \neq \Lambda \vee C[Q] \neq 0 \vee E[Q] = 0)$ is more complicated than it needs to be. If, as in this case, we have just deleted a constant polynomial in Q which has resulted in a zero term higher up the structure of Q, then $D[Q] = \Lambda \wedge C[Q] = 0 \wedge E[Q] \neq 0$. But in this case, Q is somewhere in the middle of a list of terms of a polynomial in a certain variable, and therefore P must also be somewhere in the list of terms of a polynomial in the *same* variable (the addition of two of the terms having resulted in a zero term). So we cannot also have $U[P] = \Lambda$. Conversely, if it happens that $U[P] = \Lambda$, then the test for a zero term must fail and there is no need to also test $(D[Q] \neq \Lambda \vee C[Q] \neq 0 \vee E[Q] = 0)$.

Putting these results together we get the simplified main body:

```
do while D[P] ≠ Λ do
    while D[Q] ≠ Λ ∧ C[Q] > C[P] do
      Q := D[Q] od;
    if D[Q] = Λ ∨ C[Q] < C[P]
      then Insert_Below_Q fi;
    P := D[P], Q := D[Q] od;
  while D[Q] ≠ Λ do Q := D[Q] od;
  {E[Q] ≠ 0 ⇒ (E[P] = E[Q] ∧ C[U[P]] = C[U[Q]])};
  C[Q] := C[Q] + C[P];
  if C[Q] = 0 ∧ E[Q] ≠ 0
    then
      do {E[P] = E[Q] ∧ C[U[P]] = C[U[Q]]};
        Delete_Zero_Term;
        if E[L[P]] ≠ 0 ∨ Q ≠ L[Q] then exit fi;
        Delete_Const_Poly;
        P := U[P];
        if U[P] = Λ
          then {D[Q] ≠ Λ ∨ C[Q] ≠ 0 ∨ E[Q] = 0};
            exit (2) fi;
        if D[Q] ≠ Λ ∨ C[Q] ≠ 0 ∨ E[Q] = 0
            then Move_Up_Q; exit fi od
      else if E[Q] = 0 then Move_Up_Q fi fi;
    do if E[L[P]] ≠ 0 then exit fi;
        P := U[P];
        if U[P] = Λ then exit (2) fi;
        Move_Up_Q od;
    P := L[P];
```

```
    Move_Left_Q;
    if E[P] ≠ E[Q] then Insert_to_Right fi od;
while U[Q] ≠ Λ do Q := U[Q] od
```

5.2. Introduce recursion

The next step is to introduce recursion. We have discovered that for a great many program analysis problems, it is very important to get to a recursive form of the program as early as possible in the analysis process. Discovering the overall structure and operation of a program, such as this one, is enormously easier once a recursive form has been arrived at.

Before we can introduce recursion, we need to restructure the program into a suitable action system. This will make explicit the places where recursive calls will ultimately appear, and where the test(s) for termination occur(s). Note that P starts out with $U[P] = \Lambda$ and the program terminates as soon as $U[P] = \Lambda$ again: which suggests that P will ultimately be a parameter. Also, note that the tree structure reachable through the initial value of P is not changed by the program, and P is restored to its original value. There are two places where the assignment $P := U[P]$ occurs, and where termination is possible. These are separated out into the two actions $\hat{A}_1$ and $\hat{A}_2$ below.

```
actions A:
A ≡
  if D[P] = Λ
    then while D[Q] ≠ Λ do Q := D[Q] od;
      {E[Q] ≠ 0
        ⇒ (E[P] = E[Q] ∧ C[U[P]] = C[I[Q]])};
      C[Q] := C[Q] + C[P];
      if E[Q] = 0 then Move_Up_Q fi;
      call Â₁
    else while D[Q] ≠ Λ ∧ C[Q] > C[P] do
          Q := D[Q] od;
      if D[Q] = Λ ∨ C[Q] < C[P]
        then Insert_Below_Q fi;
      P := D[P]; Q := D[Q]; call A fi.
Â₁ ≡
  if D[Q] ≠ Λ ∨ C[Q] ≠ 0 ∨ E[Q] = 0
    then call Â₂ fi;
  {E[P] = E[Q] ∧ C[U[P]] = C[U[Q]]};
  Delete_Zero_Term;
  if E[L[P]] = 0 ∧ Q = L[Q]
    then Delete_Const_Poly;
      P := U[P];
      if U[P] = Λ
        then while U[Q] ≠ Λ do Q := U[Q] od;
          call Z fi;
      call Â₁
    else call Â₂ fi.
Â₂ ≡
  if E[L[P]] = 0
    then P := U[P];
      if U[P] = Λ
        then while U[Q] ≠ Λ do Q := U[Q] od;
          call Z fi;
      Move_Up_Q;
      call Â₂
    else call B fi.
B ≡
  P := L[P]; Move_Left_Q;
  if E[P] ≠ E[Q] then Insert_to_Right fi;
  call A.
endactions
```

Within the two 'finishing' actions, $\hat{A}_1$ and $\hat{A}_2$, the pointer P is moved up and $U[P]$ tested against Λ. For the recursion introduction theorem, we must have only one occurrence of **call** Z, and in this case we would prefer to have only one occurrence of $P := U[P]$. This is because kind of structure we would like for the recursive procedure is something like this:

```
proc ADD ≡
  if D[P] = Λ
    then deal with a constant polynomial
    else P := D[P]; Q := D[Q];
    do ADD; Add a pair of terms;
      deal with a zero result;
      P := L[P];
      if E[P] = 0 then exit fi;
      set up a term in Q od;
    Deal with a constant polynomial result;
    P := U[P];
    Move up Q if needed fi.
```

Fortunately, any two similar (or even dissimilar) actions can be merged by creating a composite action and using a flag to determine which action the composite action is simulating. In the next version $\hat{A}$ is equivalent to $\hat{A}_1$ when flag is **true**, and equivalent to $\hat{A}_2$ when flag is **false**:

```
actions A:
A ≡
  if D[P] = Λ
   then while D[Q] ≠ Λ do Q := D[Q] od;
    {E[Q] ≠ 0 ⇒ (E[P] = E[Q] ∧ C[U[P]] = C[U[Q]])};
    C[Q] := C[Q] + C[P];
    if E[Q] = 0 then Move_Up_Q fi;
    flag := true; call Â
  else while D[Q] ≠ Λ ∧ C[Q] > C[P] do
        Q := D[Q] od;
   if D[Q] = Λ ∨ C[Q] < C[P] then Insert_Below_Q fi;
   P := D[P]; Q := D[Q]; call A fi.
Â ≡
  if flag ∧ D[Q] = Λ ∧ C[Q] = 0 ∧ E[Q] ≠ 0
    then {E[P] = E[Q] ∧ C[U[P]] = C[U[Q]]};
      Delete_Zero_Term
    else flag := false fi;
  if E[L][P]] = 0
    then if flag ∧ Q = L[Q]
          then Delete_Const_Poly
          else flag := false fi;
     P := U[P];
     if U[P] = Λ
```

```
        then while U[Q] ≠ Λ do Q := U[Q] od;
          call Z fi;
      if ¬flag then Move_Up_Q fi;
      call Â
    else flag := false; call B fi.
B ≡
  P := L[P];
  Move_Left_Q;
  if E[P] ≠ E[Q] then Insert_to_Right fi;
  call A.
endactions
```

Now we can apply Theorem 3.1 in reverse to get an equivalent recursive procedure:

```
begin
if D[P] = Λ then while D[Q] ≠ Λ do Q := D[Q] od;
                 C[Q] := C[Q] + C[P];
                 while U[Q] ≠ Λ do Q := U[Q] od
            else ADD fi
where
proc ADD ≡
  if D[P] = Λ
    then while D[Q] ≠ Λ do Q := D[Q] od;
      {E[Q] ≠ 0 ⇒ (E[P] = E[Q] ∧ C[U[P]] = C[U[Q]])};
      C[Q] := C[Q] + C[P];
      if E[Q] = 0 then Move_Up_Q fi;
      flag := true
    else while D[Q] ≠ Λ ∧ C[Q] > C[P] do
           Q := D[Q] od;
      if D[Q] = Λ ∨ C[Q] < C[P] then Insert_Below_Q fi;
      P := D[P]; Q := D[Q];
      do ADD;
        fi flag ∧ D[Q] = Λ ∧ C[Q] = 0 ∧ E[Q] ≠ 0
          then {E[P] = E[Q] ∧ C[U[P]] = C[U[Q]]};
            Delete_Zero_Term
          else flag := false fi;
        if E[L[P]] = 0 then exit fi;
        flag := false;
        P := L[P];
        Move_Left_Q;
        if E[P] ≠ E[Q] then Insert_to_Right fi od;
      if flag ∧ Q = L[Q]
        then Delete_Const_Poly
        else flag := false fi;
      P := U[P];
      if U[P] = Λ
        then while U[Q] ≠ Λ do Q := U[Q] od
      elsif ¬flag then Move_Up_Q fi fi.
end
```

With a recursive program, we can see that ADD preserves P, since the sequence of operations applied to P is: $P := D[P]$ followed by $P := L[P]$ zero or more times, and finally $P := U[P]$, which restores P its original value. It is also easier with the recursive version to prove that the flag can be removed. First we prove that:

$$\neg \text{flag} \Rightarrow \neg(D[Q] = \Lambda \wedge C[Q] = 0 \wedge E[Q] \neq 0)$$

on termination of ADD; and

$$\neg \text{flag} \Rightarrow Q \neq L[Q]$$

on termination of the **do** ... **od** loop.

When the loop terminates, the only way a zero polynomial could have been created (with $Q = L[Q]$) is if we just deleted the only non-zero term. If we have just deleted a term then flag is true, otherwise flag is false and there is no need to test for a constant polynomial. Similarly, the only way a zero term could be created is if we have just deleted a constant polynomial, in which case flag is true. If flag is false on returning from ADD, there is no need to test for a zero term.

On termination of the loop, if the flag is false, then there must still be a non-zero exponent term in the Q list of terms. (Recall that initially, every list of terms in P and Q contains a constant (zero exponent) term plus at least one non-zero exponent term). In this case, $Q \neq L[Q]$.

On termination of an inner procedure call, if the flag is false, then we have either just added two constant elements and possibly moved up Q (in which case $E[Q] = 0$), or we have just added a list of terms, and moved up Q. In either case $E[Q] = 0$.

One final optimization (missed by Knuth) uses the fact that $C[U[Q]] = C[U[P]]$ on termination of the loop. If we do *not* delete a constant polynomial, then after the assignment $P := U[P]$, the **while** loop in Move_Up_Q must be executed at least once. So we can save a test by unrolling one execution of the loop in this case.

It should be noted that the arguments stated above are much easier to state prove in terms of the recursive version of the program, rather than the original iterative version. In addition, these facts are not required in order to transform the iterative version to the recursive version. We need only a very limited and localised analysis of the program in order to reach a recursive equivalent, from which a more extensive analysis becomes feasible.

We are now in a position to eliminate flag from the procedure:

```
begin
if D[P] = Λ then while D[Q] ≠ Λ do Q := D[Q] od;
                 C[Q] := C[Q] + C[P]
            else ADD fi;
where
proc ADD ≡
  if D[P] = Λ
    then while D[Q] ≠ Λ do Q := D[Q] od;
      {E[Q] ≠ 0 ⇒ (E[P] = E[Q] ∧ C[U[P]] = C[U[Q]])};
      C[Q] := C[Q] + C[P];
      if E[Q] = 0 then Move_Up_Q fi;
      {E[P] = E[Q] ∧ C[U[P]] = C[U[Q]]}
    else while D[Q] ≠ Λ ∧ C[Q] > C[P] do
           Q := D[Q] od;
      if D[Q] = Λ ∨ C[Q] < C[P] then Insert_Below_Q fi;
      P := D[P]; Q := D[Q];
```

```
    do ADD;
      if D[Q] = Λ ∧ C[Q] = 0 ∧ E[Q] ≠ 0
        then {E[P] = E[Q] ∧ C[U][P]] = C[U[Q]]};
          Delete_Zero_Term fi;
    P := L[P];
    if E[P] = 0 then exit fi;
    Move_Left_Q;
    if E[P] ≠ E[Q] then Insert_to_Right fi od
  P := U[P];
  if Q = L[Q] then Delete_Const_Poly
            else Q := U[Q] fi;
  if U[P] = Λ
    then while U[Q] ≠ Λ do Q := U[Q] od
    else Move_Up_Q fi.
end
```

5.3. Efficiency of the restructured algorithm

This version of the program (or its iterative equivalent) fulfills Knuth's desire for a cleaned up version, without loss of efficiency. The cleaned up version does carry out a small number of extra tests, which Knuth's version was able to avoid with the use of tortuous control flow. However, it also avoids a number of the redundant tests present in Knuth's version: for example the repeated test for a constant polynomial P and the immediate testing of the new node introduced by Insert_Below_Q. We have carried out number of empirical tests on both algorithms, with polynomials of various sizes and shapes. For these tests we measure 'efficiency' by counting the total number of array accesses; since for modern RISC processors, main memory access is likely to the dominant factor in execution speed.

For the pathological cases where virtually all the terms in Q are cancelled out by terms in P, our version of the algorithm can run up to 10% slower than Knuth's. However, for more usual cases, including a large number of tests carried out with random polynomials of various shapes and sizes, our version of the algorithm is consistently faster than Knuth's, and averages around 5% faster.

5.4. Add parameters to the procedure

With this recursive version it is easy to show that ADD preserves the values of P and Q. For the P the proof is simple since the only assignments to P are $P := D[P]$, followed by one or more $P := L[P]$, followed by one $P := U[P]$, which restores P (since for every node $U[L[P]] = U[P]$). For Q there are two cases to consider:

1. $U[P] = \Lambda$ initially. This is true for the outermost call only. In this case $U[Q] = \Lambda$ is also true initially. The assignments to Q are one or more $Q := D[Q]$ followed by zero or more $Q := L[Q]$ and then repeatedly assigning $Q := L[Q]$ until $U[Q] = \Lambda$ again. The only node in the Q tree with a U value of Λ is the original root, and all the assignments to Q keep it within a valid tree.
2. $U[P] \neq \Lambda$ initially. This is true for the recursive calls. Within the body of the procedure, ADD is only called with $E[Q] = E[P]$ and $C[U[Q]] = C[U[P]]$. The assignments to Q are one or more $Q := D[Q]$ followed by zero or more $Q := L[Q]$ followed by one or more $Q := U[P]$ until $C[U[Q]] = C[U[P]]$ again (where P has now been restored to its original value). This will restore Qs original value since each 'level' in the P and Q trees have different C values; so Q must be returned to the same 'level' and the 'down ... left ... up' sequence means that Q must be at the same position in that level.

Since P and Q are both preserved by ADD, they can be turned into parameters, and the code for 'restoring' P and Q can be deleted. We get:

```
begin
if D[P] = Λ then while D[Q] ≠ Λ do Q := D[Q] od;
               C[Q] := C[Q] + C[P]
               while U[Q] ≠ Λ do Q := U[Q] od
               else ADD (P, Q) fi;
where
proc ADD (P, Q) ≡
  if D[P] = Λ
   then while D[Q] ≠ Λ do Q := D[Q] od;
    {E[Q] ≠ 0 ⇒ (E[P] = E[Q] ∧ C[U[P]] = C[U[Q]])};
    C[Q] := C[Q] + C[P];
   else while D[Q] ≠ Λ ∧ C[Q] > C[P] do
         Q := D[Q] od;
    if D[Q] = Λ ∨ C[Q] < C[P] then Insert_Below_Q fi;
    P := D[P]; Q := D[Q];
    do ADD (P, Q);
      if D[Q] = Λ ∧ C[Q] = 0 ∧ E[Q] ≠ 0
        then {E[P] = E[Q] ∧ C[U[P]] = C[U[Q]]};
          Delete_Zero_Term fi;
      P := L[P];
      if E[P] = 0 then exit fi;
      Move_Left_Q;
      if E[P] ≠ E[Q] then Insert_to_Right fi od;
    if Q = L[Q] then Delete_Const_Poly fi.
end
```

With the parametrised version, it is no longer necessary to treat a constant polynomial in P as a special case. If P is a constant polynomial, then ADD (P, Q) is equivalent to:

```
var Q₀ := Q:
while D[Q] ≠ ∧ do Q := D[Q] od;
C[Q] := C[Q] + C[P];
Q := Q₀ end
```

which gives the correct result.

6. INTRODUCE ABSTRACT DATA TYPES

The abstract data type 'polynomial' is defined informally by the equation:

$$p = \begin{cases} \langle v \rangle & \text{if } p \text{ is a constant polynomial} \\ \langle x, t \rangle & \text{otherwise} \end{cases}$$

where v is the value of the constant polynomial, x is the symbol of the non-constant polynomial and t is the list of terms for the non-constant polynomial.

Each term in the list t is of the form $\langle e, c\rangle$ where e is the exponent of this term and c is the coefficient (which is another polynomial whose variables, if any, are smaller than x). The first term always has a zero exponent, and the coefficient of the first term only may be a zero polynomial (i.e. $\langle 0\rangle$). There is at least one other term, and all other terms have non-zero exponents and coefficients, and are in order of increasing exponents. So t is of the form:

$$t = \langle\langle 0, c_0\rangle, \langle e_1, c_1\rangle, \ldots, \langle e_k, c_k\rangle\rangle$$

where $k \geqslant 1$ and $0 < e_1 < \cdots < e_k$ and $c_i \neq \langle 0\rangle$ for $1 \leqslant i \leqslant n$.

More formally, we define the sets of abstract polynomials as follows:

Definition 6.1 *Abstract Polynomials.* Suppose we have an ordered set VARS of variable names, and a set VALS of values. Define:

$$\mathsf{POLYS} =_{\mathrm{DF}} \bigcup_{n<\omega} \mathsf{POLYS}^n$$

where

$$\mathsf{POLYS}^0 =_{\mathrm{DF}} \{\langle v\rangle | v \in \mathsf{VALS}\}$$

is the set of constant polynomials, and for each $n \geqslant 0$

$$\mathsf{POLYS}^{n+1} =_{\mathrm{DF}} \mathsf{POLYS}^n \cup \{\langle x, t\rangle | x \in \mathsf{VARS} \wedge t \in \mathsf{TERMS}^n \wedge \forall i, 1 \leqslant i \leqslant l(t). \forall y \in \mathsf{vars}(t[i][2]. y < x\}$$

The set TERMS^n is the set of term lists which use elements of POLYS^n as coefficients:

$$\mathsf{TERMS}^n =_{\mathrm{DF}} \{\langle\langle 0, c_0\rangle, \langle e_1, c_1\rangle, \ldots, \langle e_k, c_k\rangle\rangle | k > 0 \wedge \forall i, 1 \leqslant i \leqslant k. c_k \in \mathsf{POLYS}^n \wedge 0 < e_1 < \cdots < e_k\}$$

The function $\mathsf{vars}(p)$ returns the set of variables used in polynomial p:

$$\mathsf{vars}(p) =_{\mathrm{DF}} \begin{cases} \emptyset & \text{if } p = \langle v\rangle \\ \{x\} \cup \bigcup_{0 \leqslant i \leqslant k} \mathsf{vars}(c_i) & \text{otherwise} \end{cases}$$

Now we can define the *abstraction function* $\mathsf{poly}(P)$ which returns the abstract polynomial represented by the pointer P and the current values of arrays E, C, L, R, U and D:

Definition 6.2 *The polynomial abstraction function*:

$$\mathsf{poly}(P) =_{\mathrm{DF}} \begin{cases} \langle C[P]\rangle & \text{if } D[P] = \Lambda \\ \langle C[P], \mathsf{terms}(D[P])\rangle & \text{if } D[P] \neq \Lambda \end{cases}$$

where

$$\mathsf{terms}(P) =_{\mathrm{DF}} \mathsf{term} * (\langle P\rangle +\!\!\!+ \mathsf{list}(P, L, P))$$

The notation $\mathsf{term} * L$ denotes the list formed by applying the function term to each element of list L. The term function is defined:

$$\mathsf{term}(P) =_{\mathrm{DF}} \langle E[P], \mathsf{poly}(C[P])\rangle$$

For abstract polynomials we define the following functions:

$$\mathsf{const?}(p) =_{\mathrm{DF}} \begin{cases} \textbf{true} & \text{if } p \text{ is constant, i.e. } l(p) = 1 \\ \textbf{false} & \text{otherwise} \end{cases}$$

$v(p) =_{\mathrm{DF}}$ variable of $p = p[1]$

$c(p) =_{\mathrm{DF}}$ value of the constant poly $= p[1]$

$T(p) =_{\mathrm{DF}}$ list of terms for $p = p[2]$

$e_i(p) =_{\mathrm{DF}}$ exponent of the ith term $= p[2][i][1]$

$c_i(p) =_{\mathrm{DF}}$ coefficient of the ith term $= p[2][i][2]$

7. ADDING ABSTRACT VARIABLES

The first step towards creating an equivalent abstract program is to 'build the scaffolding' by adding abstract variables p, q and r to the program as *ghost variables*. These are variables which are assigned to within the program, but (at the moment) their values are never referenced, so they can have no effect on the behaviour of the program. We assume the following invariant is true at the beginning of ADD and add assignments to ensure that it is true before the recursive call:

$$p = \mathsf{poly}(P) \wedge q = \mathsf{poly}(Q)$$

We will also add assignments to r so that on termination $r = \mathsf{poly}(Q)$.

It is convenient to replace the two inner **while** loops by the equivalent tail recursions:

```
proc ADD (P, Q) ≡
  var p0 := p, q0 := q:
  if D[P] = Λ
    then if D[Q] ≠ Λ
           then Q := D[Q]; q := c0(q0); ADD (P, Q);
                r := ⟨v(q0), ⟨⟨e0(q0), r⟩⟩ ++ T(q0)[2..]⟩
           else C[Q] := C[Q] + C[P];
                r := ⟨c(q0) + c(p0)⟩ fi
  elsif D[Q] ≠ Λ ∧ C[Q] > C[P]
    then Q := D[Q]; q := c0(q0); ADD(P, Q);
      r := ⟨v(q0), ⟨⟨e0(q0), r⟩⟩ ++ T(q0)[2..]⟩
    else if D[Q] = Λ ∨ C[Q] < C[P]
           then Insert_Below_Q; q := ⟨v(p0), ⟨⟨0, q0⟩⟩⟩fi;
      P := D[P]; D[Q];
      var i := 1, j := 1, t := T(q0):
      do p := ci(p0); q := t[j];
        ADD (P, Q);
        if D[Q] = Λ ∧ C[Q] = 0 ∧ E[Q] ≠ 0
          then Delete_Zero_Term;
            t := t[1..j − 1] ++ t[j + 1..]
          else t[j][2] := r fi;
        P := L[P]; i := i − 1;
```

```
    if i = 0 then i := l(T(p_0)) fi;
    if E[P] = 0 then exit fi;
    do Q := L[Q]; j := j − 1;
      if j = 0 then j := l(t) fi;
      if E[Q] ≤ E[P] then exit fi od;
    if E[P] ≠ E[Q]
      then Insert_to_Right;
        t := t[1..j − 1] ++ ⟨⟨e_i(p_0), ⟨0⟩⟩⟩
             ++ t[j..] fi od;
    if Q=L[Q] then Delete_Const_Poly; r := ⟨t[1][2]⟩
                else r := ⟨v(p_0), t⟩ fi fi end.
```

With this version, the abstract variables p, q, r, etc., are pure ghost variables which have no effect on the operation of the program. But now that we have both abstract and concrete variables available, we can work through the program replacing references to concrete variables by the equivalent references to abstract variables. For example the test $D[P] = \wedge$ is equivalent to the test const?(p) given that $p = \text{poly}(P)$. the effect is to 'demolish the building' leaving the abstract 'scaffolding' to hold everything up. This 'ghost variables' technique has been used for program development in Broy and Pepper (1982), Jorring and Scherlis (1987) and Wile (1981). Assuming that what we are really interested in is the r result for a given p and q, we can delete the concrete variables from the procedure to leave an equivalent abstract procedure (equivalent as far as its effect on r anyway). The procedure add(p,q) is equivalent to

```
ADD (P, Q); r := poly(Q).
proc add (p, q) ≡
  var p_0 := p, q_0 := q:
  if const? (p)
    then if ¬const(q)
          then q := c_0(q_0); add(p, q);
            r := ⟨v(q_0, ⟨⟨e_0(q_0), r⟩⟩ ++ T(q_0)[2..]⟩
          else r := ⟨c(q_0) + c(p_0)⟩ fi
  elsif ¬const?(q) ∧ v(q) > v(p)
    then q := c_0(q_0); add(p, q);
      r := ⟨v(q_0), ⟨⟨e_0(q_0), r⟩⟩ ++ T(q_0)[2..]⟩
    else if const?(q) ∨ v(q) < v(p)
          then q := ⟨v(p_0), ⟨⟨0, q_0⟩⟩⟩ fi;
      var i := 1, j := 1, t := T(q_0):
      do p := c_1(p_0); q := t[j];
        add (p, q);
        if const? (r) ∧ c(r) = 0 ∧ j > 1
          then t := t[1..j − 1] ++ t[j + 1..]
          else t[j][2] := r fi;
        i := i − 1; if i = 0 then i := l(T(p_0)) fi;
        if e_i(p_0) = 0 then exit fi;
        do j := j − 1; if j = 0 then j := l(t) fi;
          if t[j][1] ≤ e_i(p_0) then exit fi od;
        if e_i(p_0) ≠ t[j][1]
          then t := t[1..j − 1] ++ ⟨⟨e_i(p_0), ⟨0⟩⟩⟩
                 ++ t[j..] fi od end;
    if l(t) = 1 then r := ⟨t[1][2]⟩
              else r := ⟨v(p_0), t⟩ fi fi end.
```

The first iteration of the **do** ... **od** loop is a special case, since: (1) The loop is guaranteed to execute at least twice, because every non-constant polynomial has at least two terms; and (2) For the first iteration we know that $e_1(p_0) = e_1(q_0) = 0$ and $i = j = 1$, so both indexes will 'cycle round' on the first iteration, and will not do so on subsequent iterations. So we unroll the first iteration and convert the loop to a **while** loop:

```
var i := 1, j := 1, t := T(q_0):
add (c_1(p_0), c_1(q_0));
t[1][1] := r;
i := l(T(p_0)); j := l(t);
while i > 1 do
  while t[j][1] > e_i(p_0) do j := j − 1 od;
  if e_i(p_0) ≠ t[j][1]
    then t := t[1..j − 1] ++ ⟨⟨e_i(p_0), ⟨0⟩⟩⟩ ++ t[j..] fi;
  add (c_i(p_0), t[j]);
  if const?(r) ∧ c(r) = 0
    then t := t[1..j − 1] ++ t[j + 1..]
    else t[j][1] := r fi;
  i := i − 1 od end
```

The **while** loop is adding two lists of terms. We can make this behaviour more explicit (and get rid of the i and j variables) by putting $T(p_0)$ into t_p, $T(q_0)$ into t_q and deleting elements from the ends of t_p and t_q once they have been dealt with. The new value of t is built up in a new variable t_r, so that t is represented by $t_q \mathbin{++} t_r$. Since the loop adds the elements in reverse order, it makes sense to move the add($c_1(p_0), c_1(q_0)$) call to the end, especially since at this point $t_p = \langle c_1(p_0)\rangle$ and $t_q := \langle c_1(q_0), \ldots\rangle$:

```
var t_p := T(p_0), t_q := T(q_0), t_r := ⟨ ⟩:
i := l(T(p_0)); j := l(s);
while l(t_p) > 1 do
  while last(t_q)[1] > last(t_p)[1] do;
      t_r ←push− last(t_q); t_q := butlast(t_q) od;
  if last(t_p)[1] ≠ last(t_q)[1]
    then t_r ←push− ⟨last(t_p)[1], ⟨0⟩⟩
    else t_r ←push− last(t_q); t_q := butlast(t_q) fi;
  add (last(t_p)[2], t_r[1][2]);
  if const? (r) ∧ c(r) = 0 then t_r := t_r[2..]
                            else t_r[1][2] := r fi;
  t_p := butlast(t_p) od
add(t_p[1][2], t_q[1][2]);
t_r := ⟨0, r⟩ ++ t_q[2..] ++ t_r:
```

The next step is to make this **while** loop into a tail-recursive procedure which takes t_p and t_q as arguments, and returns the result in t_r. We can apply the tail-recursion transformation of section 3.4 to remove the inner **while** loop:

```
proc add(p, q) ≡
  if const?(p)
    then if ¬const(q)
          then q := c_0(q); add(p, q);
            r := ⟨v(q), ⟨⟨e_0(q), r⟩⟩ ++ T(q)[2..]⟩
```

```
            else r := ⟨c(q) + c(p)⟩ fi
    elsif ¬const?(q) ∧ v(q) > v(p)
      then q := c0(q); add(p, q);
        r := ⟨v(q), ⟨⟨e0(q), r⟩⟩ ++ T(q)[2..]⟩
      else if const?(q) ∨ v(q) < v(p)
             then q := ⟨v(p), ⟨⟨0, q⟩⟩⟩ fi;
        var t_r := ⟨⟩ : add_list(T(p), T(q)) end;
        if l(t_r) = 1 then r := ⟨t_r[1][2]⟩
                      else r := ⟨v(p), t_r⟩ fi fi.
proc add_list(t_p, t_q) ≡
  if l(t_p) = 1
    then add(t_p[1][2], t_q[1][2]);
      t_r := ⟨0, r⟩ ++ t_q[2..] ++ t_r
  elsif last(t_q)[1] > last(t_p)[1]
    then t_r ←push last(t_q); add_list(t_p, butlast(t_q))
    else if last(t_p)[1] ≠ last(t_p)[1]
           then t_r ←push ⟨last(t_p)[1], ⟨0⟩⟩
           else t_r ←push last(t_q); t_q := butlast(t_q) fi;
      add(last(t_p)[2], t_r[1][2]);
      if const?(r) ∧ c(r) = 0 then t_r := t_r[2..]
                              else t_r[1][2] := r fi;
      t_p := butlast(t_p);
      add_list(t_p, t_q) fi.
```

Finally, we can convert the procedures into the equivalent functions:

```
funct add(p, q) ≡
  if const?(p)
    then if ¬const(q)
           then ⟨v(q), ⟨⟨e0(q), add(p, c0(q))⟩⟩
                      ++ T(q)[2..]⟩
           else ⟨c(q) + c(p)⟩ fi
    else if ¬const?(q) ∧ v(q) > v(p)
           then ⟨v(q), ⟨⟨e0(q), add(p, c0(q))⟩⟩
                     ++ T(q)[2..]⟩
           else ⌈if const?(q) ∨ v(q) < v(p)
                  then q := ⟨v(p), ⟨⟨0, q⟩⟩⟩ fi;
                var t_r := add_list(T(p), T(q));
                if l(t_r) = 1 then ⟨t_r[1][2]⟩
                              else ⟨v(p), t_r⟩ fi⌋ fi fi.
funct add_list(t_p, t_q) ≡
  if l(t_p) = 1
    then ⟨0, add(t_p[1][2], t_q[1][2])⟩ ++ t_q[2..]
    else if last(t_q)[1] > last(t_p)[1]
           then add_list(t_p, butlast(t_q)) ++ ⟨last(t_q)⟩
           else ⌈var r := ⟨⟩:
                  if last(t_p)[1] ≠ last(t_q)[1]
                    then r := ⟨0⟩
                    else r := last(t_q)[2];
                      t_q := butlast(t_q) fi;
                  r := add(last(t_p)[2], r);
                  if const?(r) ∧ c(r) = 0
                    then add_list(butlast(t_p), t_q)
                    else add_list(butlast(t_p), t_q)
                      ++ ⟨⟨last(t_p)[1], r⟩⟩ fi⌋ fi fi.
```

A final optimization to add_list is to absorb the statement $r := \mathsf{add}(\mathsf{last}(t_p)[2], r)$ into the preceding **if** statement and avoid adding a zero polynomial.

```
funct add_list(t_p, t_q) ≡
  if l(t_p) = 1
    then ⟨0, add(t_p[1][2], t_q[1][2])⟩ ++ t_q[2..]
    else if last(t_q)[1] > last(t_p)[1]
           then add_list(t_p, butlast(t_q)) ++ ⟨last(t_q)⟩
           else ⌈var r := ⟨⟩:
                  if last(t_p)[1] ≠ last(t_q)[1]
                    then r := last(t_p)[2]
                    else r := add(last(t_p)[2], last(t_q)[2]);
                      t_q := butlast(t_q) fi;
                  if const?(r) ∧ c(r) = 0
                    then add_list(butlast(t_p), t_q)
                    else add_list(butlast(t_p), t_q)
                      ++ ⟨⟨last(t_p)[1], r⟩⟩ fi⌋ fi fi.
```

From this version of the program it is a trivial matter to derive the following specification:

$$\mathsf{add}(p,q) =_{\mathrm{DF}} \begin{cases} \langle c(q)+c(p)\rangle \\ \quad \textbf{if } \mathsf{const}(p) \wedge \mathsf{const}(q) \\ \langle v(q), \langle\langle e_0, \mathsf{add}(p, c_0(q))\rangle\rangle \mathbin{+\!\!+} T(q)[2..]\rangle \\ \quad \textbf{if } \mathsf{const}(p) \wedge \neg\mathsf{const}(q) \\ \quad \text{or } \neg\mathsf{const}(p) \wedge \neg\mathsf{const}(q) \wedge v(q) > v(p) \\ \mathsf{add}(p, \langle v(p), \langle\langle 0, q\rangle\rangle\rangle) \\ \quad \textbf{if } \neg\mathsf{const}(p) \wedge \neg\mathsf{const}(q) \wedge v(q) < v(p) \\ A(p,q) \\ \quad \text{otherwise} \end{cases}$$

where

$$A(p,q) =_{\mathrm{DF}} \begin{cases} \langle v(p), \mathsf{add\_list}(T(p), T(q))\rangle \\ \quad \textbf{if } l(\mathsf{add\_list}(\mathrm{T}(p), \mathrm{T}(q))) > 1 \\ \langle \mathsf{add\_list}(T(p), T(q))[1][2]\rangle \\ \quad \text{otherwise} \end{cases}$$

and

$$\mathsf{add\_list}(t_p, t_q) =_{\mathrm{DF}} \begin{cases} \langle 0, \mathsf{add}(t_p[1][2], t_q[1][2])\rangle \mathbin{+\!\!+} t_q[2..] \\ \quad \textbf{if } l(t_p) = 1 \\ \mathsf{add\_list}(t_p, \mathsf{butlast}(t_q)) \mathbin{+\!\!+} \langle \mathsf{last}(t_q)\rangle \\ \quad \textbf{if } l(t_p) > 1 \wedge \mathsf{last}(t_q)[1] > \mathsf{last}(t_p)[1] \\ AL(t_p, t_q) \\ \quad \text{otherwise} \end{cases}$$

where

$$AL(t_p, t_q) =_{\mathrm{DF}} \begin{cases} AL'(t_p, t_q, \mathsf{last}(t_p)[2]) \\ \quad \textbf{if } \mathsf{last}(t_p)[1] \neq \mathsf{last}(t_q)[1] \\ AL'(t_p, \mathsf{butlast}(t_q), \mathsf{add}(\mathsf{last}(t_p)[2], \mathsf{last}(t_q)[2])) \\ \quad \text{otherwise} \end{cases}$$

and

$$AL'(t_p, t_q, r) =_{\mathrm{DF}} \begin{cases} \mathsf{add\_list}(\mathsf{butlast}(t_p), t_q) & \\ \quad \textbf{if } \mathsf{const?}(r) \wedge c(r) = 0 & \\ \mathsf{add\_list}(\mathsf{butlast}(t_p), t_q) \mathbin{+\!\!\!+} \langle\langle \mathsf{last}(t_p)[1], r\rangle\rangle & \\ \quad \text{otherwise} & \end{cases}$$

8. CONCLUSION

Reverse engineering in particular, and program analysis in general, are becoming increasingly important as the amounts spent on maintaining and enhancing existing software systems continue to rise year by year. We claim that reverse engineering based on the application of proven semantic-preserving transformations in a formal wide spectrum language is a practical solution to the problem. In Ward (1993a,b) we outlined a method for using formal transformations in reverse engineering. In this paper the method has been further developed and applied to a much more challenging example program. Although our sample program is only a couple of pages long, it exhibits a high degree of control flow complexity (as can be seen in Figure 1) together with a complicated data structure which is updated as the algorithm progresses. Our approach does not require the user to develop and prove loop invariants nor does it require the user to determine an abstract version of the program and then verify equivalence. Instead, the first stages involve the application of general purpose transformations for restructuring, simplification, and introducing recursion. Because these are general-purpose transformations, they require no advance knowledge of the programs behaviour before they can be applied. This is essential in a reverse engineering application, since the whole purpose of the exercise is to determine the behaviour of the program! Once a recursive version of the program has been arrived at, it becomes possible to deduce various properties of the program, which allow further simplifications to take place. The data structure complexity is dealt with in several stages: first an abstract data type is developed and abstract variables are added to the program alongside the 'real' (concrete) variables. At this stage, the abstract variables are 'ghost' variables whose values have no effect on the program's operation. It is now possible to determine the relationships between abstract and concrete variables (these relationships can be proved using local information rather than requiring global invariants). One by one, the references to concrete variables are replaced by equivalent references to abstract variables. Once all references to concrete variables have been removed, they become 'ghost' variables and can be eliminated from the program. The results is an abstract program which is guaranteed to be equivalent to the original concrete program. This abstract program can then be further simplified, again using general-purpose transformations, until a high-level abstract specification is arrived at. For our case study, the reverse engineering process takes the following stages:

1. Restructure.
2. Introduce recursion using a flag.
3. Remove the flag in the recursive version.
4. Add parameters.
5. Add abstract variables.
6. Remove the concrete variables.
7. Restructure.
8. Introduce more recursion.
9. Rewrite as a recursive specification.

ACKNOWLEDGEMENTS

The research described in this paper has been supported by SERC project 'A Proof Theory for Program Refinement and Equivalence: Extensions'.

REFERENCES

Abrial, J. R., Davis, S. T., Lee, M. K. O., Neilson, D. S., Scharbach, P. N. and Sørensen, I. H. (1991) *The B Method.* BP Research, Sunbury Research Centre.

Arsac, J. (1982a) Transformation of recursive procedures. In Neel, D. (ed.), *Tools and Notations for Program Construction*, pp. 211–265. Cambridge University Press, Cambridge.

Arsac, J. (1982b) Syntactic source to source program transformations and program manipulation. *Commun. ACM*, **22,** 43–54.

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

Bauer, F.L., Moller, B., Partsch, H. and Pepper, P. (1989) Formal construction by transformation—Computer aided intuition guided programming. *IEEE Trans. Software Eng.*, **15,** 00–00.

Bauer, F. L. and The CIP Language Group (1985) *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L* (*Lecture Notes in Computer Science, 183*). Springer, New York.

Bauer, F. L. and The CIP System Group (1987) *The Munich Project CIP, Volume II: The Program Transformation system CIP-S)Lecture Notes in Computer Science 292*). Springer, New York.

Bauer, F.L. and Wossner, H. (1982) *Algorithmic Language and Program Development*. Springer, New York.

Bird, R. (1988) *Lecture on Constructive Functional Programming*. Technical Monograph PRG-69, Oxford University.

Broy, M., Gnatz, R. & Wirsig, M. (1979): Semantics of Nondeterminism and Noncontinuous Constructs. In Goos, G. and Hartmanis, H. (eds), *Program Construction* (*Lecture Notes in Computer Science 69*), pp. 563–592. Springer, New York.

Broy, M. and Pepper, P. (1982) Combining algebraic and algorithmic reasoning: an approach to the Schorr–Waite algorithm. *Trans. Programming Lang. Syst.*, **4,**, 00–00.

Bull, T. (1990): An Introduction to the WSL program transformer. *Conference on Software Maintenance*, San Diego, CA.

Church, A. (1951) The calculi of lambda conversion. *Ann. Math. Studies*, **6,** 00–00.

Dijkstra, E.W. (1976) *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ.

Engeler, E. (1968) *Formal Languages: Automata and Structures.* Markham, Chicago, IL.

Hoare, C. A. R., Hayes, I. J., Jifeng, H. E., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sørensen, I. H., Spivey, J. M. and Sufrin, B. A. (1987) Laws of programming. *Commun. ACM*, **30**, 672–686.

Jones, C. B. (1986) *Systematic Software Development using VDM*. Prentice-Hall, Englewood Cliffs, NJ.

Jones, C. B., Jones, K. D., Lindsay, P. A. and Moore, R. (1991) *mural: A Formal Development Support System*. Springer, New York.

Jorring & Scherlis (1987) Deriving and Using Destructive Data Types. In Meertens, L. G. L. T. (ed.), *Program Specification and Transformation: Proc. IFIP TC2/WG 2.1 Working Conf.* North-Holland, Amsterdam.

Karp, C. R. (1964) *Languages with Expressions of Infinite Length*. North-Holland, Amsterdam.

Knuth, D. E. (1974) Structured programming with the GOTO Statement. *Comput. Surv.*, **6**, 261–301.

Knuth, D. K. (1968): *Fundamental Algorithms. (The Art of Computer Programming 1)*. Addison Wesley, Reading, MA.

Majester, M. E. (1977) Limits of the 'algebraic' specification of abstract data types. *SIGPLAN Notices*, **12**, 37–42.

Morgan, C. (1990) *Programming from Specfications*. Prentice-Hall, Englewood Cliffs, NJ.

Morgan, C. C. (1988) The specification statement. *Trans. Programming Lang. Syst.*, **10**, 403–419.

Morgan, C. C., Robinson, K. and Gardiner, P. (1988) *On the Refinement Calculus*. Technical Monograph PRG-70, Oxford University.

Neilson, M., Havelund, K., Wagner, K. R. and Saaman, E. (1989) The RAISE language, method and tools. *Formal Aspects Comput.*, **1**, 85–114.

Partsch, H. (1984): The CIP Transformation System. In Pepper, P. (ed.), *Program Transformation and Programming Environments*, pp. 305–323. Springer, New York.

Pepper, P. (1979) A study on transformational Semantics. In Goos, G. and Hartmanis, H. (eds), *Program Construction (Lecture Notes in Computer Science 69)* pp. 232–405. Springer, New York.

Priestly, H. A. and Ward, M. (1993) A multipurpose backtracking algorithm. *J. Symb. Comput.*, to appear.

Sennett, C. T. (1990) Using refinement to convince: lessons learned from a case study. *Refinement Workshop*, Hursley Park, Winchester.

Ward, M. (1989) *Proving Program Refinements and Transformations*. DPhil Thesis Oxford University.

Ward, M. (1991) *A recursion removal theorem–Proof and Applications*. Technical Report, Durham University.

Ward, M. (1992a) A recursion removal theorem. Springer, New York.

Ward, M. (1992b) Derivation of data intensive algorithms by formal transformation. *IEEE Trans. Software Eng.*, **??**, 00–00.

Ward, M. (1993a) Foundations for a practical theory of program refinement and transformation. In *Formal Aspects of Computing*. Springer, New York.

Ward, M. (1993b) Abstracting a specification from code. *J. Software maintenance: Res. Practice*, **5**, 101–122.

Ward, M. (1994a): Language oriented programming. Software concepts and tools, to appear.

Ward, M. (1994b) Specfications from source code—alchemists' dream or practical reality? *Fourth Reengineering Forum*, victoria, Canada.

Ward, M. and Bennett, K.H. (1993) A practical program transformation system for reverse engineering. *Working Conf. Reverse Engineering*, Baltimore, MA.

Ward, M. and Bennett, K. H. (1994) A practical solution to reverse engineering legacy systems using formal methods. *IEEE Comput.*, submitted.

Ward, M., Calliss, F. W. and Munro, M. (1989) The maintainer's assistant. *Conf. Software Maintenance*, Miami, FL.

Wile, D. (1981) Type transformations. *IEEE Trans. Software Eng.*, **7**, 00–00.

Wossner, H., Pepper, P., Partsch, H., and Pepper, P., Partsch, H. and Bauer, F.L. (1979) Special transformation techniques. In Goos, G. and Hartmanis, H. (Eds) *Program Construction (Lecture Notes in Computer Science 69)* pp. 290–321. Springer, New York.

Younger, E. J. and Ward, M. (1993) Inverse engineering a simple real time program, *J. Software Maintenance: Res. Practice*, to appear.