Reverse Engineering through Formal Transformation

Knuths "Polynomial Addition" Algorithm

M.P. Ward
Martin.Ward@durham.ac.uk
Computer Science Dept
Science Labs
South Rd
Durham DH1 3LE
Phone: 091 374 3655

August 12, 1994

Abstract

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.

1 Introduction

There has been much research in recent years on the formal development of programs by refining a specification to an executable program via a sequence of intermediate stages, where each stage is proved to be equivalent to the previous one, and hence the final program is a correct implementation of the specification. However, there has been very little work on applying program transformations to reverse-engineering and program understanding. This may be because of the considerable technical difficulties involved: in particular, a refinement method has total control over the structure and organisation of the final program, while a reverse-engineering method has to cope with any code that gets thrown at it: including unstructured ("spaghetti") code, poor documentation, misuse of data structures, programming "tricks", and undiscovered errors. 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 by 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 pre-conditions and infinitary logic, and described in Ward (1989), Ward (1993) 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 Priesley & 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 (1993) 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 & Ward (1993).

In this paper we treat a much more challenging example than the one in Ward (1993): 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, Robinson & Gardiner (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 (for example, 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. (1986), Bauer & (The CIP Language Group) (1985), Bauer & (The CIP System Group) (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 conditional (if . . . then) statement. This language is provided with a set of "axiomatic transformations" consisting of: $\alpha$, $\beta$-and $\eta$-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, Gnatz & Wirsig (1979), Pepper (1979), Wosnner et al. (1979) and Bauer & Wosnner (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 (for example, Bauer et al. (1986), 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 (1993).

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 reverse 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 goto, and high-level constructs, including nonexecutable 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 of 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, Calliss & Munro (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 (for example, 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, Calliss & Munro (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 assembly 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 & Bennett (1993), Ward & Bennett (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 \texttt{goto}s 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 Priestley & Ward (1993), Ward (1992b) for examples of program development in WSL using formal transformations. Within Fermat, transformations are themselves coded in an extension of WSL called \texttt{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 don't 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 $S_1 \approx S_2$ if statements $S_1$ and $S_2$ 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 nondeterministic behaviour (for example, 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, Robinson & Gardiner (1988) and Back (1980) for a description of refinement. We write $S_1 \preceq S_2$ if $S_2$ is a refinement of $S_1$, or if $S_1$ is an abstraction of $S_2$.

2.1 Syntax and Semantics

The syntax and semantics of WSL are described in Priestley & Ward (1993), Ward (1989), Ward (1993), Ward (1993) 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:

- $\ell(a)$ denotes the length of the sequence $a$;
- $a[i]$ is the $i$th element $a_i$;
- $a[i..j]$ denotes the subsequence $\langle a_i, a_{i+1}, \ldots, a_j \rangle$;
- $\text{last}(a)$ denotes the element $a[\ell(a)]$;
- $\text{butlast}(a)$ denotes the subsequence $a[1..\ell(a)-1]$;
- 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 $x \leftarrow a$ sets $x$ to $a_1$ and $a$ to $\langle a_2, a_3, \ldots, a_n \rangle$;
- The statement $a \leftarrow x$ sets $a$ to $\langle x, a_1, a_2, \ldots, a_n \rangle$;
- The statement $x \leftarrow x$ 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 $a + b$.

Most of the constructs in WSL, for example if statements, while 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) $y_1, \ldots, y_n := y'_1, \ldots, y'_n$. The statement assigns new values to the variables $x_1, \ldots, x_n$. In the formula $Q$, $x_i$ represent the old values and $y'_i$ represent the new values. The new values are chosen so that $Q$ will be true, then they are assigned to the variables. If there are several sets of values which satisfy $Q$ then one set is chosen nondeterministically. If there are no values which satisfy $Q$ then the statement does not terminate. For example, the assignment $\langle x \rangle := \langle x' \rangle$, $x = 2x'$ halves $x$ if it is even and aborts if $x$ is odd. If the
The simple assignment \( (x_1, \ldots, x_n) := (e_1, \ldots, e_n) \) assigns the values of the expressions \( e_i \) to the variables \( x_i \). The assignments are carried out simultaneously, so for example \( (x, y) := (y, x) \) swaps the values of \( x \) and \( y \). The single assignment \( (x) := (e) \) can be abbreviated to \( x := e \).

The local variable statement \( \text{var } x : S \end \) 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 initialised using the notation \( \text{var } (x_1) := (e_1), \ldots, (x_n) := (e_n) : S \end \).

An action is a parameterless procedure acting on global variables (cf. Arsac (1982a), Arsac (1982b)). 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 \( \text{call } X \) 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 \( A_1 \):

\[
\text{actions } A_1:
A_1 \equiv S_1.
A_2 \equiv S_2.
\ldots
A_n \equiv S_n. \end actions
\]

For example, this action system is equivalent to the while loop \( \text{while } B \text{ do } S \od \):

\[
\text{actions } A:
A \equiv
\text{if } \neg B \text{ then call } Z \text{ fl; S; call } A. \end actions
\]

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 nondeterministic iteration over \( X \) is written \( \text{for } i \in X \text{ do } S \od \). 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:

\[
\text{var } (i := 0, X' := X):\end actions
\]

\[
\text{while } X' \neq \emptyset \text{ do } i := i'. (i' \in X'); X' := X' \setminus \{i\}; S \od \end actions\]

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

\[
\text{var } (i := 0, X' := X):\end actions
\]

\[
\text{while } X' \neq \emptyset \text{ do } i \in X'; S \od \end actions\]

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 \( S_1 \) is a proper sequence, i.e. it cannot cause termination of an enclosing loop. Then if \( S_1 \) appears at the beginning of a loop body, we can take it out of the loop provided we insert a second copy of \( S_1 \) at the end of the loop. In other words, the statement \( \text{do } S_1; S_2 \od \) is equivalent to \( S_1; \text{ do } S_2; S_1 \od \).

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

\[
\text{do } S_1; \text{ if } B \text{ then exit fl; S}_2 \od \approx \text{S}_1; \text{ while } B \text{ do } S_2; \text{ S}_1 \od
\]

when \( S_1 \) and \( S_2 \) 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 \( S_1 \).

3.2 Loop Unrolling

The simplest loop unrolling transformation is the following:

\[
\text{while } B \text{ do } S \od \approx \text{if } B \text{ then } S; \text{ while } B \text{ do } S \od \text{ fl}
\]

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 \( Q \):

\[
\text{while } B \text{ do } S \od \approx \text{while } B \text{ do } S; \text{ if } B \land Q \text{ then } S \text{ fl od}
\]

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:

\[
\text{while } B \text{ do } S \text{ od} \approx \text{while } B \text{ do } S; \text{ while } B \land Q \text{ do } S \text{ od od}
\]

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

\[
\text{while } B \text{ do } \begin{cases} 
\text{if } B_1 \text{ then } S_1 \\
\text{else if } B_2 \text{ then } S_2 \\
\text{else } S_3 \text{ fi od}
\end{cases}
\]

where executing \( S_1 \) makes \( B_2 \) true and \( B_1 \) false (i.e. \( \{ B_1 \}; S_1 \lessdot \{ B_1 \}; S_1; \{ B_2 \land \neg B_1 \} \)), and \( S_2 \) is the only statement which can affect condition \( B_1 \). If we selectively unroll after \( S_2 \), then \( B \) will still be true, \( B_1 \) will be false, and \( B_2 \) will be true. So we can prune the inserted if statement to get:

\[
\text{while } B \text{ do } \begin{cases} 
\text{if } B_1 \text{ then } S_1 \\
\text{else if } B_2 \text{ then } S_2; S_3 \\
\text{else } S_3 \text{ fi od}
\end{cases}
\]

Since \( S_1 \) does not affect \( B \), we can selectively unroll the entire loop after \( S_1 \) under the condition \( B \land B_1 \) (which reduces to \( B_1 \) since \( B \) is true initially and not affected by \( S_1 \)):

\[
\text{while } B \text{ do } \begin{cases} 
\text{if } B_1 \text{ then } S_1; \text{ while } B_1 \text{ do } S_1 \text{ od} \\
\text{else if } B_2 \text{ then } S_2; S_3 \\
\text{else } S_3 \text{ fi od}
\end{cases}
\]

Convert the else to else if: take out \( S_3 \), and roll up one step of the inner while loop to get:

\[
\text{while } B \text{ do } \begin{cases} 
\text{while } B_1 \text{ do } S_1 \text{ od} \\
\text{if } \neg B_2 \text{ then } S_2 \text{ fi:}
\end{cases}
\]

\[ S_3 \text{ od} \]

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:

\[
\text{proc } F[x] \equiv \begin{cases} 
\text{actions } A_1: \\
A_1 \equiv S_1; \\
A_1 \equiv S_1; \\
A_1 \equiv S_1; \\
A_1 \equiv S_1;
\end{cases}
\]

\[ \ldots B_j \equiv \]

\[
\text{S}_j; \ F(g_{j1}[x]); \ S_{j1}; \ F(g_{j2}[x]); \ldots \ F(g_{j_{n_j}}[x]); \ S_{j_{n_j}}; \ldots \text{endactions.}
\]

where the statements \( S_{j1}, \ldots, S_{j_{n_j}} \), 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 \( S_{j0}, S_{j1}, \ldots, S_{j_{n_j} - 1} \) contain no action calls. There are \( M + N \) actions in total: \( A_1, \ldots, A_M, B_1, \ldots, B_N \). Note that the 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 \( \langle 0, c \rangle \) onto \( K \) (where \( c \) is the parameter required for the recursive call). Execution of the statements \( S_{j1, k} \) also has to be postponed (since they occur between recursive calls), we record the postponement of \( S_{j1, k} \) by pushing \( \langle j, k, x \rangle \) 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 \( m \):

\[
\text{proc } F[x] \equiv \begin{cases} 
\text{var } K := \emptyset, m; \\
\text{actions } A_1: \\
A_1 \equiv S_1; \text{[call } \hat{F}/\text{call }] Z, \\
\ldots B_k \equiv \begin{cases} 
S_{0}; \ K := \langle \emptyset, g_{j1}[x], \langle j, 1, x \rangle, \langle \emptyset, g_{j2}[x], \ldots \langle \emptyset, g_{j_{n_j}}[x] \rangle \rangle \# K; \\
\langle \text{call } \hat{F} \rangle, \\
\ldots \text{endactions.}
\end{cases}
\end{cases}
\]

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:

\[
\text{proc } F[x] \equiv \begin{cases} 
\text{actions } A_1: \\
A_1 \equiv S_1; \\
A_1 \equiv S_1; \\
A_1 \equiv S_1; \\
\ldots B_j \equiv \]

\[
\text{S}_j; \ F(g_{j1}[x]); \ S_{j1}; \ F(g_{j2}[x]); \ldots \ F(g_{j_{n_j}}[x]); \ S_{j_{n_j}}; \ldots \text{endactions.}
\]
var K := \{\}; m := 0;

actions A₁:

\ldots
A₁ \equiv
S₁[call \hat{F}/call Z].

\ldots
B₁ \equiv
S₁₀; K := \langle\langle j, 1\rangle, x, \langle 0, g_{j1}(x)\rangle, \ldots, \\
\langle 0, g_{j1}(x)\rangle, \langle 0, n₁\rangle, x \rangle \mid+ K; x := g₁(x); \text{call A₁}.

\ldots
\hat{F} \equiv
if K = \{\}
then call Z
else \langle m, x \rangle \triangleq K;
\text{if} m = 0 \rightarrow \text{call A₁}
fi \ldots
\text{if} m = \langle j, kl \rangle \rightarrow S₁[call \hat{F}/call Z]
\ldots \text{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:

\text{proc} F(x) \equiv \text{if} B₁ \text{ then } S₁; F(y)
else S₂ fi.

where S₁ and S₂ may both call F(). The terminal call can be implemented with a while loop as follows:

\text{proc} F(x) \equiv \text{while } B₁ \text{ do } S₁; x := y \text{ od; } S₂.

A slightly more complicated example:

\text{proc} F(x) \equiv \text{if} B₁ \text{ then if } B₂ \text{ then } S₁; F(y)
else S₂ fi

\text{else S₃ fi.}

is equivalent to:

\text{proc} F(x) \equiv \text{while } B₁ \land B₂ \text{ do } S₁; x := y \text{ od; } S₂ fi

\text{if } B₁ \text{ then } S₂ \text{ else } S₃ fi.

4 Polynomial Addition

A polynomial P in several variables may be expressed as:

\begin{equation}
P = \sum_{0 \leq j \leq n} g_j x^e_j \tag{1}
\end{equation}

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 \(0 \leq j \leq n\), \(g_j\) (the coefficient of the \(i\)th 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 \(\Lambda\) (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 polynomial) or the primary variable.

E[P] is zero.

L[P] points to P.

R[P] points to P.

U[P] is \(\Lambda\): an otherwise unused pointer value.

D[P] is either \(\Lambda\) (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 \textit{gotos}. We have translated the algorithm into WSL, using an action system with one action for each label.

ADD \equiv
\begin{align*}
&\text{if } D[P] = \Lambda \\
&\text{then while } D[Q] \neq \Lambda \text{ do } Q := D[Q] \text{ od; } \\
&\text{call A₁} \\
&\text{else if } D[Q] = \Lambda \lor C[Q] < C[P] \text{ then call A₂} \\
\end{align*}
else C[Q] = C[P]
    then P := D[P]; Q := D[Q]; call ADD
else Q := D[Q]; call ADD fi fi.

A_2 \equiv
r \notin \text{ avail};
if s \neq \lambda then do U[s] := r; s := R[s];
if E[s] = 0 then exit fi od fi;
U[r] := C[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.

A_3 \equiv
E[Q] \neq 0 \Rightarrow [E[P] = E[Q] \land C[U[P]] = C[U[Q]]];
C[Q] := C[Q] + C[P];
if C[Q] = 0 \land E[Q] \neq 0 then call A_6 fi;
if E[Q] = 0 then call A_7 fi;
call A_4.

A_4 \equiv
P := L[P];
if E[P] = 0 then call A_6
else do Q := L[Q];
if E[Q] \leq E[P] then exit fi od;
if E[Q] = E[P] then call ADD fi fi;
call A_5.

A_5 \equiv
r \notin \text{ 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] \land C[U[P]] = C[U[Q]]];
r := Q; Q := R[r]; s := L[r]; R[s] := Q;
L[Q] := s; avail \leftarrow r;
if E[L[P]] = 0 \land 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 \leftarrow 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 \land C[Q] = 0 \land 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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{call-graph.png}
\caption{The Call Graph of Knuth's Polynomial Addition Algorithm}
\end{figure}

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.

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 “localised”. 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 ≡
if D[P] - A
then while D[Q] /\ D[Q] /\ do Q := D[Q] od; call A3;
else if D[Q] - A /\ (C[Q] < C[P]) then call A2
else C[Q] = C[P]
then P := D[P], Q := D[Q]; call ADD
else Q := D[Q]; call ADD fi.
A2 ≡
Insert_Below_Q; call ADD.
A3 ≡
(E[Q] \not= 0 \Rightarrow (E[Q] - (C[U[P]] - (C[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 A8
else Move_Left_Q;
    if E[P] - E[Q] then call ADD fi
    call A5.
A5 ≡
Insert_to_Right; call ADD.
A6 ≡
P := L[P];
call A7.
A7 ≡
if U[P] - A
then call A11
    else Move_Up_Q;
    call A4 fi.
A8 ≡
(E[P] - E[Q] \land C[Q] = C[Q];
Delete_Zero_Term;
if (E[L[P]] - 0) /\ (Q - L[Q])
then call A8
else call A4 fi.
A9 ≡
Delete_Const_Poly; call A10.
A10 ≡
if ((D[Q] - A) /\ (C[Q] - 0)) /\ (E[Q] - 0)
then P := U[P]; call A8
else call A4 fi.
A11 ≡
while U[Q] /\ U[Q] /\ do Q := U[Q];
call Z
endactions

where

proc Insert_Below_Q ≡

r \not= avail; s := D[Q];
if s \not= A
then do U[s] := r; r := R[s];
if E[s] = 0 then exit fi;
(C[r] := C[Q], E[r] := 0);
(C[Q] := C[P], D[Q] := r).

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

proc Insert_to_Right ≡

r \not= avail;
L[R[Q]] := r, R[Q] := r;
(E[r] := E[P], C[Q] := C[P];
Q := r.

proc Move_Up_Q ≡
while C[U[Q]] \not= C[U[P]] do Q := U[Q];

proc Delete_Zero_Term ≡
t := Q;
(Q := R[Q], s := L[r]);
R[s] := Q, L[Q] := s;
avail := 1 r.

proc Delete_Const_Poly ≡
t := Q;
if C[Q] \not= 0 then exit 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] - A
then while D[Q] /\ D[Q] /\ do Q := D[Q];

(E[Q] \not= 0
\Rightarrow (E[Q] - (C[U[P]] - (C[Q]))
(C[Q] := C[Q] + C[P];
if (U[P] - A \land E[Q] - 0)
\land (C[Q] \not= 0 \lor E[Q] - 0)
then while U[Q] /\ U[Q] /\ do Q := U[Q];

exit fi;
do (E[P] - E[Q] \land C[Q] = C[Q]);
Delete_Zero_Term;
if (E[L[P]] - 0 \lor (Q - L[Q])
then exit fi;
Delete_Const_Poly;
P := U[P];
if (U[P] - A)
\land (C[Q] \not= 0 \lor D[Q] \not= A \lor E[Q] - 0)
then while U[Q] /\ U[Q] /\ do Q := U[Q];

exit fi;
do (D[Q] - A) \land (C[Q] < C[P]);
then Insert_Below_Q

end.
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:

\[
\text{do while } D[P] \neq \Lambda \text{ ... od do } \\
\text{if } D[Q] \neq \Lambda \vee C[Q] < C[P] \text{ then Insert_Below_Q} \\
\text{else if } C[Q] < C[P] \text{ then } P := D[P], Q := D[Q] \\
\text{else } Q := D[Q] \text{ fi od} .
\]

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:

\[
\text{while } D[P] \neq \Lambda \text{ do } \\
\text{while } D[Q] \neq \Lambda \vee C[Q] > C[P] \text{ do } Q := D[Q] \text{ od} ; \\
\text{if } D[Q] = \Lambda \vee C[Q] < C[P] \text{ then Insert_Below_Q fi}; \\
P := D[P], Q := D[Q] \text{ od} .
\]

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:

\[
\text{do } P := L[P]; \\
\text{if } E[P] \neq 0 \text{ then exit fi}; \\
P := U[P]; \\
\text{if } U[P] = \Lambda \text{ then exit(2) fi}; \\
\text{Move_Up_Q od};
\]

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:

\[
\text{do if } E[L[P]] \neq 0 \text{ then exit fi}; \\
P := U[P]; \\
\text{if } U[P] = \Lambda \text{ then exit(2) fi}; \\
\text{Move_Up_Q od}; \\
P := L[P];
\]

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:

\[
\text{do while } D[P] \neq \Lambda \text{ do } \\
\text{while } D[Q] \neq \Lambda \wedge C[Q] > C[P] \text{ do } Q := D[Q] \text{ od}; \\
\text{if } D[Q] = \Lambda \wedge C[Q] < C[P] \text{ then Insert_Below_Q fi}; \\
P := D[P], Q := D[Q] \text{ od}; \\
\text{while } D[Q] \neq \Lambda \wedge C[Q] > C[P] \text{ do } Q := D[Q] \text{ od}; \\
\text{if } D[Q] = \Lambda \wedge C[Q] < C[P] \text{ then Insert_Below_Q fi}; \\
P := D[P], Q := D[Q] \text{ od};
\]

for the other two cases, the loop test is redundant.
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 occurs. 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 $\hat{A}$:**

$\hat{A} \equiv$

- **$\hat{A}_1$**
  - **Append**
  - **Do**
  - **while**
  - $D[Q] \neq \lambda$ and $C[Q] \neq C[\lambda]$
  - $Q := D[Q] od$
  - \textbf{if} $D[\lambda] = \lambda$
  - \textbf{then} $\hat{A}_1$
  - \textbf{else} $\hat{A}_2$

**actions $\hat{A}_2$**

- **Call**
- **Do**
- **while**
- $D[Q] \neq \lambda$ and $C[Q] \neq C[\lambda]$
- $Q := D[Q] od$
- \textbf{if} $D[\lambda] = \lambda$
- \textbf{then} $\hat{A}_2$
- \textbf{else} $\hat{A}_3$

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 $\hat{A}$**

$\hat{A} \equiv$

- **Call**
- **Do**
- **while**
- $D[Q] \neq \lambda$ and $C[Q] \neq C[\lambda]$
- $Q := D[Q] od$
- \textbf{if} $D[\lambda] = \lambda$
- \textbf{then} $\hat{A}_1$
- \textbf{else} $\hat{A}_2$

Within the two “finishing” actions, $\hat{A}_1$ and $\hat{A}_2$, the pointer $P$ is moved up and $U[P]$ tested against $\lambda$. 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**

$\textbf{proc ADD} \equiv$

- **Call**
- **Do**
- **while**
- $D[Q] \neq \lambda$ and $C[Q] \neq C[\lambda]$
- $Q := D[Q] od$
- **if** $D[\lambda] = \lambda$
- **then** deal with a constant polynomial
- **else** set up a polynomial in $Q$
- **do** ADD; Add a pair of terms;
- **deal** with a zero result;
- **do** ADD; Add a pair of terms;
- **set** up a term in $Q od$
- **Deal** with a constant polynomial result;
- **Move** up $Q$ if needed.

**endactions**
if $U[P] = \lambda$

then while $U(Q) \neq \lambda$ do $Q := U(Q)$ od;

 call $Z$ fl;

if $\neg \text{flag}$ then Move-Up $Q$ fl;

 call $\lambda$

else flag := false; call $B$ fl.

end.

\begin{align*}
& P := L[P];
& \text{Move Left} Q;
& \text{if } E[P] \neq E[Q] \text{ then Insert To Right } Q; \text{fi};
& \text{call} \lambda.
\end{align*}\\
endactions

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

begin
if $D[P] = \lambda$

then while $D[Q] \neq \lambda$ do $Q := D[Q]$ od;

$C[Q] := C[Q] + C[P]$;

while $U[Q] \neq \lambda$ do $Q := U[Q]$ od

else ADD fl.

where

proc ADD $\equiv$

if $D[P] = \lambda$

then while $D[Q] \neq \lambda$ do $Q := D[Q]$ od;

$[E[Q] \neq 0 \Rightarrow [E[P] = E[Q] \land C[U[P]] = C[U[Q]]] ;$

$C[Q] := C[Q] + C[P]$;

if $E[Q] = 0$ then Move-Up $Q$ fl;

flag := true

else while $D[Q] \neq \lambda \land C[Q] > C[P]$ do

$Q := D[Q]$ od;

if $D[Q] = \lambda \lor C[Q] < C[P]$ then Insert Below $Q$ fl;

$P := D[P]; Q := D[Q]$;

do ADD;

if flag $\land D[Q] = \lambda \land C[Q] = 0 \land E[Q] \neq 0$

then $[E[P] = E[Q] \land C[U[P]] = C[U[Q]]]$;

Delete Zero Term

else flag := false fi;

if $E[L[P]] = 0$ then exit fl;

flag := false;

$P := L[P];$

\text{Move Left} Q;

\text{if } E[P] = E[Q] \text{ then Insert To Right } Q \text{ od};$

if $\neg \text{flag} \land Q = L[Q]$

then Delete Const Poly

else flag := false fi;

$P := L[P];$

if $U[P] = \lambda$

then while $U[Q] \neq \lambda$ do $Q := U[Q]$ od

else flag $\Rightarrow$ Move-Up $Q$ fl 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 $P$ to its original value. It is also easier with the recursive version to prove that the flag can be removed. First we prove that:

$\neg \text{flag} \Rightarrow \neg \{D[Q] = \lambda \land C[Q] = 0 \land E[Q] \neq 0\}$
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 optimisation (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 $D[P] = \lambda$

then while $D[Q] \neq \lambda$ do $Q := D[Q]$ od;

$C[Q] := C[Q] + C[P]$;

else ADD fl.

where

proc ADD $\equiv$

if $D[P] = \lambda$

then while $D[Q] \neq \lambda$ do $Q := D[Q]$ od;

$[E[Q] \neq 0 \Rightarrow [E[P] = E[Q] \land C[U[P]] = C[U[Q]]] ;$

$C[Q] := C[Q] + C[P]$;

if $E[Q] = 0$ then Move-Up $Q$ fl;

$\text{if } E[P] = E[Q] \land C[U[P]] = C[U[Q]]$;

else while $D[Q] \neq \lambda \land C[Q] > C[P]$ do

$Q := D[Q]$ od;

\text{ADD}$;

if $\neg \text{flag} \land D[Q] = \lambda \land C[Q] = 0 \land E[Q] \neq 0$

then $[E[P] = E[Q] \land C[U[P]] = C[U[Q]]]$;

\text{Delete Zero Term}$

else flag := false fi;

$P := D[P]; Q := D[Q]$;

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 not carry out a small number of extra tests, which Knuth's version would allow with the use of tortuous control flow. However, it also avoids all 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 \text{Insert\_Below\_Q}. We have carried out a 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 be 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 15% 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 3% 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 \( P \) 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 \( U[U[P]] = U[P] \)). 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 \( Q := U[Q] \) until \( U[Q] = \Lambda \) again. The only node in the \( Q \) tree with a \( U \) value of \( \Lambda \) 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 \( C[Q] = C[U[P]] \). 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 \( Q \) 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:

\[
\begin{align*}
\text{if } D[P] - \Lambda & \text{ then } \text{while } D[Q] - \Lambda \text{ do } Q := D[Q] \text{ od}; \\
& C[Q] := C[Q] + C[P]; \\
& \text{while } U[Q] - \Lambda \text{ do } Q := U[Q] \text{ od} \\
& \text{ADD}(P, Q); \\
& \text{end}
\end{align*}
\]

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

\[
\begin{align*}
\text{var } Q_0 & := Q; \\
\text{while } D[Q] - \Lambda \text{ do } Q := D[Q] \text{ od}; \\
& C[Q] := C[Q] + C[P]; \\
& Q := Q_0 \text{ end}
\end{align*}
\]

which gives the correct result.

6 Introduce Abstract Data Types

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

\[
p = \begin{cases} 
(v) & \text{if } p \text{ is a constant polynomial} \\
(x, t) & \text{otherwise}
\end{cases}
\]

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 \( (c, c) \) where \( c \) 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. \( \langle 0,0 \rangle \)). 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:

\[
  t = \langle \langle 0, c_0 \rangle, \langle c_1, c_1 \rangle, \ldots, \langle c_k, c_k \rangle \rangle
\]

where \( k \geq 1 \) and \( 0 < c_1 < \cdots < c_k \) and \( c_i \neq 0 \) for \( 1 \leq i \leq n \).

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

**Definition 6.1 Abstract Polynomials.** Suppose we have an ordered set \( \text{VARS} \) of variable names, and a set \( \text{VALS} \) of values. Define:

\[
  \text{POLYS} =_{_{df}} \bigcup_{n \geq 0} \text{POLYS}^n
\]

where

\[
  \text{POLYS}_n =_{_{df}} \{ \langle v \rangle \mid v \in \text{VALS} \}
\]

is the set of constant polynomials, and for each \( n \geq 0 \)

\[
  \text{POLYS}^{n+1} =_{_{df}} \text{POLYS}^n \cup \{ \langle x, t \rangle \mid x \in \text{VARS} \land t \in \text{TERMS}^n \land \forall i, 1 \leq i \leq \ell(t) . \forall y \in \text{vals}(t[i][2]), y < x \}
\]

The set \( \text{TERMS}^n \) is the set of term lists which use elements of \( \text{POLYS}^n \) as coefficients:

\[
  \text{TERMS}^n =_{_{df}} \{ \langle \langle c_0, c_0 \rangle, \langle c_1, c_1 \rangle, \ldots, \langle c_k, c_k \rangle \rangle \mid k > 0 \land \forall i, 1 \leq i \leq k . c_i \in \text{POLYS}^n \land 0 < c_1 < \cdots < c_k \}
\]

The function \( \text{vars} \langle p \rangle \) returns the set of variables used in polynomial \( p \):

\[
  \text{vars} \langle p \rangle =_{_{df}} \begin{cases} \emptyset & \text{if } p = \langle v \rangle \\ \{ x \} \cup \bigcup_{0 \leq i \leq k} \text{vars} \langle c_i \rangle & \text{otherwise} \end{cases}
\]

Now we can define the abstraction function \( \text{poly} \langle P \rangle \) 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:**

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

where

\[
  \text{terms} \langle P \rangle =_{_{df}} \text{term} \ast \text{list} \langle \text{P}, \text{L}, \text{P} \rangle
\]

The notation \( \text{term} \ast \text{list} \) denotes the list formed by applying the function \( \text{term} \) to each element of list \( \text{list} \). The function \( \text{term} \) is defined:

\[
  \text{term} \langle P \rangle =_{_{df}} \langle E \langle P \rangle, \text{polynomial}(C \langle P \rangle) \rangle
\]

For abstract polynomials we define the following functions:

\[
  \text{const}\langle p \rangle =_{_{df}} \begin{cases} \text{true} & \text{if } p \text{ is constant, i.e. } \ell(p) = 1 \\ \text{false} & \text{otherwise} \end{cases}
\]

\[
  \nu(p) =_{_{df}} \text{variable of } p = p[1]
\]

\[
  \text{c}(p) =_{_{df}} \text{value of the constant poly } = p[1]
\]

\[
  \text{T}(p) =_{_{df}} \text{list of terms for } p = p[2]
\]

\[
  \text{c}(i)(p) =_{_{df}} \text{coefficient of the } i\text{th term } = p[2][i][1]
\]

\[
  \text{c}(i)(p) =_{_{df}} \text{coefficient of the } i\text{th term } = p[2][i][2]
\]

**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:

\[
  p = \text{poly}(P) \land q = \text{poly}(Q)
\]

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

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

\[
  \text{proc ADD}(P,Q) \equiv
\]

\[
  \text{var } p_0 := p, q_0 := q:
  \text{if } D[P] = \Lambda
    \text{then if } D[Q] \neq \Lambda
      \text{then } Q := D[Q]; q := c_0(q_0); \text{ADD}(P,Q);
      r := \langle \nu(q) \rangle, \langle \nu(q) \rangle, \ell(T(q) \langle 2 \rangle) + T(q) \langle 2 \rangle
    \text{else } C[Q] := C[Q] + C[P];
    r := \langle \nu(q) \rangle + \text{c}(p) \fi
  \text{else if } D[Q] \neq \Lambda \land C[Q] > C[P]
    \text{then Insert_Below_Q; } q := \langle \nu(p), \langle \nu(q) \rangle \rangle \fi
  \text{else if } D[Q] = \Lambda \land C[Q] > C[P]
    \text{then Insert_Below_P; } P := D[P]; Q := D[Q];
  \text{var } i := 1, j := 1, t := T(q_0):
    \text{do } p := c_1(p_0); q := t[j];
    \text{ADD}(P,Q);
    \text{if } D[Q] = \Lambda \land C[Q] = 0 \land E[Q] \neq \emptyset
      \text{then Delete_Zero_Term;}
      t := t[i][j] + t[i][j + 1];
    \text{else } t[i][j] := r \fi
    P := t[P]; i := i - 1;
    \text{if } i = 0 \text{ then } t := T(P) \fi
    \text{if } E[P] = 0 \text{ then exit fi;}
    \text{do } Q := L[Q]; j := j - 1;
    \text{if } j = 0 \text{ then } j := T(q) \fi
    \text{if } E[Q] \leq E[P] \text{ then exit fi od;}
    \text{if } E[P] \neq E[Q]
      \text{then Insert_To_Right;}
      t := t[i][j] + \langle \nu(p), \langle 0 \rangle \rangle
      + t[i][j + 1] \fi \text{ od;}
\]
if Q = 1[Q] then Delete_Const_Poly; r := t[1][2];
else r := (v[p], t) 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] = A is equivalent to the test const[P] given that p = 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 & Pepper (1982), Jorring & Scherlis (1987), Wile (1981). Assuming that what we are really interested in is the result for a given v 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 p0 := p, q0 := q:
if const?(p)
then if ~const?(q)
then q := c0(q0); add(p, q);
r := (v[p], (c0(q0), r)) + T(q0)[2..]
else r := (c0(q0) + c1(p0)) fi
else ~const?(q) \& v[q] > v[p]
then q := c0(q0); add(p, q);
r := (v[p], (c0(q0), r)) + T(q0)[2..]
else ~const?(q) \& v[q] < v[p]
then q := (v[p], (0, q0)) fi
var i := -1, j := -1, t := T(q0):
do p := c1(p0); q := t[i];
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 := 1 fi;
if c1(p0) = 0 then exit fi;
do j := j - 1; if j = 0 then j := t[i] fi;
if t[j][1] \leq c1(p0) then exit fi;
else r := (v[p], t) fi end;
if t[i] = 1 then r := (t[i][2])
else r := (v[p], t) 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 e1[0] = e1[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(q0):
add(c1(p0), c1(q0));
t[1][1] := r;
i := t[1][2]; j := t[i];
while i > 1 do
while t[i][1] > c1(p0) do j := i - 1 od;
if c1(p0) = t[i][1] then t := t[1..j-1] + (c1(p0), 0)) + t[j+1..]
add(c1(p0), t[i]);
if const?(r) \& c(r) = 0
then t := t[1..j-1] + t[j+1..]
else t[i][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(tp0) into ttp, T(tq0) into tq, and deleting elements from the ends of ttp and tq once they have been dealt with. The new value of t is built up in a new variable tr, so that t is represented by ttp + tr. Since the loop adds the elements in reverse order, it makes sense to move the add(c1(p0), c1(q0)) call to the end, especially since at this point ttp = c1(p0) and tq := c1(q0):

var ttp := T(tp0), tq := T(tq0), tr := ()
if ttp > 1 do
while ttp > 1 do
last(ttt(tp0), ttp := butlast(ttp) od;
if last(tp0) \& last(tq0) then
else ttp := ttp + 1 od;
add(last(tp0), last(tq0), tr := butlast(tq0) fi;
add(last(tp0), last(tq0), tr := ttp + tpp + tr fi.

The next step is to make this while loop into a tail-recursive procedure which takes ttp and tq as arguments, and returns the result in tr. 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 := c0(q0); add(p, q);
r := (v[p], (c0(q0), r)) + T(q0)[2..]
else r := (c0(q0) + c1(p0)) fi
else ~const?(q) \& v[q] > v[p]
then q := c0(q0); add(p, q);
r := (v[p], (c0(q0), r)) + T(q0)[2..]
else ~const?(q) \& v[q] < v[p]
then q := (v[p], (0, q0)) fi
var i := -1, j := -1, t := T(q0):
do p := c1(p0); q := t[i];
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 := 1 fi;
if c1(p0) = 0 then exit fi;
do j := j - 1; if j = 0 then j := t[i] fi;
if t[j][1] \leq c1(p0) then exit fi;
else r := (v[p], t) fi end;
if t[i] = 1 then r := (t[i][2])
else r := (v[p], t) fi end.

while t[i][1] > c1(p0) do j := i - 1 od;
if c1(p0) = t[i][1] then t := t[1..j-1] + (c1(p0), 0)) + t[j+1..]
add(c1(p0), t[i]);
if const?(r) \& c(r) = 0
then t := t[1..j-1] + t[j+1..]
else t[i][1] := r fi;
i := i - 1 od end.

var ttp := T(tp0), tq := T(tq0), tr := ()
if ttp > 1 then
last(ttt(tp0), ttp := butlast(ttp) od;
if last(tp0) \& last(tq0) then
else ttp := ttp + 1 od;
add(last(tp0), last(tq0), tr := butlast(tq0) fi;
add(last(tp0), last(tq0), tr := ttp + tpp + tr fi.

proc addlist(tp, tq) ==
if ttp = 1 then
add(tp[1][2], tq[1][2]);
tr := (0, r) + ttp + tr
else if last(tp[1][1]) \& last(tq[1][1])
then tr := last(tp); addlist(tp, butlast(tp))
else if last(tp[1][1]) \& last(tq[1][1])
then tr := butlast(tp); addlist(tp, butlast(tp))
else tr := (0, r) + ttp + tr fi.

while tpp = 1 do
add(tp[1][2], tq[1][2]);
tr := (0, r) + ttp + tr
else if last(tp[1][1]) \& last(tq[1][1])
then tr := last(tp); addlist(tp, butlast(tp))
else if last(tp[1][1]) \& last(tq[1][1])
then tr := butlast(tp); addlist(tp, butlast(tp))
else tr := (0, r) + ttp + tr fi.

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(tp0) into ttp, T(tq0) into tq, and deleting elements from the ends of ttp and tq once they have been dealt with. The new value of t is built up in a new variable tr, so that t is represented by ttp + tr. Since the loop adds the elements in reverse order, it makes sense to move the add(c1(p0), c1(q0)) call to the end, especially since at this point ttp = c1(p0) and tq := c1(q0):

var ttp := T(tp0), tq := T(tq0), tr := ()
if ttp > 1 do
while ttp > 1 do
last(ttt(tp0), ttp := butlast(ttp) od;
if last(tp0) \& last(tq0) then
else ttp := ttp + 1 od;
add(last(tp0), last(tq0), tr := butlast(tq0) fi;
add(last(tp0), last(tq0), tr := ttp + tpp + tr fi.

The next step is to make this while loop into a tail-recursive procedure which takes ttp and tq as arguments, and returns the result in tr. We can apply the tail-recursion transformation of Section 3.4 to remove the inner while loop:
Finally, we can convert the procedures into the equivalent functions:

\[
\text{funct \_add\_list\(\{p, q\}\) \equiv}
\]

\[
\text{if const\(\{p\}\) then if \neg \text{const}\(\{q\}\) then \(\langle\text{add}(p, c_0(q))\rangle + T(q)[2]\) else \(c(q) + c(p)\) else if \neg \text{const}\(\{q\}\) \& \text{\(\langle v(q)\rangle + T(q)[2]\) else \(v(p)\) else if \text{const}\(\{q\}\) \& \(v(q) < v(p)\) then \text{add\_list}(T(p), T(q)) else \text{add\_list}(T(p), T(q))) \fi fl fl.}
\]

\[
\text{funct \_add\_list\(\{t_p, t_q\}\) \equiv}
\]

\[
\text{if } \ell(t_p) - 1 \text{ then } \langle\text{add}(t_p[1][2], t_q[1][2])\rangle + t_q[2]\] else if \text{last}(t_q[1][1]) > \text{last}(t_p[1][1])\ then \text{add\_list}(t_p, \text{butlast}(t_q)) + \langle\text{last}(t_q)\rangle \] else \text{var } t_r := \langle\rangle;\ then \text{last}(t_q[1][1]) \not= \text{last}(t_q[1][1]) then \text{add\_list}(t_r[1][1], t_q) else \text{add\_list}(t_r[1][1], t_q) + \langle\text{last}(t_p[1][1], t_r)\rangle \fi fl fl.}
\]

A final optimisation to \text{add\_list} is to absorb the statement \(r \leftarrow \text{add\_list}(\{2\}, r)\) into the preceding if statement and avoid adding a zero polynomial.

\[
\text{funct \_add\_list\(\{t_p, t_q\}\) \equiv}
\]

\[
\text{if } \ell(t_p) - 1 \text{ then } \langle\text{add}(t_p[1][2], t_q[1][2])\rangle + t_q[2]\] else if \text{last}(t_q[1][1]) > \text{last}(t_p[1][1])\ then \text{add\_list}(t_p, \text{butlast}(t_q)) + \langle\text{last}(t_q)\rangle \] else \text{var } t_r := \langle\rangle;\ then \text{last}(t_q[1][1]) \not= \text{last}(t_q[1][1]) then \text{add\_list}(t_r[1][1], t_q) else \text{add\_list}(t_r[1][1], t_q) + \langle\text{last}(t_p[1][1], t_r)\rangle \fi fl fl.}
\]

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

\[
\text{add\(\{p, q\}\) = \eta}\]

\[
\langle\text{add}(p, c_0(q))\rangle + T(q)[2] \text{ if } \text{const}\{p\} \& \text{const}\{q\} \text{ otherwise}
\]

\[
\text{add\_list\(\{t_p, t_q\}\) = \eta}\]

\[
\langle\text{add\_list}(T(p), T(q))\rangle + t_q[2] \text{ if } \ell(t_p) = 1 \text{ otherwise}
\]

and

\[
\text{AL\(\{t_p, t_q\}\) = \eta}\]

\[
\langle\text{add\_list}(T(p), T(q))\rangle + \langle\text{last}(t_q)\rangle \text{ if } \ell(t_p) > 1 \& \text{last}(t_q)[1] > \text{last}(t_p)[1] \text{ otherwise}
\]

and

\[
\text{AL'}\(\{t_p, t_q\}, r\) = \eta}\]

\[
\langle\text{add\_list}(\text{butlast}(t_p), t_q)\rangle + \langle\text{last}(t_q)\rangle \text{ if } \text{const}\{r\} \& \text{c}(r) = 0 \text{ otherwise}
\]

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 (1983) 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 result 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”.

**9 References**


Church, A. (1951): The Calculi of Lambda Conversion. Annals of Mathematical Studies 6


---

1The UK Science and Engineering Research Council