Program Analysis by Formal Transformation

M.P. Ward
Martin.Ward@@durham.ac.uk
Computer Science Department
University of Durham
Science Laboratories
South Rd
Durham DH1 3LE, UK

Abstract

This paper treats Knuth and Szwarcfiter's topological sorting program, presented in their paper "A Structured Program to Generate All Topological Sorting Arrangements" (Knuth and Szwarcfiter 1974), as a case study for the analysis of a program by formal transformations. This algorithm was selected for the case study because it is a particularly challenging program for any reverse engineering method. Besides a complex control flow, the program uses arrays to represent various linked lists and sets, which are manipulated in various "ingenious" ways so as to squeeze the last ounce of performance from the algorithm. Our aim is to manipulate the program, using semantics-preserving operations, to produce an abstract 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

In (Knuth and Szwarcfiter 1974) a program is presented for "topological sorting", i.e. determining all the extensions of a given partial order to a linear order. Despite the title of the paper ("A Structured Program to Generate All Topological Sorting Arrangements") , the program is, by modern standards, highly unstructured, containing jumps into and out of the middle of loops and implementing linked lists and circular lists using pointers into arrays. Our aim in this paper is to transform it into a structured program, and from there into a formal specification, by means of formal, semantics-preserving transformations.

The language and transformation techniques to be used are described in (Ward 1989) and (Ward 1994c). The techniques are used in software development (refining a specification into an efficient implementation) in (Ward 1990) and (Priestley and Ward 1994), and a simple example of program analysis by transformation is given in (Ward 1993). In (Ward 1994b) we deal with a program which implements a tree structure as a collection of four-way linked nodes, represented as pointers into arrays. The example in this paper also uses pointers, but in this case there are no "nodes": the arrays implement disjoint sets of elements and the pointer values are the actual elements.

Our example program exhibits a high degree of both control flow complexity and data representation complexity, using linked lists and circular lists implemented with pointers into arrays. In the process of our analysis we discover that one of the arrays is used for two different purposes: to record integer counts and to save and restore pointers. Our aim in this paper is to demonstrate that our program transformation theory, based on weakest preconditions and infinitary logic, can form the basis for a method for reverse engineering programs with complex data structures and control flow. The reverse engineering method is a heuristic method based on the selection and application of formal transformations and abstractions, 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.

1.1 The FermaT Project

The WSL language and transformation theory forms the basis of the FermaT project (Bull 1990; Ward, Calliss and Munro 1989) at Durham University and Software Migrations Ltd. which is developing 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 and target languages.

The initial prototype tool was developed as part of an Alvey project at the University of Durham. 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[1] and IBM UK Ltd. whose aim was to develop a tool to interactively transform assembly code into high-level language code and **Z** formal specifications (McMorran and Nicholls 1989). A prototype translator was built and tested on sample sections of up to 20,000 lines 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 transformation tool we were able to turn 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". (The programmer had selected the wrong Assembler instruction to implement a particular operation in the WSL program: this mistake was easily discovered and fixed). See (Ward and Bennett 1993; Ward and Bennett 1995a; Ward and Bennett 1995b) 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*, which we called $\mathcal{METAWSL}$. The extensions include an abstract data type for representing programs as tree structures and constructs for pattern matching and schema filling, together with powerful operators for iterating over the structure of a program: for example, to selectively modify all the terminal statements in a section of code. The "transformation engine" of FermaT is implemented entirely in $\mathcal{METAWSL}$. The implementation of $\mathcal{METAWSL}$ involves a translator from $\mathcal{METAWSL}$ to LISP, a small LISP runtime library (for the main abstract data types) and a WSL runtime library (for the high-level $\mathcal{METAWSL}$ constructs). 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 1994a), with an explicitly "middle-out" approach to the development (developing a very high level, domain-specific language) 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 $\mathcal{METAWSL}$ 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. The maintainer provides high-level "guidance" to the transformation process, while all the tedious condition checking and source code manipulation is carried out automatically. In the course of the development of the tool, 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 **goto**s and labels) can now be handled completely automatically through a single transformation. See (Ward 1994) for more details.

The strategy for reverse engineering using the FermaT tool is to start by restructuring the program, using general-purpose transformations and heuristics, and then attempting to massage the structure into a suitable form for the recursion introduction transformation: this "massaging" usually involves nothing more difficult than taking out the stack manipulation operations into separate actions (the "B-type" actions of Appendix A.3). Occasionally two or more actions will have to be combined into one, using a transformation dedicated to this process (see Section 6.2).

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: and 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 a valid abstraction of 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.

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 and Ward 1994; Ward 1990) for examples of program development in WSL using formal transformations.

[1] Centre for Software Maintenance Ltd: a Durham company, now called Software Migrations Ltd

More recently, FermaT has been applied to the task of migrating from assembler code (IBM 370 Assembler and a proprietary 16-bit assembler) to maintainable structured programs written in high level languages (including C and COBOL). Assembler modules of up to 39,000 lines have been processed, with FermaT automating much of the restructuring and data translation process. One of the more difficult tasks which FermaT has achieved is to move blocks of code into procedures: in the Assembler, a procedure call is implemented by storing a return address and jumping into the procedure, which returns by loading a register with the return address and branching to the register. FermaT has to cope with overwritten return addresses, multiple entry points into procedures, "calls" which do not store a return address, incrementing the return address before returning, and other "tricks" beloved by Assembler programmers!

FermaT has also been applied to the "year 2000 problem" in assembler: determining which instructions manipulate two-digit dates, and tracing data flows through assembler programs.

1.2 Overview of the Paper

In Section 2 we discuss related work in the area of program transformations. In Section 3 we introduce the concept of topological sorting and prove that the nodes in a directed graph can be topologically sorted if and only if the graph is cycle-free. Section 4 introduces the WSL language and transformation theory, (Appendix A gives some example program transformations which are used in the analysis). Knuth's algorithm is described in Section 5 and Section 6 shows that this algorithm can be transformed into an equivalent high-level abstract specification. Finally, in Section 7 we conclude with some remarks on the transformational approach to reverse engineering.

2 Related Work

The *Refinement Calculus* approach to program derivation (Hoare et al. 1987; Morgan 1994; Morgan, Robinson and Gardiner 1988; Morgan and Vickers 1993) 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. 1989; Bauer and The CIP Language Group 1985; Bauer and 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: 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, Gnatz and Wirsig 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.

The method adopted in the FermaT project: representing a transformation as an algorithm in a language designed for the purpose, is also used in the ZAP system (Feather 1987; Feather 1982). But with this system, the user needs to write (often large and complex) scripts in the meta language, HOPE (Burstall, McQueen and Sannella 1980), for each transformational development, making the system cumbersome to use. In (Hildum and Cohen 1990) a transformation is also specified as an algorithm which takes a given program as input and produces an equivalent program as output. Programs are stored as text sequences, necessitating a large time overhead in parsing. None of these meta-languages has the $\mathcal{METAWSL}$ facilities for pattern matching, schema filling, and iterating over the structure of a program.

The REDO project (Zuylen 1992), involving LLoyds Register and ten other partners including the University of Durham, ran from January 1989 to December 1991, and aimed to assist software engineers in the maintenance, restructuring and validation of large software systems and their transportation between different environments. The objective was to articulate a theoretical framework for doing this and develop methods and prototype tools. An intermediate language was developed, with business applications in mind, and a handbook on reverse engineering was produced.

3 Topological Sorting

In this section we will introduce the concept of topological sorting, and give a proof that the nodes of any cycle-free directed graph can be arranged into a straight line such that all the arrows point in the same direction.

One application of topological sorting is resolving dependencies: for example, the Unix "`make`" command which reads a "`Makefile`", listing the dependencies between source files, and executes the sequence of commands (compile, load etc.) required to bring the system up to date. Another application is displaying call graphs: the FermaT tool uses a modified algorithm which combines cycle detection with topological sorting to arrange the nodes on the page to minimise the number of upward pointing arrows. The graph in Figure 1 was generated by this algorithm.

Given a set P of pairs of elements taken from a base set B, the aim of topological sorting is to determine a permutation $\langle x_1, x_2, \ldots, x_n\rangle$ of B such that for each pair $\langle x_i, x_j\rangle \in P$ we have $i \leqslant j$. The problem is equivalent to arranging the vertices of a directed graph into a straight line in such a way that all the arrows point in the same direction. This will be possible if and only if there are no *cycles* in the directed graph, i.e. if there is no sequence $\langle c_1, c_2, \ldots, c_m\rangle$ of elements of B such that for each $1 \leqslant i < m$ the pair $\langle c_i, c_{i+1}\rangle$ is in P. The following theorem proves the existence of a "topological sort" (a *topsort*) whenever this condition holds:

Theorem 3.1 *Topological sorting of a finite set P of pairs of elements in a finite base set B is possible if and only if P is cycle free.*

Proof: The proof is a simple induction on the size of B. For the base case, if B is the empty set $\varnothing$, then P must also be empty, and the empty sequence $\langle\rangle$ suffices as a topsort. If B is not empty, then let x_1 be any *minimal element* of B, i.e. any element which does not appear as the first element of any pair in P. Such an element must exist, since otherwise there must be a cycle in P (start from any element of B and "work backwards" through pairs in P. Since the set is finite, we must eventually reach either a minimal element, or a cycle of elements). Then we remove all the pairs containing x_1 from P to form P', which will also be cycle-free, and remove x_1 from B to form B'. The set B' is smaller then B so, by the induction hypothesis, we can topologically sort P' to construct $\langle x_2, x_3, \ldots, x_n\rangle$. Prepending x_1 to this sequence then gives a topsort of B. ■

We have given the proof of this well-known result, because the proof that a solution exists also provides an algorithm for constructing the solution. In addition, we note that *any* topsort of P could be constructed by this algorithm. In other words, if $\langle x_1, x_2, \ldots, x_n\rangle$ is a topsort of P, then x_1 must be a minimal element (or there will be an arrow in the wrong direction) and $\langle x_2, \ldots, x_n\rangle$ must be a topsort of the P' constructed by the algorithm. A small modification of the algorithm can therefore be used to generate *all* topsorts of the given set P: for each minimal element x_1 of B, prepend it to each of the topsorts of the set $B \setminus \{x_1\}$ formed by removing all the pairs containing x_1 from P. The result will be a list of all the topsorts of P.

We will return to topological sorting in Section 5. First we discuss the wide spectrum language and give examples of some transformations.

4 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 $\mathbf{S}_1 \approx \mathbf{S}_2$ 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 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 1994; Morgan and Vickers 1993) and (Back 1980; Back 1988; Back and von Wright 1990) for a description of refinement. We write $\mathbf{S}_1 \leq \mathbf{S}_2$ if $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$, or if $\mathbf{S}_1$ is an abstraction of $\mathbf{S}_2$.

4.1 Syntax and Semantics

The syntax and semantics of WSL are described in (Priestley and Ward 1994; Ward 1989; Ward 1994c; 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 ith element a_i;
- $a[i\,..\,j]$ denotes the *subsequence* $\langle a_i, a_{i+1}, \ldots, a_j \rangle$;
- $\mathsf{butlast}(a)$ denotes the subsequence $a[1\,..\,\ell(a) - 1]$, that is, the sequence consisting of all but the last element in the sequence a;
- $\mathsf{set}(a)$ denotes the set of elements in the sequence, i.e. $\{a_1, a_2, \ldots, a_n\}$;
- The statement $x \xleftarrow{\text{pop}} a$ (where both x and a are variables, and a contains the sequence $\langle a_1, a_2, \ldots, a_n \rangle$) sets x to a_1 and a to $\langle a_2, a_3, \ldots, a_n \rangle$. It is equivalent to: $x := a[1]$; $a := a[2\,..]$;
- The statement $a \xleftarrow{\text{push}} e$ (where a contains a sequence as above, and e is any expression) sets a to $\langle x, a_1, a_2, \ldots, a_n \rangle$. It is equivalent to: $a := \langle e \rangle \mathbin{+\!\!+} a[1]$;
- The statement $x \xleftarrow{\text{last}} a$ (where both x and a are variables, and a contains a sequence) sets x to a_n and a to $\langle a_1, a_2, \ldots, a_{n-1} \rangle$. It is equivalent to: $x := a[\ell(a)]$; $a := a[1\,..\,\ell(a) - 1]$.

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

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. This means that statements in WSL can include existential and universal quantification over infinite sets, formulae with an infinite number of terms, and similar (non-executable) operations.

An example of a non-executable operation in WSL is the *specification statement*, written: $\langle x_1, \ldots, x_n \rangle := \langle x'_1, \ldots, x'_n \rangle.\mathbf{Q}$ which assigns new values to the variables $x_1, x_2, \ldots, x_n$. In the formula $\mathbf{Q}$, the variables 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, so for example the statement $\langle x \rangle := \langle x' \rangle.(x' = x + 1)$ will increment the value of x. If there are several sets of values which satisfy $\mathbf{Q}$ then one set is chosen nondeterministically. If there are no values which will satisfy $\mathbf{Q}$ then the statement does not terminate. For example, the assignment $\langle x \rangle := \langle x' \rangle.(x = 2 * x' \wedge x \in \mathbb{Z} \wedge x' \in \mathbb{Z})$ halves x if it is an even integer and aborts (does not terminate) if x is odd or not an integer. If the sequence contains one variable then the sequence brackets may be omitted, for example: $x := x'.(x = 2 * x')$. Another example is $x := x'.(y = 0)$ which 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* $\langle x_1, \ldots, x_n \rangle := \langle e_1, \ldots, e_n \rangle$, where the x_i are variables and the e_i are expressions, is an abbreviation for $\langle x_1, \ldots, x_n \rangle := \langle x'_1, \ldots, x'_n \rangle.(x'_1 = e_1 \wedge \cdots \wedge x'_n = e_n)$. It assigns the values of the expressions e_i to the variables x_i. The assignments are carried out simultaneously, so for example $\langle x, y \rangle := \langle y, x \rangle$ will swap the values of variables x and y. The single assignment $\langle x \rangle := \langle e \rangle$ can be abbreviated to $x := e$.

The *local variable statement*, written **var** $\langle x_1 := e_1, \ldots, x_n := e_n \rangle$: $\mathbf{S}$ **end** introduces a new local variables $x_1, \ldots, x_n$, initialised to the values of expressions $e_1, \ldots, e_n$ respectively. The local variable only exists while the statement $\mathbf{S}$, the body of the **var** statement, is executed. If x also exists as a *global* variable, then its value is saved and restored at the end of the block. (Technical note: the expressions e_i are allowed to refer the variables x_i, in which case it is the external *global* variable which is being referenced. So for example: **var** $\langle x := x \rangle$: $\mathbf{S}$ **end** will create a local copy of the global variable x, whose value will be restored at the end of the block, undoing any changes $\mathbf{S}$ has made to x).

An *assertion* in WSL is a statement of the form $\{\mathbf{P}\}$ where $\mathbf{P}$ is any formula. If the condition $\mathbf{P}$ is true, then the assertion acts as a **skip** statement (it terminated immediately without changing the value of any variable). Otherwise it acts as an **abort** (a non-terminating statement). Assertions are used for several purposes, for example: (1) A transformation which introduces an assertion at a certain point in the program has in effect proved that the formula will always be true at that point. (2) An assertion at the beginning of a program can be treated as a *precondition*: the transformations can assume that the condition is true initially. The assertion $\{\mathbf{P}\}$ is equivalent to a specification statement $\langle\rangle := \langle\rangle.\mathbf{P}$ with an empty list of variables. The statement **skip** is equivalent to {**true**}, and the statement **abort** is equivalent to {**false**}.

An *unbounded loop* is written **do S od** and can only be terminated by the execution of an exit statement **exit**(n) (where n is a non-negative integer, not a variable or expression), or an action call (see below). **exit**(n) causes immediate termination of the n nested enclosing **do** ... **od** loops, with execution continuing after the outermost loop. **exit**(1) may be abbreviated to **exit**, while **exit**(0) is equivalent to **skip**.

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 **call** X within the action body is a call of another action. There are some restrictions on the placing of action calls within an action system: action calls can only appear at the top level or within **if** statements and **do** ... **od** loops. The symbol $\equiv$ is also used for procedure definition (see below) and should be distinguished from the symbol for semantic equivalence: $\approx$.

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

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

actions A:
$A \equiv$
  if $\neg$**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.*

Normal procedures (with or without parameters) and functions are defined using a *where clause*:

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

Within the main body **S**, and the bodies $\mathbf{S}_i$, statements $F_i(x)$ are procedure calls with parameters. These can occur in any statement position since every procedure call always returns.

For a given set X, the *nondeterministic 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 $X' \neq \varnothing$ **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** $x \xleftarrow{\text{pop}} X$ **do S od**. The elements are taken in their order in the sequence, so in this case the iteration is deterministic. The loop is equivalent to:

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

5 Knuth's Topological Sorting Algorithm

The algorithm presented by Knuth and Szwarcfiter (Knuth and Szwarcfiter 1974) is written in a "pseudo PASCAL" notation. Although the title claims it is a *structured* program, it in fact contains various features which make the program very difficult to analyse and prove correct. These include the use of comments as the labels for **goto** statements, jumping into and out of the middle of loop structures, and the use of pointers into arrays to represent linked lists.

We have translated the algorithm into WSL, with some slight changes:

- The input is taken from an array R rather than by calling the read procedure;
- Instead of printing the result, we call process(s) for each topological sorting arrangement s. This generalises the algorithm without introducing any complications.

var $\langle \mathsf{count} := \langle\rangle, \mathsf{top} := \langle\rangle, \mathsf{link} := \langle\rangle,$
    $\mathsf{suc} := \langle\rangle, \mathsf{next} := \langle\rangle, s := \langle\rangle,$
    $d := 0, k := 0, t := 0, q := 0, n := 0,$
    $p := 0, j := 0, d_1 := 0 \rangle$:
for $j := 1$ **to** n **step** 1 **do**
  $\mathsf{count}[j] := 0$; $\mathsf{top}[j] := 0$ **od**;

for $k := 1$ **to** m **step** 1 **do**
  $\langle i, j\rangle := R[k]$; $\mathsf{suc}[k] := j$;
  $\mathsf{next}[k] := \mathsf{top}[i]$; $\mathsf{top}[i] := k$;
  $\mathsf{count}[j] := \mathsf{count}[j] + 1$ **od**;
$\mathsf{link}[0] := 0$; $d := 0$;
for $j := 1$ **to** n **step** 1 **do**
  if $\mathsf{count}[j] = 0$ **then** $\mathsf{link}[d] := j$; $d := j$ **fi od**;
actions start :
start $\equiv$
  if $d = 0$ **then call** done **else** $\mathsf{link}[d] := \mathsf{link}[0]$ **fi**;
  $k := 0$; $t := 0$; **call** alltopsorts.
alltopsorts $\equiv$
  if $k = n - 1$ **then** $s[n] := d$; $\mathsf{process}(s)$
    else $\mathsf{base}[k] := \mathsf{link}[d]$; **call** L **fi**;
  call endloop.
L $\equiv$
  $q := \mathsf{link}[d]$; $d_1 := \mathsf{link}[q]$;
  $p := \mathsf{top}[q]$;
  while $p \neq 0$ **do**
    $j := \mathsf{suc}[p]$;
    $\mathsf{count}[j] := \mathsf{count}[j] - 1$;
    if $\mathsf{count}[j] = 0$
      then if $d = q$ **then** $d := j$ **else** $\mathsf{link}[j] := d_1$ **fi**;
        $d_1 := j$ **fi**;
    $p := \mathsf{next}[p]$ **od**;
  $\mathsf{link}[d] := d_1$;
  $s[k+1] := q$;
  if $d_1 = q$ **then call** done **fi**;
  $\mathsf{count}[q] := t$; $t := q$; $k := k + 1$;
  call alltopsorts.
return $\equiv$
  $k := k - 1$;
  $q := t$; $t := \mathsf{count}[q]$; $\mathsf{count}[q] := 0$;
  $p := \mathsf{top}[q]$;
  while $p \neq 0$ **do**
    $j := \mathsf{suc}[p]$; $\mathsf{count}[j] := \mathsf{count}[j] + 1$;
    $p := \mathsf{next}[p]$ **od**;
  $\mathsf{link}[d] := q$; $d := q$;
  if $\mathsf{link}[d] \neq \mathsf{base}[k]$ **then call** L **fi**;
  call endloop.
endloop $\equiv$
  if $k > 0$ **then call** return **fi**; **call** done.
done $\equiv$
  call Z. **endactions end**

Figure 1 shows the call graph of the main action system.

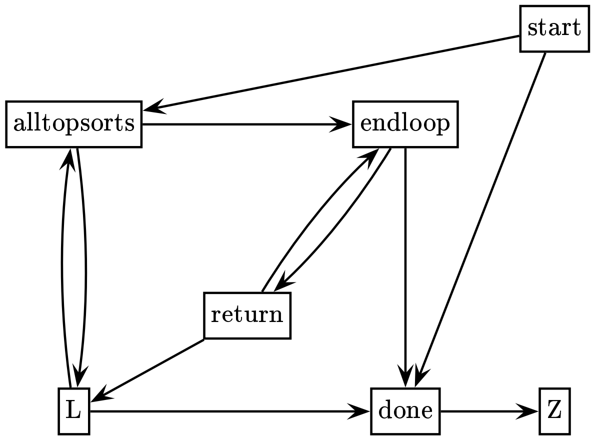

Figure 1: The Call Graph of Knuth's Topological Sorting Algorithm

6 Transformational Analysis

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 practicable to examine a small part of the program at a time.

The analysis of the algorithm breaks down into several stages:

1. Restructure to remove some of the control-flow complexity;
2. Recast as an iterative procedure in the right format for applying Corollary A.6;
3. Restructure the resulting recursive procedure;
4. Hypothesise a specification for the procedure, and remove the recursive call(s);
5. Add abstract variables to the program and update them in parallel with the actual (concrete) variables;
6. Replace references to concrete variables by equivalent references to abstract variables and remove the concrete variables to give an abstract program;

7. Show that the abstract program is a refinement of the expected specification.

6.1 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 the variables p and j are used locally in two sections of code, which we turn into procedures. Since we have yet to determine what these procedures do, they have been called P_1 and P_2. 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 FermaT tool this process is automated in the 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 heuristics include:

- Unfolding and removing an action which is only called once;
- Removing the tail recursion in an action which calls itself: by introducing a double-nested **do** ... **od** loop and replacing the self-calls by **exit**s. Further transformations are then attempted to reduce the double loop to a single loop;
- Merging calls—if an action calls another action in several places, then it is sometimes possible to combine these into a single call. For example: **if B then** $\mathbf{S}_1$; **call** A **fi**; $\mathbf{S}_2$; **call** A is equivalent to **if B then** $\mathbf{S}_1$; **call** A **else** $\mathbf{S}_2$; **call** A **fi** which in turn is equivalent to: **if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**; **call** A;
- Automatically converting a block of code into a procedure: for example, an action which is called in several places and which calls one other action (possibly several times), can be converted into a procedure call followed by an action call. Then all the calls to the action can be unfolded cheaply and the action removed from the system.

The overall aim of these heuristics is to reduce the number of actions in the system; the secondary aim is to reduce the total number of action calls in the system. These aims are to be achieved as far as possible without introducing new variables, or copying large blocks of code. See (Ward 1991; Ward 1993; Ward and Bennett 1993) for further details of these and other heuristics and examples of their application.

For Knuth's algorithm, the following sequence of transformations was discovered and applied automatically by the system:

1. Unfold the single call to return and remove that action from the system;
2. Remove the tail recursion thus created in the endloop action (this introduced what will become the inner loop in the program);
3. Unfold the remaining call to endloop and remove that action;
4. Now the two calls to L are brought together into the same action, and can be merged into a single call;
5. Unfold and remove the action L;
6. This creates a tail recursion in action alltopsorts which is removed to create the outer loop in the program;
7. Unfold and remove the action alltopsorts, since there is now a single call to it;
8. Now the three calls to done have been brought into the same action and can be merged;
9. Unfold and remove the action done.

The resulting action system contains a single action (start), so the action system is replaced by a **do** ... **od** loop with the calls to Z replaced by appropriate **exit** statements. Further transformations allow us to restructure the loop body and remove this extra loop.

The result of this totally automated process is as follows:

```
for j := 1 to n step 1 do count[j] := 0; top[j] := 0 od;
for k := 1 to m step 1 do
   ⟨i, j⟩ := R[k]; suc[k] := j; next[k] := top[i];
   top[i] := k; count[j] := count[j] + 1 od;
link[0] := 0; d := 0;
for j := 1 to n step 1 do
   if count[j] = 0 then link[d] := j; d := j fi od;
if d ≠ 0
  then link[d] := link[0];
       k := 0; t := 0;
       do if k = n − 1
            then s[n] := d; process(s);
                 do if k ⩽ 0 then exit(2) fi;
                    k := k − 1;
                    q := t; t := count[q];
                    count[q] := 0;
                    P2();
                    if link[d] ≠ base[k]
                      then exit(1) fi od
            else base[k] := link[d] fi;
          P1();
          s[k + 1] := q;
          if d1 = q then exit(1) fi;
          count[q] := t;
          t := q; k := k + 1 od fi
```

where the two procedures we have taken out are:

```
proc P1() ≡
  q := link[d]; d1 := link[q];
  var ⟨p := top[q], j := 0⟩ :
  while p ≠ 0 do
    j := suc[p];
    count[j] := count[j] − 1;
    if count[j] = 0
      then if d = q then d := j else link[j] := d1 fi;
           d1 := j fi;
    p := next[p] od;
  link[d] := d1 end.
```

and

```
proc P2() ≡
  var ⟨p := top[q], j := 0⟩:
  while p ≠ 0 do
    j := suc[p]; count[j] := count[j] + 1;
    p := next[p] od;
  link[d] := q;
  d := q end.
```

6.2 Prepare for Recursion Introduction

Our next aim is to restructure the program in the form of a recursive procedure, by applying Corollary A.6. 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 complex 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 tests for termination occur. First we introduce the procedure and create an action system for its body.

```
begin
  if d ≠ 0 then link[d] := link[0]; k := 0; t := 0; F() fi
where
proc F() ≡
  actions A1 :
  A1 ≡
    if k = n − 1
      then s[n] := d; process(s); call F̂1
       else base[k] := link[d]; call A2 fi.
  A2 ≡
    P1(); s[k + 1] := q;
    if d1 = q then call Z
              else call B1 fi.
  B1 ≡
    count[q] := t; t := q; k := k + 1; call A1.
  F̂1 ≡
    if k ⩽ 0
      then call Z
       else k := k − 1;
            q := t; t := count[q]; count[q] := 0;
            P2();
            if link[d] ≠ base[k] then call A2
                                 else call F̂1 fi fi.
  end
```

For the transformation of Corollary A.6 to be applicable there must be a single **call** Z statement in the $\hat{F}_1$ action, and none elsewhere. So we somehow need to eliminate the **call** Z in A_2. The "A-type" actions are supposed to call $\hat{F}_1$, which in turn will call Z provided $k \leqslant 0$. So our first attempt is to replace **call** Z by $k := 0$; **call** Z. Recall that since k is a *local* variable, this additional assignment does not change the semantics of the program as a whole. The procedure in Corollary A.6 only changes k by incrementing or decrementing it, but we can implement the assignment $k := 0$ as a **while** loop: **while** $k > 0$ **do** $k := k - 1$ **od**; **call** Z. This loop can be transformed into a tail-recursive action:

```
F̂2 ≡
  if k ⩽ 0 then call Z
           else k := k − 1; call F̂2 fi.
```

Action $\hat{F}_2$ is now quite similar to $\hat{F}_1$, and the two actions can be combined into one by adding a flag (a new local variable) and calling a "composite" action $\hat{F}$ which either acts like $\hat{F}_1$ or $\hat{F}_2$, depending on whether flag $= 1$ or flag $= 2$. The result of these manipulations is:

```
proc F() ≡
  actions A1 :
  A1 ≡
    if k = n − 1
      then s[n] := d; process(s); call F̂
       else base[k] := link[d]; call A2 fi.
  A2 ≡
    P1(); s[k + 1] := q;
    if d1 = q then flag := 2; call F̂
              else call B1 fi.
  B1 ≡
    count[q] := t; t := q; k := k + 1; call A1.
  F̂ ≡
    if k ⩽ 0
      then call Z
       else k := k − 1;
            if flag = 2
              then call F̂
               else q := t; t := count[q]; count[q] := 0;
                    P2();
                    if link[d] ≠ base[k]
                      then call A2 else call F̂1 fi fi.
```

One final point is that this program contains more references to the k variable than are required for Corollary A.6, although it does contain the right assignments to k in the right places. So we do not (at this stage) want to remove k from the program. These extra references are easily accommodated by adding a new local variable k' to the program which "shadows" the k variable (adding an identical assignment to k' after each assignment to k). The transformation in Corollary A.6 can then be allowed to eliminate k', leaving k behind. The result is the following recursive procedure:

```
proc F() ≡
  actions A1 :
  A1 ≡
    if k = n − 1
      then s[n] := d; process(s); call Z
       else base[k] := link[d]; call A2 fi.
  A2 ≡
    P1(); s[k + 1] := q;
    if d1 = q then flag := 2; call Z
              else call B1 fi.
  B1 ≡
    count[q] := t; t := q; k := k + 1; F(); k := k − 1;
```

if flag $= 2$
  then call Z
  else $q := t$; $t :=$ count$[q]$; count$[q] := 0$;
    $P_2()$;
    if link$[d] \neq$ base$[k]$ **then call** A_2
      else call Z **fi fi.**

It is often very important to cast a program into a recursive form, as early as possible in the analysis process. This is because recursive programs are generally much easier to analyse than their iterative equivalents.

6.3 Restructure the Recursive Procedure

For the next step in the analysis, we can remove the array base since it is only used to save the value of link$[d]$ across the inner recursive calls. Since base is a local variable, we can replace the array by a local variable within the procedure body. It is also clear that the array elements $s[k+1\,..\,n]$ are not used, so we will replace s by a sequence of length k. The variable k is then redundant, as its value is available as the length of s:

proc $F()\ \equiv$
 if $\ell(s) = n-1$
  then $s := s +\!\!+ \langle d\rangle$; process$(s)$; $s :=$ butlast(s)
  else var $\langle$base := link$[d]\rangle$:
   do $P_1()$;
    $s := s +\!\!+ \langle q\rangle$;
    if $d_1 = q$ **then** flag $:= 2$; **exit fi**;
    count$[q] := t$; $t := q$;
    $F()$;
    if flag $= 2$ **then exit fi**;
    $s :=$ butlast(s);
    $q := t$; $t :=$ count$[q]$; count$[q] := 0$;
    $P_2()$;
    if link$[d] =$ base **then exit fi od end fi.**

Up to this point in the reverse engineering process, we have not needed to know anything about the specification or purpose or method of implementation of the program. Only general-purpose transformations were required to get to a much more readable and understandable recursive version of the original iterative program—no new transformations needed to be developed in order to get to this stage.

In (Knuth and Szwarcfiter 1974) the authors claim that their "structured" algorithm was developed by a transformation process from a recursive algorithm, so it may seem hardly surprising that this transformation process can be reversed. However, there is some evidence to suggest that the $d_1 = q$ test was added *after* conversion to an iterative program: possibly to fix a "bug" which caused the program to loop indefinitely when the input file contained an oriented cycle. If the test *was* present in the recursive version, then some mechanism must have been used to "unwind" the nested recursive calls: either a flag (as in our version), or an exception mechanism such as LISPs "catch and throw" or C's "longjump". In addition, the code which implemented the exception mechanism must have been optimised away (which seems unlikely since no other optimisations have been performed).

Up to this point in the analysis, we have been working with precisely equivalent versions of the original program. The remarks in the previous paragraph suggest that the $d_1 = q$ test is checking for an error case, and this is bourne out by the program structure: if the test succeeds then the nested recursive calls will be unwound and the program terminated, cutting across the neat recursion structure. At this stage in the analysis however, we are only interested in the *normal* behaviour of the program, ignoring its behaviour for erroneous input data. We will therefore *abstract* the program (i.e. carry out the reverse of a refinement operation) by replacing the statement **if** $d_1 = q$ **then** flag $:= 2$; **exit fi** by a more abstract (less refined) statement: the assertion $\{d_1 \neq q\}$. For the normal case, both **if** statement and assertion are equivalent to **skip**, while for the error case, the assertion will abort, while the **if** statement will eventually lead to the program terminating, without any further assignments or calls to the process procedure. Once the normal case has been sorted out, it should be much easier to determine the condition on the input variable R which leads to the abnormal termination. The abstracted procedure looks like this (where the, now redundant, flag variable has been removed from the program):

proc $F()\ \equiv$
 if $\ell(s) = n-1$
  then process$(s +\!\!+ \langle d\rangle)$
  else var $\langle$base := link$[d]\rangle$:
   do $P_1()$; $\{d_1 \neq q\}$;
    count$[q] := t$; $t := q$;
    $s := s +\!\!+ \langle q\rangle$;
    $F()$;
    $s :=$ butlast(s);
    $q := t$; $t :=$ count$[q]$; count$[q] := 0$;
    $P_2()$;
    if link$[d] =$ base **then exit fi od end fi.**

6.4 Recursion Removal

With a recursive version of the program it is much easier to deduce various properties of the algorithm, directly from the structure of the recursive procedure. For example, it is clear that $F()$ preserves the sequence s (a value is appended, $F()$ is called, and the last value removed). This property cannot even be *stated*, let alone proved, for the original version of the algorithm. It also appears, from the structure of the program, that t is used to save and restore the value of q, while t is saved and restored in count$[q]$. If this is indeed the case, then we can avoid all the assignments to count and t, because the original value of q is already available after the call to $F()$: as the last element in the sequence s, which we know for certain is preserved by $F()$.

In order to eliminate the recursion and determine a "closed" form for the specification of the program we will need to apply Theorem A.1 in reverse. To do so we need a conjectured specification SPEC for the procedure, and a variant function. The variant function is easy: the sequence s is appended to just before the recursive call of $F()$ and its length is no greater than n (s grows from an empty sequence, but no recursive call is made when $\ell(s)$ reaches $n-1$). So the positive expression $n - \ell(s)$ is reduced before every recursive call. If we replace the recursive call by a copy of SPEC and show that the result is a valid refinement of SPEC, then Theorem A.1 tells us that the recursive procedure is also a valid refinement of SPEC. So the next three stages in the analysis are:

1. Conjecture which variables and array elements are preserved by the procedure;
2. Conjecture a possible specification SPEC as an abstraction of the procedure;
3. Replace the recursive call $F()$ by SPEC and show that the result is a refinement of SPEC.

Most of the variables and array elements assigned in $F()$ seem to be restored before the end of $F()$. The exceptions are the variable q and the link array: it is clear that some elements of link are overwritten in $P_1()$ and $P_2()$. In $P_1()$, we decrement a count value and, if it reaches zero, assign to the corresponding link value. So we conjecture that $F()$ preserves $\mathsf{link}[j]$ for all j such that $\mathsf{count}[j] = 0$.

We know that the program is supposed to be calling $\mathsf{process}(s)$ where s is a topsort of the elements $\{1, 2, \ldots, n\}$ according to the partial order $\mathsf{set}(R)$. Looking at the "base case" for $F()$ (the case where no recursive call is made), we see that when $\ell(s) = n - 1$ it simply calls $\mathsf{process}(s \mathbin{+\!\!+} \langle d\rangle)$. So s must be a partial topsort of $\{1, 2, \ldots, n\}$, and presumably $d \in \{1, 2, \ldots, n\} \setminus \mathsf{set}(s)$. So the recursive case must generate and process all the extensions of s which are topsorts. An extension $s \mathbin{+\!\!+} t$ of s is a topsort of $\{1, 2, \ldots, n\}$ if and only if t is itself a topsort of $\{1, 2, \ldots, n\} \setminus \mathsf{set}(s)$. Our specification for $F()$ may therefore be conjectured as:

$\mathsf{SPEC} =_{\text{DF}}$
for $t \in \mathsf{TOPSORTS}(\mathsf{set}(R), \{1, 2, \ldots, n\} \setminus \mathsf{set}(s))$ **do**
  $\mathsf{process}(s \mathbin{+\!\!+} t)$ **od**;
$q := q'.\textbf{true}$;
$\mathsf{link} := \mathsf{link}'.(\forall j, 1 \leqslant j \leqslant n.\, (\mathsf{count}[j] = 0)$
  $\Rightarrow (\mathsf{link}'[j] = \mathsf{link}[j]))$

where $\mathsf{TOPSORTS}(P, B)$ is the set of topsorts of the partial order P on base set B, i.e.

$\mathsf{TOPSORTS}(P, B) =_{\text{DF}}$
$\{\, t \in \mathsf{PERMS}(B) \mid \forall i, j.\, 1 \leqslant i < j \leqslant \#B.\langle t[j], t[i]\rangle \notin P \,\}$

where $\mathsf{PERMS}(B)$ is the set of all permutations of B, i.e. sequences of length $\#B$ containing all the elements of B:

$\mathsf{PERMS}(B) =_{\text{DF}}$
$\{\, t \in B^* \mid \ell(t) = \#B$
  $\wedge\ \forall x \in B.\, \exists i, 1 \leqslant i \leqslant \ell(t).\, x = t[i] \,\}$

Theorem 3.1 shows that if P is cycle-free, then

$$\mathsf{TOPSORTS}(P, B) \neq \varnothing$$

We replace the procedure call in the body of $F()$ by the specification SPEC to give a *non-recursive* procedure $F'()$:

proc $F'()\ \equiv$
 if $\ell(s) = n - 1$
  then $\mathsf{process}(s \mathbin{+\!\!+} \langle d\rangle)$
  else var $\langle \mathsf{base} := \mathsf{link}[d]\rangle$:
   do $P_1()$; $\{d_1 \neq q\}$;
    $\mathsf{count}[q] := t$; $t := q$;
    $s := s \mathbin{+\!\!+} \langle q\rangle$; SPEC; $s := \mathsf{butlast}(s)$;
    $q := t$; $t := \mathsf{count}[q]$; $\mathsf{count}[q] := 0$;
    $P_2()$;
    if $\mathsf{link}[d] = \mathsf{base}$ **then exit fi od end fi.**

If we can prove that $F'()$ is a refinement of SPEC then we can apply Theorem A.1 to prove that the recursive procedure $F()$ is also a refinement of SPEC.

Procedure $F'()$ can immediately be simplified somewhat, and the result makes some progress towards the proof: it is now clear that q is saved and restored from t, while t is saved and restored in $\mathsf{count}[q]$. In order for $\mathsf{count}[q]$ to be preserved over $F'()$, the assignment $\mathsf{count}[q] := 0$ must be restoring its original value (this is confirmed by Knuth's commentary on the algorithm which refers to using a "spare" count value to save the value of t). So we *abstract* the procedure by inserting the assertion $\mathsf{count}[q] = 0$ just after $P_1()$, in the hope that this abstraction will not prevent us from proving $\mathsf{SPEC} \leq F'()$. With this addition, we can delete the assignments to t, q and count, since the value of q is available as the last element of s. The variable t can be removed altogether (recall that t is a local variable in the whole program, so we do not need to preserve its final value):

proc $F'()\ \equiv$
 if $\ell(s) = n - 1$
  then $\mathsf{process}(s \mathbin{+\!\!+} \langle d\rangle)$
  else var $\langle \mathsf{base} := \mathsf{link}[d]\rangle$:
   do $P_1()$; $\{d_1 \neq q\}$;
    $\{\mathsf{count}[q] = 0\}$;
    $s := s \mathbin{+\!\!+} \langle q\rangle$; SPEC; $q \xleftarrow{\text{last}} s$;
    $P_2()$;
    if $\mathsf{link}[d] = \mathsf{base}$ **then exit**(1) **fi od fi.**

6.5 Changing the Data Representation—Adding Abstract Variables

The program makes much use of linked lists to represent sets of integers. Our aim in this section is to replace these concrete data structures by the equivalent abstract data structures (i.e. actual sequences and sets instead of linked lists). The program represents a small set of (positive) integers by using a variable, say v, and an array, say a. It represents an empty set by $v = 0$ and a singleton set $\{x_1\}$ by $v = x_1$ and $a[x_1] = 0$. In general, v holds the first element of the set, while for the last element $a[x_n] = 0$, and for the other elements $a[x_i] = x_{i+1}$ (for $1 \leqslant i < n$). Thus the set is represented as an unordered list without duplicates. We define an *abstraction function*, $\mathsf{list}(v, a, w)$ which returns the list represented by v and a, where w is the special value (in this case, zero) which indicates the end of the list:

$$\mathsf{list}(v, a, w) \;=_{\text{DF}}\; \begin{cases} \langle\rangle, & \text{if } v = w \\ \langle v\rangle \mathbin{+\!\!+} \mathsf{list}(a[v], a, w) & \text{if } v \neq w \end{cases}$$

For a (possibly empty) set of positive integers, represented as a linked list terminated by 0, the list is $\mathsf{list}(v, a, 0)$ which gives $\langle v, a[v], a[a[v]], \ldots\rangle$. If v is non-zero, then the effect of the assignment $v := a[v]$ is to remove the first element from the list, while the assignments $a[x] := v$; $v := x$ add the element x to the front of the list (assuming that x is not already present in the list).

If we know that the set will be non-empty, then we can avoid the need for a terminating value by creating a circular-linked list. We do this by setting $a[x_n]$ to v, where x_n is the last element in the list: so, for example, a one element list will have $a[v] = v$. The list represented by this "circular linking" is $\mathsf{list}(a[v], a, v) \mathbin{+\!\!+} \langle v\rangle$. Note that here, the value in v is the *last* element in the list. In this representation, the assignment $v := a[v]$ has the effect of rotating the list by one step, rather than removing the first element.

If a zero-terminated list is used to store a set S in decreasing order, then the value of v and of each array element whose index is in S, is completely determined by S:

$$v = \begin{cases} 0 & \text{if } S = \varnothing \\ \mathsf{max}(S) & \text{otherwise} \end{cases}$$

and for each $x \in S$:

$$a[x] = \begin{cases} 0 & \text{if } x = \mathsf{min}(S) \\ \mathsf{max}\,\{\, y \in S \mid y < x \,\} & \text{otherwise} \end{cases}$$

Conversely, $\mathsf{list}(v, a, 0) = \mathsf{sort}(S)$, where sort takes a set of integers and returns the corresponding sorted sequence (in decreasing order).

A single array can be used to represent several disjoint sets of integers: since each integer is in at most one set, it can be associated with the unique "next" element in its set (The "next" element is zero for the last element in the set).

If variable v and array a record the set S in zero-terminated form, then the nondeterministic iteration **for** $p \in S$ **do S od** can be implemented by these statements:

var $\langle p := v\rangle$:
while $p \neq 0$ **do**
  S;
  $p := a[p]$ **od end**

(This is a deterministic refinement of the nondeterministic iteration). An equivalent program using an iteration over a sequence is:

for $p \xleftarrow{\text{pop}} \mathsf{list}(v, a, 0)$ **do**
  S od

If the list records the set S in decreasing order, then another equivalent program is:

for $p \xleftarrow{\text{pop}} \mathsf{sort}(S)$ **do**
  S od

The first **for** loop in Knuth and Szwarcfiter's program simply initialises the arrays count and top to zeros. The second **for** loop is:

for $k := 1$ **to** m **step** 1 **do**
  $\langle i, j\rangle := R[k]$; $\mathsf{suc}[k] := j$;
  $\mathsf{next}[k] := \mathsf{top}[i]$; $\mathsf{top}[i] := k$;
  $\mathsf{count}[j] := \mathsf{count}[j] + 1$ **od**;

This reads each pair in the R list and updates count, top and suc. We claim that count records for each element j, how many pairs in R have j as the *second* component. The arrays top and suc partition the set $\{1, 2, \ldots, k\}$ of indices of elements of R into a collection of *disjoint* subsets, one per element, where each subset contains all the indices of pairs with the same *first* component. We add an abstract variable A which records the set of "active" indices of R (it records which pairs of R are currently included in the representations), and this gives a suitable invariant for the second loop:

$$\left.\begin{array}{l} \forall j, 1 \leqslant j \leqslant n.\, \mathsf{count}[j] \\ \qquad = \#\,\{\, x \in A \mid R[x][2] = j \,\} \\ \forall i, 1 \leqslant i \leqslant n.\, \mathsf{list}(\mathsf{top}[i], \mathsf{next}, 0) \\ \qquad = \mathsf{sort}\,\{\, x \in A \mid R[x][1] = i \,\} \\ \forall x \in A.\, \mathsf{suc}[x] = R[x][2] \end{array}\right\} \qquad (I_1)$$

Thus $\mathsf{count}[j]$ is the number of pairs read so far which have j as the second element (this is not necessarily the number of predecessors of j in the input relation, since the same pair may appear more than once in the input). For each i:

$$\mathsf{top}[i] = \begin{cases} 0 & \text{if } \neg\exists k \in A.\, R[k][1] = i \\ \mathsf{max}\,\{\, k \in A \mid R[k][1] = i \,\} & \text{otherwise} \end{cases}$$

and for any $k \in A$, let $i = R[k][1]$ and then:

$$\mathsf{next}[k] = \begin{cases} 0 & \text{if } \neg\exists k' \in A.\, (k' > k \,\wedge\, R[k'][1] = i) \\ \mathsf{max}\,\{\, k' \in A \mid k' > k \,\wedge\, R[k'][1] = i \,\} \end{cases}$$

After the second **for** loop, the statements $\mathsf{link}[0] := 0$; $d := 0$ set up an empty, zero-terminated list in variable d and array link. The third **for** loop iterates over the set of elements which have zero count (i.e. they do not appear as the second component of any pair), and puts these elements into the list formed by d and the link array. We add another abstract variable B (the "active base") which records which elements have been processed so far. All assignments to B and s maintain the invariant $B = \{1, 2, \ldots, n\} \setminus \mathsf{set}(s)$. Define:

$\mathsf{MINS}(R, A, B) \;=_{\text{DF}}$
  $\{\, x \in b \mid \neg\exists k \in A.\, R[k][2] = x \,\}$

These are the minimal elements in the active base set. From I_1 we have

$$\mathsf{MINS}(R, A, B) = \{\, x \in B \mid \mathsf{count}[x] = 0 \,\}$$

We add another abstract variable D to record the value of the list $\mathsf{list}(\mathsf{link}[0], \mathsf{link}, d) +\!\!+ \langle d\rangle$. So we have this invariant for the third loop:

$$\mathsf{MINS}(R, A, B) = \mathsf{set}(D) \qquad (I_2)$$

We also have an invariant which links d, link and D:

$$D = \begin{cases} \langle\rangle & \text{if } d = 0 \\ \mathsf{list}(\mathsf{link}[0], \mathsf{link}, d) +\!\!+ \langle d\rangle & \text{if } d \neq 0 \end{cases} \qquad (I_3)$$

The statement $\mathsf{link}[d] := \mathsf{link}[0]$ turns the list into a circular list with $\mathsf{link}[d]$ as the first element (provided $d \neq 0$). We can ignore the case $d = 0$ since by I_2 and I_3 this implies $\mathsf{MINS}(R, A, B) = \varnothing$ which is impossible (for $B \neq \varnothing$) if there are no cycles in R. So after the assingment $\mathsf{link}[d] := \mathsf{link}[0]$, a slightly modified I_3 is true:

$$D = \mathsf{list}(\mathsf{link}[d], \mathsf{link}, d) +\!\!+ \langle d\rangle \qquad (I_3')$$

Finally, a simple relationship between A and B will be maintained, apart from when it is temporarily disrupted within procedures P_1 and P_2:

$$A = \{\, k \in 1\,..\,m \mid R[k][1] \in B \,\} \qquad (I_4)$$

the active indices are simply the indices of all pairs whose first element is in the active base set.

The rest of the annotated initialisation code, with assertions and abstract variables added, looks like this:

$\mathsf{link}[0] := 0;\ d := 0;\ B := \varnothing;\ D := \langle\rangle;$
$\{I_1 \wedge I_2 \wedge I_3\};$
for $j := 1$ **to** n **step** 1 **do**
  $B := B \cup \{j\};$
  if $\mathsf{count}[j] = 0$
    then $\mathsf{link}[d] := j;\ d := j;\ D := D +\!\!+ \langle j\rangle$ **fi**;
  $\{I_1 \wedge I_2 \wedge I_3\}$ **od**;
$\{d \neq 0\};\ \mathsf{link}[d] := \mathsf{link}[0];$
$\{I_1 \wedge I_2 \wedge I_3'\};$
$k := 0;\ t := 0;$

Note that after the third **for** loop, if $d = 0$ then $\mathsf{MINS}(R, A, B) = \varnothing$ which is impossible unless the input sequence R contains a cycle, or $B = \varnothing$. The former is an error case, and the latter is the trivial case $n = 0$, so we may safely ignore these cases and assume $d \neq 0$.

Having established the abstract variables and the invariants relating these to the concrete variables, the next step is to add assignments to the abstract variables to ensure that the invariants are preserved throughout the program. To do this it is sufficient to examine each block of code in turn, without needing to understand the program as a whole. This is one reason why our approach is able to "scale up" to much larger programs: only at a later stage, when we have a more abstract version of the program, do we need to consider the program as a whole. At this point it is sufficient to work on each small "chunk" at a time.

First, we consider the body of procedure P_1:

$q := \mathsf{link}[d];\ d_1 := \mathsf{link}[q];$
var $\langle p := \mathsf{top}[q], j := 0\rangle$:
while $p \neq 0$ **do**
  $j := \mathsf{suc}[p];$
  $\mathsf{count}[j] := \mathsf{count}[j] - 1;$
  if $\mathsf{count}[j] = 0$
    then if $d = q$ **then** $d := j$ **else** $\mathsf{link}[j] := d_1$ **fi**;
      $d_1 := j$ **fi**;
  $p := \mathsf{next}[p]$ **od**;
$\mathsf{link}[d] := d_1$ **end**

The **while** loop iterates over the list represented by $\mathsf{top}[q]$ and next, which we know from I_1 are the elements in the set $\{\, x \in \{1, 2, \ldots, m\} \mid R[x][2] = q \,\}$. It decrements $\mathsf{count}[j]$ for each element j in the list. In order to maintain I_1 we must therefore remove p (the index of the pair $\langle q, j\rangle$) from the active indices set A. If the count becomes zero, then in order to maintain I_2 we must eventually add j to the D list. Having removed all the indices of pairs based on q from A, we must remove q from B to restore I_4. We will therefore also have to remove q from D in order to maintain I_2.

This is the motivation for adding the following abstract variables to the program:

- D_1 records $\mathsf{list}(d_1, \mathsf{link}, d) +\!\!+ \langle d\rangle$;
- D' records the list required to maintain the invariant $\mathsf{MINS}(R, A, B) = \mathsf{set}(D') \cup \{q\}$ as elements are removed from A. Once q is removed from B, D' will be the new value of D required to restore I_2;
- D_0 records the initial value of D;
- d_0 records the initial value of d.

Note that the initialisation of d_1 means that the circular list $\mathsf{list}(d_1, \mathsf{link}, d) +\!\!+ \langle d\rangle$ is either $D[2\,..]$ (if D has two or more elements), or D (if D has only one element).

$D_0 := D;\ d_0 := d;$
var $\langle j := 0, D' := D[2\,..], D_1 := \langle\rangle\rangle$:
$\{I_1 \wedge I_2 \wedge I_3' \wedge I_4\};$
$q := \mathsf{link}[d];\ \{q = D[1] \wedge d = D[\ell(D)]\};$
$\{\mathsf{list}(q, \mathsf{link}, d_0) +\!\!+ \langle d_0\rangle = D_0\};$
$d_1 := \mathsf{link}[q];$
if $D = \langle q\rangle$ **then** $D_1 := D$ **else** $D_1 := D[2\,..]$ **fi**;
$\{D_1 = \mathsf{list}(d_1, \mathsf{link}, d) +\!\!+ \langle d\rangle\};$
$\{D = \langle q\rangle = D_1 \wedge D' = \langle\rangle \vee \ell(D) > 1 \wedge D' = D_1$
  $\wedge\ \mathsf{MINS}(R, A, B) = \mathsf{set}(D') \cup \{q\}\};$
for $p \xleftarrow{\text{pop}} \mathsf{sort}\,\{\, k \in A \mid R[k][1] = q \,\}$ **do**
  $j := R[p][2];$
  $A := A \setminus \{p\};$
  $\mathsf{count}[j] := \mathsf{count}[j] - 1;$
  $\{I_1\};$
  if $\mathsf{count}[j] = 0$
    then if $d = q$
        then $\{D = \langle q\rangle \wedge D' = \langle\rangle\};$
          $d := j;\ d_1 := j;$
          $\{d \in A \wedge R[d][1] = q\};$
          $D' \xleftarrow{\text{push}} j;\ D_1 := \langle j\rangle;$
          $\{D_1 = \mathsf{list}(d_1, \mathsf{link}, d) +\!\!+ \langle d\rangle$
            $\wedge\ D' = D_1\}$
      else $\{D \neq \langle q\rangle \vee D' \neq \langle\rangle\};$

$\mathsf{link}[j] := d_1;\ d_1 := j;$
$\{d = d_0\}$
$D' \xleftarrow{\text{push}} j;\ D_1 \xleftarrow{\text{push}} j;$
$\{D_1 = \mathsf{list}(d_1, \mathsf{link}, d) +\!\!+ \langle d \rangle\}$ **fi fi**;
$\{D_1 = \mathsf{list}(d_1, \mathsf{link}, d) +\!\!+ \langle d \rangle$
$\wedge\ \mathsf{MINS}(R, A, B) = \mathsf{set}(D') \cup \{q\}\}$ **od**;
$B := B \setminus \{q\};\ \{I_1\ \wedge\ I'_3\ \wedge\ I_4\};$
$\mathsf{link}[d] := d_1;$
$\{\mathsf{MINS}(R, A, B) = \mathsf{set}(D')\};$
$D := D_1$ **end**

The final assignment $\mathsf{link}[d] := d_1$ ensures that the list $\mathsf{list}(\mathsf{link}[d], \mathsf{link}, d)\ +\!\!+\ \langle d \rangle$ is equal to D_1. To ensure that I_2 is restored by the assignment $D := D_1$ it is sufficient to show that $D_1 = D'$ at this point. However, the assertion $(D = \langle q \rangle = D_1\ \wedge\ D' = \langle\rangle)\ \vee\ (\ell(D) > 1\ \wedge\ D' = D_1)$ is preserved over the loop, so if $D_1 \neq D'$ we must have $D' = \langle\rangle$ which in turn implies $\mathsf{MINS}(R, A, B) = \varnothing$. This is impossible for a non-empty B since R is cycle-free, and we know that B is non-empty because $B = \{1, 2, \ldots, n\} \setminus \mathsf{set}(s)$ and $\ell(s) \leqslant n - 1$.

Finally, note that d is unchanged, (i.e. $d = d_0$), unless D was originally a singleton $\langle q \rangle$. In the latter case, the final value of d is some element of $\{\, k \in A \mid R[k][1] = q \,\}$. (In fact, it will be the largest element j such that $\mathsf{count}[j] = 1$.) So we have an additional assertion at the end of $P_1()$:

$$\begin{aligned} &\ell(D_0) > 1\ \wedge\ d = d_0 \\ &\quad \vee\ (\ell(D_0) = 1\ \wedge\ d \in A \\ &\qquad \wedge\ R[d][1] = q\ \wedge\ D_0 = \langle q \rangle) \end{aligned} \qquad (I_5)$$

Procedure P_2 is more straightforward. It also iterates over a set, in fact the same set of elements as P_1, namely $\{\, x \in \{1, 2, \ldots, m\} \mid R[x][2] = q \,\}$, but incrementing rather than decrementing the counts.

var $\langle p := \mathsf{top}[q], j := 0 \rangle:$
while $p \neq 0$ **do**
$j := \mathsf{suc}[p];\ \mathsf{count}[j] := \mathsf{count}[j] + 1;$
$p := \mathsf{next}[p]$ **od**;
$\mathsf{link}[d] := q;$
$d := q$ **end**

From the assertion on q near the beginning of $P_1()$ we know that $\mathsf{list}(q, \mathsf{link}, d_0)\ +\!\!+\ \langle d_0 \rangle\ =\ D_0$. After the assignment $\mathsf{link}[d]\ :=\ q$ we will therefore have $\mathsf{list}(\mathsf{link}[d], \mathsf{link}, d_0)\ +\!\!+\ \langle d_0 \rangle = D_0$. From the assertion I_5, there are two cases depending on the initial value D_0:

1. $\ell(D_0)\ >\ 1$ in which case $d\ =\ d_0$, so $D_0\ =\ \mathsf{list}(q, \mathsf{link}, d)\ +\!\!+\ \langle d \rangle$ and after $\mathsf{link}[d] = q$ therefore, we have $D_0 = \mathsf{list}(\mathsf{link}[d], \mathsf{link}, d)\ +\!\!+\ \langle d \rangle$. So I'_3 can be restored by the assignment $D := D_0$. Then the assignment $d := q$ rotates the circular list starting at $\mathsf{link}[d]$ by one place, and I'_3 can again be restored by the assignment $D := D[2\,..] +\!\!+ \langle D[1] \rangle$;
2. $\ell(D_0)\ =\ 1$. In this case $D_0\ =\ \langle q \rangle$ and $\mathsf{link}[d_0]\ =\ d_0 = q$ and $d \in A\ \wedge\ R[d][1] = q$. The assignment $\mathsf{link}[d]\ :=\ q$ merely clobbers a link value for which count is non-zero (and this is allowed by the proposed specification of $F'()$. Then the assignment $d := q$ restores $\mathsf{list}(\mathsf{link}[d], \mathsf{list}, d)\ +\!\!+\ \langle d \rangle\ =\ \langle q \rangle\ =\ D_0$. So I_3 can be restored by the assignment $D\ :=\ D_0$. However, since in this case D is a singleton sequence, the assignment $D := D[2\,..]\ +\!\!+\ \langle D[1] \rangle$ will have no effect and can be added if required.

Therefore, in either case, I_3 is restored by the assignment $D := D_0[2\,..]\ +\!\!+\ \langle D_0[1] \rangle$. So we can preserve invariants I_1 to I_4 with the following assignments to abstract variables within procedure P_2:

var $\langle j := 0 \rangle:$
for $p \xleftarrow{\text{pop}} \mathsf{sort}\,\{\, k \in A \mid R[k][1] = q \,\}$ **do**
$j := R[p][2];$
$A := A \cup \{p\};$
$\mathsf{count}[j] := \mathsf{count}[j] + 1;$
$\{I_1\}$ **od**;
$B := B \cup \{q\};$
$\mathsf{link}[d] := q;$
$d := q;$
$D := D_0[2\,..] +\!\!+ \langle D_0[1] \rangle$ **end**

6.6 Changing the Data Representation—Removing Concrete Variables

We have now "built the scaffolding" around the various parts of the program: this consists of adding abstract variables with assignments to them, and invariants relating the abstract and concrete variables. We were able to do this for each section of the program independently of the others, without needing to determine the "big picture". We are now in a position to put all the pieces together to form a "hybrid" program. Having done so, we can make use of the assertions to replace references to concrete variables by equivalent references to abstract variables. For example the test $\mathsf{count}[j] = 0$ is replaced by the equivalent test $\neg\exists x \in A.\, R[x][2] = j$ by appealing to invariant I_1. References to concrete variables appearing in assignments to other concrete variables do not need to be removed, for example the statement $\mathsf{count}[j] := \mathsf{count}[j] + 1$ can remain. Once all relevant references to concrete variables have been removed, these become "ghost variables", since they have no effect on the execution of the program, and the concrete variables can be removed in their entirety. This "ghost variables" technique has been used for program development in (Broy and Pepper 1982; Jorring and Scherlis 1987; Wile 1981). The result is an abstract procedure equivalent to $F'()$:

proc $F''()\ \equiv$
if $\ell(s) = n - 1$
then $\mathsf{process}(s +\!\!+ \langle d \rangle)$
else var $\langle \mathsf{base} := D[1] \rangle:$
do $\{\mathsf{MINS}(R, A, B) = \mathsf{set}(D)\};$
var $\langle D_0 := D, D' := D[2\,..] \rangle:$
$q := D[1];$
for $p \xleftarrow{\text{pop}} \mathsf{sort}\,\{\, k \in A \mid R[k][1] = q \,\}$ **do**
$j := R[p][2];$
$A := A \setminus \{p\};$

if $\{ x \in A \mid R[x][2] = j \} = \varnothing$
then $D' \xleftarrow{\text{push}} j$ **fi od**;
$B := B \setminus \{q\}$;
$\{\mathsf{MINS}(R, A, B) = \mathsf{set}(D')\}$;
$D := D'$ **end**;
$\{\neg \exists x \in A.\, R[x][2] = q\}$;
$\{D[\ell(d)] \neq q\}$;
$s := s \mathbin{+\!\!+} \langle q \rangle$;
SPEC;
$q \xleftarrow{\text{last}} s$;
var $\langle j := 0 \rangle$:
for $p \xleftarrow{\text{pop}} \mathsf{sort}\{ k \in A \mid R[k][1] = q \}$ **do**
$A := A \cup \{p\}$;
$j := R[p][2]$;
$\{I_1\}$ **od**;
$B := B \cup \{q\}$;
$D := D_0[2\,..] \mathbin{+\!\!+} \langle D_0[1] \rangle$ **end**
if $D[1] = \mathsf{base}$ **then exit**(1) **fi od fi.**

Recall that SPEC is our hypothesised specification for the whole program. Our aim is to prove that $F''()$ itself is a refinement of SPEC, which by appealing to Theorem A.1 proves that the recursive procedure $F()$ is also a refinement of SPEC.

We can now remove the two assertions $\{\neg \exists x \in A.\, R[x][2] = q\}$ (which was originally $\{\mathsf{count}[q] = 0\}$) and $\{D[\ell(d)] \neq q\}$ (which was originally $\{d_1 \neq q\}$). In the first case, $q = D[1]$ and D is a list of all minimal elements in the active set. So no active pair (pair whose index is in A) can have q as a second element. In the second case, we have removed q from D and possibly added some new elements, so q cannot still be in D.

6.7 Verifying the Abstract Program

The final step is to prove that our abstract program (containing a copy of the hypothesised specification SPEC) is a valid refinement of SPEC. To ease this step we can make use of the assertions to simplify the abstract program still further, in particular by removing the two **for** loops. This is because the assertions tell us the final values of the variables modified by the loops. For example, using I_4 we can remove A completely. Define:

$\mathsf{MINS}(R, B) \ =_{\text{DF}}$
$\mathsf{MINS}(R, \{ k \in 1\,..\,m \mid R[k][1] \in B \}, B)$

Also, if $\ell(s) = n - 1$ then B contains the single element d, so the only topsort of B is $\langle d \rangle$. So $\mathsf{process}(s \mathbin{+\!\!+} \langle d \rangle) \approx$ SPEC in this case.

proc $F''() \ \equiv$
if $\ell(s) = n - 1$
then SPEC
else $D := \mathsf{sort}(\mathsf{MINS}(R, B))$;
var $\langle \mathsf{base} := D[1] \rangle$:
do var $\langle D_0 := D \rangle$:
$q := D[1]$;
$B := B \setminus \{q\}$;
$D := \mathsf{sort}(\mathsf{MINS}(R, B))$;
$s := s \mathbin{+\!\!+} \langle q \rangle$; SPEC; $q \xleftarrow{\text{last}} s$;
$D := D_0[2\,..] \mathbin{+\!\!+} \langle D_0[1] \rangle$
$B := B \cup \{q\}$ **end**
if $D[1] = \mathsf{base}$ **then exit**(1) **fi od fi.**

Now D and B do not appear in SPEC, so there is no need to modify and then restore them:

proc $F''() \ \equiv$
if $\ell(s) = n - 1$
then SPEC
else $D := \mathsf{sort}(\mathsf{MINS}(R, B))$;
var $\langle \mathsf{base} := D[1] \rangle$:
do $q := D[1]$;
$s := s \mathbin{+\!\!+} \langle q \rangle$; SPEC; $q \xleftarrow{\text{last}} s$;
$D := D[2\,..] \mathbin{+\!\!+} \langle D[1] \rangle$ **end**
if $D[1] = \mathsf{base}$ **then exit**(1) **fi od fi.**

Now it is clear that the loop simply iterates over the elements of $\mathsf{MINS}(R, B)$ in ascending order. We can abstract this loop to a nondeterministic iteration which iterates over $\mathsf{MINS}(R, B)$ in an arbitrary order:

proc $F''() \ \equiv$
if $\ell(s) = n - 1$
then SPEC
else for $q \in \mathsf{MINS}(R, B))$ **do**
$s := s \mathbin{+\!\!+} \langle q \rangle$; SPEC; $q \xleftarrow{\text{last}} s$ **od fi.**

The proof of Theorem 3.1 shows that when B contains two or more elements (or equivalently, when $\ell(s) < n - 1$, SPEC is refined by:

for $q \in \mathsf{MINS}(P, B)$ **do**
for $t \in \mathsf{TOPSORTS}(\mathsf{set}(R), \{1, 2, \dots, n\} \setminus \mathsf{set}(s))$ **do**
$\mathsf{process}(s \mathbin{+\!\!+} t)$ **od**;
$q := q'.\mathbf{true}$;
$\mathsf{link} := \mathsf{link}'.(\forall j, 1 \leqslant j \leqslant n.\, (\mathsf{count}[j] = 0)$
$\Rightarrow (\mathsf{link}'[j] = \mathsf{link}[j]))$ **od**

Hence $F'()$ is a refinement of SPEC, and this was what we needed at the end of Section 6.4 to show that $F()$ is a refinement of SPEC.

The program as a whole is therefore a refinement of the abstract program:

$B := \{1, 2, \dots, n\}$; $s := \langle \rangle$;
for $t \in \mathsf{TOPSORTS}(\mathsf{set}(R), \{1, 2, \dots, n\} \setminus \mathsf{set}(s))$ **do**
$\mathsf{process}(s \mathbin{+\!\!+} t)$ **od**

which simplifies to:

for $t \in \mathsf{TOPSORTS}(\mathsf{set}(R), \{1, 2, \dots, n\})$ **do**
$\mathsf{process}(t)$ **od**

We have shown that Knuth's topological sorting algorithm is a refinement of the above specification, and hence that it does indeed generate all the topological sorts of the given relation.

7 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 approach to some challenging program analysis and reverse engineering problems.

In (Ward 1993) 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 together with complicated data structures which are 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 original 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 advanced knowledge of the program's behaviour before they can be applied. This is essential in a reverse engineering application where it is not clear at the onset of the process precisely how the source code of the program implements the (formal or informal) specification—or indeed, whether the specification has been implemented correctly!

For the introduction of abstract variables, preparatory to the derivation of an abstract program, it is helpful for the programmer to have some knowledge of the implementation of data structures in the program. Of course, this knowledge was not required for the restructuring and transformation to a recursive program, and it is much easier to glean useful information such as this from the transformed (recursive) program, than from the original spaghetti code. The abstract variables are introduced by making use of only local knowledge of the program's behaviour—a deep understanding of the algorithm is not required at this stage, only a local knowledge of the data manipulations. When the abstract program has been derived and restructured and simplified, then it can be analysed to give a deeper knowledge of the algorithm as a whole. In this way, the reverse engineering and program analysis tasks proceed step-by-step with progress in analysis leading to progress in reverse engineering, which in turn makes a more detailed analysis achievable ... and so on.

In our case study, we were able to analyse this particularly difficult program, which combines a complex control flow with complex linked data structures and tight coding. An example of "tight coding" is the pair of assignments $\mathsf{link}[d] := q;\ d := q$ which have two quite different effects, depending on whether d points to a singleton list or a longer list. Fortunately, the two different effects just happen to produce the right results in each case, so the programmers have cleverly avoided the need to test for a singleton list[2].

Appendix

A Example Transformations

In this appendix we describe some of the transformations used in the transformational analysis of Knuth and Szwarcfiter's algorithm.

A.1 Introducing Recursion

The first transformation shows how a general statement, which may be a non-recursive specification statement, or a recursive or iterative statement, can be refined into a recursive procedure. Applications of this important result include implementing specifications as recursive procedures, introducing recursion into an abstract program to get a "more concrete" program (i.e. closer to a programming language implementation), and transforming a recursive procedure into a different form. The transformation is also used in the algorithm derivations of (Ward 1989; Ward 1990) and (Priestley and Ward 1994).

Suppose we have a statement $\mathbf{S}'$ which we wish to transform into the recursive procedure **proc** $F \equiv \mathbf{S}$. This is possible whenever:

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

Note that the order $<$ is *not* well-founded on $\mathbb{Z}$, but it is well-founded on $\mathbb{N}$. The expression $\mathbf{t}$ need not be an integer expression: any set Γ on which there is a well-founded order $\preccurlyeq$, is a suitable type for $\mathbf{t}$. To prove that the value of $\mathbf{t}$ is reduced it is sufficient to prove that, if $\mathbf{t} \preccurlyeq t_0$ initially (where t_0 is an otherwise unused variable), then the assertion $\{\mathbf{t} \prec t_0\}$ can be inserted before each occurrence of $\mathbf{S}'$ in $\mathbf{S}[\mathbf{S}'/F]$. The theorem combines our two requirements into a single condition:

Theorem A.1 *If $\preccurlyeq$ is a well-founded partial order on some set Γ and $\mathbf{t}$ is an expression giving values in Γ and t_0 is a variable which does not occur in $\mathbf{S}$ then if for some premiss $\mathbf{P}$ we have:*

$$\{\mathbf{P} \wedge \mathbf{t} \preccurlyeq t_0\};\ \mathbf{S}' \leq \mathbf{S}[\{\mathbf{P} \wedge \mathbf{t} \prec t_0\};\ \mathbf{S}'/F]$$

then

$$\{\mathbf{P}\};\ \mathbf{S}' \leq \textsf{proc}\ F \equiv \mathbf{S}.$$

Proof: See (Ward 1989) for the proof. This theorem is based on Dijkstra's technique for reasoning about **while** loops using weakest preconditions (Dijkstra 1976).

It is frequently possible to *derive* a suitable procedure body $\mathbf{S}$ from the statement $\mathbf{S}'$ by applying transformations to $\mathbf{S}'$, splitting it into cases etc., until we get the statement $\mathbf{S}[\mathbf{S}'/F]$ which is still defined in terms of $\mathbf{S}'$.

[2] Such cleverness is not always an unmixed blessing. I believe it was Dijkstra who remarked that maintaining a program is acknowledged to be more difficult than writing the program in the first place, *"So if you are as clever as you can be when you are writing the program, how will you ever be able to maintain it?"*

If we can find a suitable variant function for $\mathbf{S}[\mathbf{S}'/F]$ then we can apply the theorem and refine $\mathbf{S}[\mathbf{S}'/F]$ to $\textsf{\textbf{proc}}\ F \ \equiv\ \mathbf{S}.$ which is no longer defined in terms of $\mathbf{S}'$.

A simple example of such a derivation is the familiar factorial function. Let $\mathbf{S}'$ be the statement $r := n!$, where $n!$ is the usual factorial function, defined as:

$$0! = 1 \quad \text{and for all } n \geqslant 0\text{:} \quad (n+1)! = (n+1) * n!$$

We can transform the assignment $r := n!$ (by appealing to the definition of factorial) to show that:

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

Separate the assignment:

$$\begin{aligned}\mathbf{S}' \approx\ & \textsf{\textbf{if}}\ n = 0 \\ & \quad \textsf{\textbf{then}}\ r := 1 \\ & \quad \textsf{\textbf{else}}\ n := n-1;\ r := n!; \\ & \qquad n := n+1;\ r := n * r\ \textsf{\textbf{fi}}\end{aligned}$$

So we have:

$$\begin{aligned}\mathbf{S}' \approx\ & \textsf{\textbf{if}}\ n = 0 \\ & \quad \textsf{\textbf{then}}\ r := 1 \\ & \quad \textsf{\textbf{else}}\ n := n-1;\ \mathbf{S}'; \\ & \qquad n := n+1;\ r := n * r\ \textsf{\textbf{fi}}\end{aligned}$$

The positive integer n is decreased before the copy of $\mathbf{S}'$, so if we set $\mathbf{t}$ to be n, Γ to be $\mathbb{N}$ and $\preccurlyeq$ to be $\leqslant$ (the usual order on natural numbers), and $\mathbf{P}$ to be **true** then we can prove:

$$\begin{aligned}& \{n \leqslant t_0\};\ \mathbf{S}' \leq \\ & \quad \textsf{\textbf{if}}\ n = 0 \\ & \qquad \textsf{\textbf{then}}\ r := 1 \\ & \qquad \textsf{\textbf{else}}\ n := n-1;\ \{n < t_0\};\ \mathbf{S}'; \\ & \qquad\quad n := n+1;\ r := n * r\ \textsf{\textbf{fi}}\end{aligned}$$

So we can apply Theorem A.1 to get:

$$\begin{aligned}& \mathbf{S}' \leq \\ & \quad \textsf{\textbf{proc}}\ F\ \equiv \\ & \qquad \textsf{\textbf{if}}\ n = 0 \\ & \qquad\quad \textsf{\textbf{then}}\ r := 1 \\ & \qquad\quad \textsf{\textbf{else}}\ n := n-1;\ F(); \\ & \qquad\qquad n := n+1;\ r := n * r\ \textsf{\textbf{fi.}}\end{aligned}$$

and we have derived a recursive implementation of factorial.

This theorem is a fundamental result towards the aim of a system for transforming specifications into programs, since it "bridges the gap" between a recursively defined specification, and a recursive procedure which implements it. It is of use even when the final program is iterative rather than recursive, since many algorithms may be more easily and clearly specified as recursive functions—even if they may be more efficiently implemented as iterative procedures. This theorem may be used by the programmer to transform the recursively defined specification into a recursive procedure or function, which can then be transformed into an iterative procedure using Theorem A.4 below.

A.2 The Induction Rule for Recursion

Our second transformation shows that to prove a refinement of a recursive or iterative program, it is sufficient to examine the set of "finite truncations" of the program. This result is extremely valuable in proving many transformations involving recursive and iterative statements since, in a great many cases, the proof can be carried out by induction over the set of all finite truncations. The theorem shows that the set of all finite truncations of a recursive statement tells us everything we need to know about the full recursion. Using this induction rule we have proved a powerful collection of general purpose transformations. These enable many algorithm derivations to be carried out by appealing to general transformation rules rather than *ad hoc* induction proofs.

The nth truncation of a procedure $\textsf{\textbf{proc}}\ F \ \equiv\ \mathbf{S}.$ is defined recursively:

$(\textsf{\textbf{proc}}\ F \ \equiv\ \mathbf{S}.)^0 \ =_{\text{DF}}\ \textsf{\textbf{abort}}$

and

$(\textsf{\textbf{proc}}\ F \ \equiv\ \mathbf{S}.)^{n+1} \ =_{\text{DF}}\ \mathbf{S}[(\textsf{\textbf{proc}}\ F \ \equiv\ \mathbf{S}.)^n/F]$

Here, the notation $=_{\text{DF}}$ indicates that the left hand side of the symbol is defined to mean the right hand side, and should be distinguished from the notation $\approx$ which means that the statement on the left hand side is semantically equivalent to the statement on the right hand side.

The nth truncation of any statement $\mathbf{S}^n$ is formed by replacing each recursive component by its nth truncation.

A statement has *bounded nondeterminacy* if each specification statement within it has a finite set of values it can assign to the variables to satisfy the given condition. For statements with bounded nondeterminacy we have the following induction rule:

Theorem A.2 *The Induction Rule for Recursion:* If $\mathbf{S}$ is any statement with bounded nondeterminacy, and $\mathbf{S}'$ is another statement such that for every $n < \omega$, $\mathbf{S}^n \leq \mathbf{S}'$, then $\mathbf{S} \leq \mathbf{S}'$.

Proof: See (Ward 1989).

This transformation is related to the concept of a sequence of approximations to a continuous function which is fundamental to denotational semantics (Stoy 1977; Tennet 1976)—the semantics of the truncations $\mathbf{S}^n$ form a sequence of approximations to the semantics of $\mathbf{S}$.

An example of a transformation proved by induction is the following:

Theorem A.3 *Invariant Maintenance*

(i) *If for any statement* $\mathbf{S}_1$ *we can prove:*

$$\{\mathbf{P}\};\ \mathbf{S}[\mathbf{S}_1/X] \ \leq\ \mathbf{S}[\{\mathbf{P}\};\ \mathbf{S}_1/X]$$

then:

$$\{\mathbf{P}\};\ \textsf{\textbf{proc}}\ X \ \equiv\ \mathbf{S}. \ \leq\ \textsf{\textbf{proc}}\ X \ \equiv\ \{\mathbf{P}\};\ \mathbf{S}.$$

(ii) *If in addition*

$$\{\mathbf{P}\};\ \mathbf{S}_1 \ \leq\ \mathbf{S}_1;\ \{\mathbf{P}\}$$

implies

$$\{\mathbf{P}\};\ \mathbf{S}[\mathbf{S}_1/X]\ \leq\ \mathbf{S}[\mathbf{S}_1/X];\ \{\mathbf{P}\}$$

then

$$\{\mathbf{P}\};\ \textsf{proc}\ X\ \equiv\ \mathbf{S}.\ \leq\ \textsf{proc}\ X\ \equiv\ \mathbf{S}.;\ \{\mathbf{P}\}$$

Proof: (i) Claim:

$$\{\mathbf{P}\};\ (\textsf{proc}\ X\ \equiv\ \mathbf{S})^n\ \leq\ (\textsf{proc}\ X\ \equiv\ \{\mathbf{P}\};\ \mathbf{S})^n$$

If this claim is proved then the result follows from the induction rule for recursion (Theorem A.2). We prove the claim by induction on n. For $n = 0$ both sides are **abort**, so suppose the result holds for n.

Put $\mathbf{S}_1 = (\textsf{proc}\ X\ \equiv\ \mathbf{S})^n$ in the premise. Then:

$$\begin{aligned}&\{\mathbf{P}\};\ (\textsf{proc}\ X\ \equiv\ \mathbf{S})^{n+1}\\ &\quad\leq \{\mathbf{P}\};\ \mathbf{S}[(\textsf{proc}\ X\ \equiv\ \mathbf{S})^n/X]\\ &\quad\leq (\{\mathbf{P}\};\ \mathbf{S})[\{\mathbf{P}\};\ (\textsf{proc}\ X\ \equiv\ \mathbf{S})^n/X]\end{aligned}$$

from the premise

$$\leq (\{\mathbf{P}\};\ \mathbf{S})[(\textsf{proc}\ X\ \equiv\ \{\mathbf{P}\};\ \mathbf{S})^n/X]$$

by the induction hypothesis

$$\leq (\textsf{proc}\ X\ \equiv\ \{\mathbf{P}\};\ \mathbf{S})^{n+1}$$

The result follows by induction on n.

(ii) Claim: $\{\mathbf{P}\};\ (\textsf{proc}\ X\ \equiv\ \mathbf{S})^n\ \leq\ (\textsf{proc}\ X\ \equiv\ \{\mathbf{P}\};\ \mathbf{S})^n;\ \{\mathbf{P}\}$ for all n. Again, we prove the claim by induction on n: For $n = 0$ both sides are **abort**, so suppose the result holds for n.

$$\begin{aligned}&\{\mathbf{P}\};\ (\textsf{proc}\ X\ \equiv\ \{\mathbf{P}\};\ \mathbf{S})^{n+1}\\ &\quad\leq \{\mathbf{P}\};\ \mathbf{S}[(\textsf{proc}\ X\ \equiv\ \mathbf{S})^n/X]\\ &\quad\leq \{\mathbf{P}\};\ \mathbf{S}[\{\mathbf{P}\};\ (\textsf{proc}\ X\ \equiv\ \mathbf{S})^n/X]\ \text{by part (i)}\end{aligned}$$

Put $\mathbf{S}_1 = (\textsf{proc}\ X\ \equiv\ \mathbf{S})^n$ in the premise and use:
$\{\mathbf{P}\};\ (\textsf{proc}\ X\ \equiv\ \mathbf{S})^n\ \leq\ (\textsf{proc}\ X\ \equiv\ \mathbf{S})^n;\ \{\mathbf{P}\}$ to get:

$$\leq (\textsf{proc}\ X\ \equiv\ \mathbf{S})^{n+1};\ \{\mathbf{P}\}$$

from premise (ii).
The result follows by induction on n. ∎

A.3 General Recursion Removal

Our third 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 1992), 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 (where a **call** Z appearing in one of the action bodies in the action system will terminate the action system, and hence only the current invocation of the procedure):

$$\begin{aligned}&\textsf{proc}\ F(x)\ \equiv\\ &\quad\textsf{actions}\ A_1:\\ &\quad A_1\ \equiv\\ &\qquad\mathbf{S}_1.\\ &\quad\ldots\\ &\quad A_M\ \equiv\\ &\qquad\mathbf{S}_M.\\ &\quad\ldots\\ &\quad B_j\ \equiv\\ &\qquad\mathbf{S}_{j0};\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\\ &\qquad F(g_{jn_j}(x));\ \mathbf{S}_{jn_j}.\\ &\quad\ldots\ \textsf{endactions.}\end{aligned}$$

The actions in action system which forms the body of the procedure are divided into two classes, the "A-type" actions A_i and the "B-type" actions B_j. The A-type action bodies may contain calls to any actions and assignments to any variables, but contain no calls to F. All the calls to F are as listed explicitly in the B-type actions, in addition ,the statements $\mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j}$ must preserve the value of x and the statements $\mathbf{S}_{j0}, \mathbf{S}_{j1} \ldots, \mathbf{S}_{jn_j-1}$ must contain no action calls (the $\mathbf{S}_{jn_j}$ statements will contain action calls). There are $M + N$ actions in total, M "A-type" actions $A_1, \ldots, A_M$ which contain no recursive calls, and N "B-type" actions $B_1, \ldots, B_N$ each of which contains one or more recursive calls. 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 $\langle 0, e\rangle$ 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 A.4 *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:*

$$\begin{aligned}&\textsf{proc}\ F'(x)\ \equiv\\ &\quad\textsf{var}\ \langle K := \langle\rangle, m := 0\rangle:\\ &\quad\textsf{actions}\ A_1:\\ &\quad A_1\ \equiv\\ &\qquad\mathbf{S}_1[\textsf{call}\ \hat{F}/\textsf{call}\ Z].\\ &\quad\ldots\\ &\quad A_M\ \equiv\\ &\qquad\mathbf{S}_M[\textsf{call}\ \hat{F}/\textsf{call}\ Z].\\ &\quad\ldots\\ &\quad B_j\ \equiv\\ &\qquad\mathbf{S}_{j0};\ K := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots,\\ &\qquad\langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle +\!\!\!+ K;\ \textsf{call}\ \hat{F}.\\ &\quad\ldots\\ &\quad \hat{F}\ \equiv\\ &\qquad\textsf{if}\ K = \langle\rangle\end{aligned}$$

then call Z
else $\langle m, x\rangle \xleftarrow{\text{pop}} K$;
if $m = 0 \rightarrow$ **call** A_1
$\square \ldots \square\ m = \langle j, k\rangle \rightarrow \mathbf{S}_{jk}[$**call** $\hat{F}/$**call** $Z]$;
call $\hat{F}$
$\ldots$ **fi fi. endactions end.**

Proof: See (Ward 1991; Ward 1992).

In contrast to the usual "iteration plus stack" method of recursion removal (discussed in (Knuth 1974) and elsewhere), in which only a single statement (the return point) is stacked, our method allows a whole sequence of recursive calls an intermediate statements to be stacked. *Any* recursive procedure can be restructured into a suitable form for Theorem A.4 simply by putting each recursive call into its own "B-type" action. Many recursive procedures can be restructured differently (but still meeting the requirements of the theorem) by collecting two or more recursive calls into "B-type" actions. These different recursive forms will lead to very different iterative versions of the program. See (Ward 1991; Ward 1992) for some examples and further applications of the theorem.

The proof of Theorem A.4 is rather involved and too long to include here. It relies on applying various transformations which have been proved using weakest preconditions, together with multiple applications of the general induction rule (Theorem A.2).

Corollary A.5 By unfolding some calls to $\hat{F}$ in B_j and pruning, we get a slightly more efficient version:

$B_j \equiv$
$\mathbf{S}_{j0}$;
$K := \langle\langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots,$
$\langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} K$;
$x := g_{j1}(x)$; **call** A_1.

In the case where $n_j = 1$ for all j, this version will never push a $\langle 0, x\rangle$ pair onto the stack. This fact can be significant for a parameterless procedure with a small number of j values, since it enables us to reduce the amount of storage required by the stack. For example, if there are two j values, the stack can be represented as a binary number.

A particularly simple case is a parameterless procedure with only one B action which contains only one recursive call. In this case, all the elements pushed on the stack will be equal, in fact they will all be $\langle 1, 1\rangle$, so we only need to record the *length* of the stack, ignoring its actual contents. Technically, we prove the following corollary by introducing a new local variable k which records the length of K, and then replacing the test $K = \langle\rangle$ by $k = 0$.

Corollary A.6 *The parameterless procedure:*

proc $F() \equiv$
actions A_1:
$A_1 \equiv$
$\mathbf{S}_1$.
$\ldots$
$A_M \equiv$
$\mathbf{S}_M$.
$B_1 \equiv$
$\mathbf{S}_{10}$; $F()$; $\mathbf{S}_{11}$. **endactions.**

(where the only recursive call is the single call in B_1) is equivalent to the non-recursive procedure:

proc $F() \equiv$
var $\langle k := 0\rangle$:
actions A_1:
$A_1 \equiv$
$\mathbf{S}_1[$**call** $\hat{F}/$**call** $Z]$.
$\ldots$
$A_M \equiv$
$\mathbf{S}_M[$**call** $\hat{F}/$**call** $Z]$.
$B_1 \equiv$
$\mathbf{S}_{10}$; $k := k + 1$; **call** A_1.
$\hat{F} \equiv$
if $k = 0$ **then call** Z
else $k := k - 1$; $\mathbf{S}_{11}$; **call** $\hat{F}$ **fi.**
endactions end.

Proof: This is a simple application of Theorem A.4 and Corollary A.5. Since there is but a single B-type action and no parameters, the stack K consists of a list of identical elements. Such a stack can be more efficiently implemented as an integer k, where $k = \ell(K)$, and $K = \langle\langle 1, 1\rangle, \langle 1, 1\rangle, \ldots\rangle$.

A.4 Recursion Removal Examples

Consider the simple recursive procedure:

proc $F(x) \equiv$
if $x = 0$ **then** $G(x)$
else $F(x-1)$; $H(x)$; $F(x-1)$ **fi.**

There are two ways to convert the body of the procedure into an action system appropriate for Theorem A.4. The first method is to put both recursive calls into the same B-type action:

proc $F(x) \equiv$
actions A_1:
$A_1 \equiv$
if $x = 0$ **then** $G(x)$; **call** Z **else call** B_1 **fi.**
$B_1 \equiv$
$F(x-1)$; $H(x)$; $F(x-1)$; **call** Z. **endactions.**

So for Theorem A.4, $\mathbf{S}_1$ is the statement

if $x = 0$ **then** $G(x)$; **call** Z **else call** B_1 **fi**

$\mathbf{S}_{11}$ is the statement $H(x)$ and $\mathbf{S}_{12}$ is the statement **call** Z. Applying the theorem gives:

proc $F'(x) \equiv$
var $\langle K := \langle\rangle;\ m := 0\rangle$:
actions A_1:
$A_1 \equiv$
if $x = 0$ **then** $G(x)$; **call** $\hat{F}$ **else call** B_1 **fi.**
$B_1 \equiv$
$K := \langle\langle 0, x-1\rangle, \langle\langle 1, 1\rangle, x\rangle, \langle 0, x-1\rangle,$
$\langle\langle 1, 2\rangle, x\rangle\rangle \mathbin{+\!\!+} K$; **call** $\hat{F}$.
$\hat{F} \equiv$
if $K = \langle\rangle$
then call Z

else $\langle m, x\rangle \xleftarrow{\text{pop}} K$;
if $m = 0 \rightarrow$ **call** A_1
$\square\ m = \langle 1, 1\rangle \rightarrow H(x)$; **call** $\hat{F}$
$\square\ m = \langle 1, 2\rangle \rightarrow$ **call** $\hat{F}$; **call** $\hat{F}$ **fi fi.**
endactions.

We can represent the values $\langle 1, 1\rangle$ and $\langle 1, 2\rangle$ on the stack and in m by 1 and 2 respectively. Then, unfold everything into $\hat{F}$, replace the initial call to A_1 by $K := \langle\langle 0, x\rangle\rangle$, remove the recursion in $\hat{F}$ and then remove the action system:

proc $F'(x) \equiv$
var $\langle K := \langle\langle 0, x\rangle\rangle;\ m := 0\rangle$:
while $K \neq \langle\rangle$ **do**
$\langle m, x\rangle \xleftarrow{\text{pop}} K$;
if $m = 0 \rightarrow$ **if** $x = 0$
then $G(x)$
else $K := \langle\langle 0, x-1\rangle, \langle 1, x\rangle,$
$\langle 0, x-1\rangle, \langle 2, x\rangle\rangle +\!\!\!+\ K$ **fi**
$\square\ m = 1 \rightarrow H(x)$; **call** $\hat{F}$
$\square\ m = 2 \rightarrow$ **skip fi od end.**

The other way to restructure the recursive program is to put the two recursive calls into separate B-type actions:

proc $F(x) \equiv$
actions A_1:
$A_1 \equiv$
if $x = 0$ **then** $G(x)$; **call** Z **else call** B_1 **fi.**
$B_1 \equiv$
$F(x-1)$; $H(x)$; **call** B_2**.**
$B_2 \equiv$
$F(x-1)$; **call** Z**. endactions.**

So for Theorem A.4, $\mathbf{S}_1$ is the statement

if $x = 0$ **then** $G(x)$; **call** Z **else call** B_1 **fi**

$\mathbf{S}_{11}$ is the statement $H(x)$; **call** B_2 and $\mathbf{S}_{21}$ is the statement **call** Z. Applying the theorem gives:

proc $F'(x) \equiv$
var $\langle K := \langle\rangle;\ m := 0\rangle$:
actions A_1:
$A_1 \equiv$
if $x = 0$ **then** $G(x)$; **call** $\hat{F}$ **else call** B_1 **fi.**
$B_1 \equiv$
$K := \langle\langle 0, x-1\rangle, \langle\langle 1, 1\rangle, x\rangle\rangle +\!\!\!+\ K$; **call** $\hat{F}$**.**
$B_2 \equiv$
$K := \langle\langle 0, x-1\rangle, \langle\langle 2, 1\rangle, x\rangle\rangle +\!\!\!+\ K$; **call** $\hat{F}$**.**
$\hat{F} \equiv$
if $K = \langle\rangle$
then call Z
else $\langle m, x\rangle \xleftarrow{\text{pop}} K$;
if $m = 0 \rightarrow$ **call** A_1
$\square\ m = \langle 1, 1\rangle \rightarrow H(x)$; **call** B_2
$\square\ m = \langle 2, 1\rangle \rightarrow$ **call** $\hat{F}$; **call** $\hat{F}$ **fi fi.**
endactions.

Note that $\langle\langle 0, x-1\rangle\rangle +\!\!\!+\ K$; **call** $\hat{F}$ is equivalent to $x := x - 1$; **call** A_1. Then we never need to push $\langle 0, x\rangle$ onto K. Also we can represent the values $\langle 1, 1\rangle$ and $\langle 2, 1\rangle$ on the stack and in m by 1 and 2 respectively. Then, unfold B_1 into A_1, unfold B_2 into $\hat{F}$, remove the recursion in $\hat{F}$, unfold everything into A_1, remove the recursion and the action system:

proc $F'(x) \equiv$
var $\langle K := \langle\rangle;\ m := 0\rangle$:
do do if $x = 0$
then $G(x)$; **exit**
else $K := \langle\langle 1, x\rangle\rangle +\!\!\!+\ K$;
$x := x - 1$ **fi od**;
do if $K = \langle\rangle$ **then exit**(2) **fi**;
$\langle m, x\rangle \xleftarrow{\text{pop}} K$;
if $m = 1 \rightarrow H(x)$; $K := \langle\langle 2, x\rangle\rangle +\!\!\!+\ K$;
$x := x - 1$; **exit**
$\square\ m = 2 \rightarrow$ **skip fi od od end.**

Notice how the two different restructurings of the initial recursive procedure led to very different (but equivalent) interative procedures.

The power and generality of the recursion removal transformations comes from the fact that the body of the procedure is expressed as an action system, with the recursive calls collected into a number of actions. Because of this, a wide variety of recursive programs can be easily restructured into one or more forms, where the theorem can be applied. We can also apply the theorem in reverse to produce a recursive program from an iterative one.

Acknowledgements

The research described in this paper has been supported by an EPSRC[3] project "A Proof Theory for Program Refinement and Equivalence: Extensions". I would like to thank an anonymous referee for some helpful and perceptive comments on a previous version of the paper.

References

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

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

Arsac, J. (1982b): Syntactic Source to Source Program Transformations and Program Manipulation. Comm. ACM **22**, 1, pp. 43–54

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

Back, R.J.R. (1988): A Calculus of Refinements for Program Derivations. Acta Informatica **25**, Springer, pp. 593–624

Back, R.J.R., von Wright, J. (1990): Refinement Concepts Formalised in Higher-Order Logic. Formal Aspects of Computing **2**, Springer, pp. 247–272

[3]The UK Engineering and Physical Sciences Research Council

Bauer, F.L., Moller, B., Partsch, H., Pepper, P. (1989): Formal Construction by Transformation—Computer Aided Intuition Guided Programming. IEEE Trans. Software Eng. **15**, 2

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

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

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

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

Broy, M., Pepper, P. (1982): Combining Algebraic and Algorithmic Reasoning: an Approach to the Schorr-Waite Algorithm. Trans. Programming Lang. and Syst. **4**, 3

Bull, T. (1990): An Introduction to the WSL Program Transformer. Conference on Software Maintenance 26th–29th November 1990, San Diego

Burstall, R.M., McQueen, D.B., Sannella, D.T. (1980): HOPE: An Experimental Applicative Language. Department of Computer Science, University of Edinburgh, Internal Report

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

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

Feather, M.S. (1987): A Survey and Classification of Some Program Transformation Techniques. In: Meertens, L.G.L.T. (ed.) Proceedings of the IFIP TC2/WG2 Working Conference on Program Specification and Transformation. nhpc, Amsterdam, pp. 165–196

Feather, M.S. (1982): A System for Assisting Program Transformation. Trans. Programming Lang. and Syst. **4**, 1, pp. 1–20

Hildum, D., Cohen, J. (1990): A Language for Specifying Program Transformations. IEEE Trans. Software Eng. **16**, 6

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., Sufrin, B.A. (1987): Laws of Programming. Comm. ACM **30**, 8, pp. 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., Moore, R. (1991): mural: A Formal Development Support System. Springer, New York Berlin Heidelberg

Jorring, Scherlis (1987): Deriving and Using Destructive Data Types. In: Meertens, L.G.L.T. (ed.) Program Specification and Transformation: Proceedings of the IFIP TC2/WG 2.1 Working Conference, Bad Tölz, FRG, 15-17 April, 1986. 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. Surveys **6**, 4, pp. 261–301

Knuth, D.E., Szwarcfiter, J.L. (1974): A Structured Program to Generate All Topological Sorting Arrangements. Inform. Process. Lett. **2**, pp. 153–157

McMorran, M.A., Nicholls, J.E. (1989): Z User Manual. IBM UK Laboratories Ltd., Hursley Park. TR12.274

Morgan, C.C. (1988): The Specification Statement. Trans. Programming Lang. and Syst. **10**, pp. 403–419

Morgan, C.C. (1994): Programming from Specifications. Prentice-Hall, Englewood Cliffs, NJ. Second Edition

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

Morgan, C.C., Vickers, T. (1993): On the Refinement Calculus. Springer, New York Berlin Heidelberg

Neilson, M., Havelund, K., Wagner, K.R., Saaman, E. (1989): The RAISE Language, Method and Tools. Formal Aspects of Computing **1** , Springer, pp. 85–114

Pepper, P. (1979): A Study on Transformational Semantics. In: Goos, G., Hartmanis, H. (eds.) Program Construction. (Lecture Notes in Computer Science, vol. 69) Springer, New York Berlin Heidelberg, pp. 232–405

Priestley, H.A., Ward, M. (1994): A Multipurpose Backtracking Algorithm. J. Symb. Comput. **18**, pp. 1–40. ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/backtr-t.ps.gz`⟩

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

Stoy, J.E. (1977): Denotational Semantics: the Scott-Strachy Approach to Programming Language Theory. MIT Press, Cambridge, MA

Tennet, R.D. (1976): The Denotational Semantics of Programming Languages. Comm. ACM **19**, 8, pp. 437–453

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

Ward, M. (1990): Derivation of a Sorting Algorithm. Durham University, Technical Report

Ward, M. (1991): A Recursion Removal Theorem—Proof and Applications. Durham University, Technical Report

Ward, M. (1992): A Recursion Removal Theorem. Springer, New York Berlin Heidelberg. Proceedings of the 5th Refinement Workshop, London, 8th–11th January. ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/ref-ws-5.ps.gz`⟩

Ward, M. (1994a): Language Oriented Programming. Software—Concepts and Tools **15**, Springer, pp. 147–161. ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/middle-out-t.ps.gz`⟩

Ward, M. (1994b): Reverse Engineering through Formal Transformation Knuths "Polynomial Addition" Algorithm. Comput. J. **37**, 9, pp. 795–813. ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/poly-t.ps.gz`⟩

Ward, M. (1994c): Foundations for a Practical Theory of Program Refinement and Transformation. Durham University, Technical Report

Ward, M. (1993): Abstracting a Specification from Code. J. Software Maintenance: Research and Practice **5**, 2, John Wiley & Sons, pp. 101–122. ⟨`http://www.dur.ac.uk/~dcs0mpw/martin/papers/prog-spec.ps.gz`⟩

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

Ward, M., Bennett, K.H. (1993): A Practical Program Transformation System For Reverse Engineering. Working Conference on Reverse Engineering, May 21–23, 1993, Baltimore MA. ⟨http: // www. dur. ac. uk/ ~dcs0mpw/ martin/papers/icse.ps.gz⟩

Ward, M., Bennett, K.H. (1995a): Formal Methods to Aid the Evolution of Software. International Journal of Software Engineering and Knowledge Engineering **5**, 1, pp. 25–47. ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/evolution-t.ps.gz⟩

Ward, M., Bennett, K.H. (1995b): Formal Methods for Legacy Systems. J. Software Maintenance: Research and Practice **7**, 3, John Wiley & Sons, pp. 203–219. ⟨http: // www. dur. ac. uk/ ~dcs0mpw/ martin/ papers/ legacy-t.ps.gz⟩

Ward, M., Calliss, F.W., Munro, M. (1989): The Maintainer's Assistant. Conference on Software Maintenance 16th–19th October 1989, Miami Florida. ⟨http: // www. dur.ac.uk/~dcs0mpw/martin/papers/MA-89.ps.gz⟩

Wile, D. (1981): Type Transformations. IEEE Trans. Software Eng. **7**, 1

Wossner, H., Pepper, P., Partsch, H., Bauer, F.L. (1979): Special Transformation Techniques. In: Goos, G., Hartmanis, H. (eds.) Program Construction. (Lecture Notes in Computer Science, vol. 69) Springer, New York Berlin Heidelberg, pp. 290–321

Zuylen, H.J.van (ed.) (1992): The REDO Compendium: Reverse Engineering for Software Maintenance.. John Wiley & Sons, New York, NY