Recursion Removal/Introduction by Formal Transformation: An Aid to Program Development and Program Comprehension

M. P. Ward and K. H. Bennett

Department of Computer Science University of Durham, Durham, UK
Email: Martin.Ward@durham.ac.uk, Keith.Bennett@durham.ac.uk

The transformation of a recursive program to an iterative equivalent is a fundamental operation in Computer Science. In the reverse direction, the task of *reverse engineering* (analysing a given program in order to determine its specification) can be greatly ameliorated if the program can be re-expressed in a suitable recursive form. But the existing recursion removal transformations, such as the techniques discussed by Knuth [1] and Bird [2], can only be applied in the reverse direction if the source program happens to match the structure produced by a particular recursion removal operation. In this paper we describe a much more powerful recursion removal and introduction operation which describes its source and target in the form of an *action system* (a collection of labels and calls to labels). A simple, mechanical, restructuring operation can be applied to a great many iterative programs which will put them in a suitable form for recursion introduction. Our transformation generates strictly more iterative versions than the standard methods, including those of Knuth and Bird [1,2]. With the aid of this theorem we prove a (somewhat counterintuitive) result for programs that contain sequences of two or more recursive calls: under a reasonable commutativity condition, "depth-first" execution is more general than "breadth-first" execution. In "depth-first" execution, the execution of each recursive call is completed, including all sub-calls, before execution of the next call is started. In "breadth-first" execution, each recursive call in the sequence is partially executed but any sub-calls are temporarily postponed. This result means that any breadth-first program can be reimplemented as a corresponding depth-first program, but the converse does not hold. We also treat the case of "random-first" execution, where the execution order is implementation dependent. For the more restricted domain of tree searching we show that breadth first search is the most general form. We also give two examples of recursion introduction as an aid to formal reverse engineering.

Keywords: Recursion, Formal Methods, Reverse Engineering, Re-engineering, WSL, Refinement, Program Comprehension

Received 16th March 1999; accepted 8th September 1999

1. INTRODUCTION

The transformation of a recursive program to an iterative equivalent is a fundamental operation in Computer Science. In the reverse direction, the task of *reverse engineering* (analysing a given program in order to determine its specification) can be greatly ameliorated if the program can be re-expressed in a suitable recursive form. But the existing recursion removal transformations, such as the techniques discussed by Knuth [1] and Bird [2], can only be applied in the reverse direction if the source program happens to match the structure produced by a particular recursion removal operation.

The authors have developed a *wide-spectrum language* which includes both abstract specifications and low-level programming constructs in a single language. This has the advantage that one does not need to differentiate between programming and specification languages: the entire transformational development of a program from abstract specification to detailed implementation can be carried out in the same language. Conversely, the entire reverse-engineering process, from

a transliteration of the source program to high-level specification can also be carried out in the same language. During either of these processes, different parts of the program may be expressed at different levels of abstraction. So a wide-spectrum language forms an ideal tool for developing methods for formal program development and also for formal reverse engineering (for which we have coined the term *inverse engineering*).

Over the last eight years we have been developing this language (called WSL), in parallel with the development of a transformation theory and proof methods. Over this time the language has developed from a simple and tractable kernel language [3,4] to a complete and powerful programming language. At the "low-level" end of the language there exists an automatic translator from IBM Assembler into WSL. At the "high-level" end it is possible to write high-level, abstract specifications, similar to **Z** and VDM specifications.

The WSL language includes constructs for loops with multiple exits, action systems, side-effects etc. and the transformation theory includes a large catalogue of proven transformations for manipulating these constructs. Many of the transformations have been implemented in the FermaT transformation engine developed by Software Migrations Ltd. [5–7].

In [8–10] program transformations are used to derive a variety of efficient algorithms from abstract specifications. In [11] the same transformations are used in the reverse direction: starting with a small but tangled and obscure program we were able to use transformations to restructure the program and derive a concise abstract representation of its specification.

1.1. Transformation Methods

The *Refinement Calculus* approach to program derivation [12–14] is superficially similar to our program transformation method. It is based on a wide spectrum language, using Morgan's specification statement [15] and Dijkstra's guarded commands [16]. However, this language has very limited programming constructs: lacking loops with multiple exits, action systems, and expressions with 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. ("The refinement law for iteration *relies on* capturing the potentially unbounded repetition in a single formula, the invariant", [12] p. 60, my emphasis). This makes the method unsuitable for a practical reverse-engineering method where such all-encompassing invariants are not readily available. By contrast, our WSL language has all these constructs, together with an extensive catalogue of proven transformations for manipulating programs which use them.

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 μral [17], RAISE [18] and the B-tool [19] 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 in [20] indicates that for "real" sized programs it is impractical to discharge 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, *as the first step in the process*. This is very difficult to do in general, especially for programs which have not been developed according to some structured programming method. In contrast, our approach does not depend on proof obligations: the user simply chooses which transformation to apply and only has to check that the applicability condition for the transformation is satisfied. In most cases, these applicability conditions are quite straightforward and can be mechanically checked: the FermaT transformation engine automatically checks the applicability condition before applying any transformation.

The Munich project CIP (Computer-aided Intuition-guided Programming) [21–23] uses a wide-spectrum language based on algebraic specifications and an applicative kernel language. However this approach does have some problems with the numbers of axioms required, and the difficulty of determining the exact correctness conditions of transformations. These problems are greatly exacerbated when imperative constructs are added to the system.

Problems with purely algebraic specification methods have been noted by Majester [24]. She presents an abstract data type with a simple constructive definition, but which requires several infinite sets of axioms to define algebraically. Another point is that it is important for any algebraic specification to be consistent, and the usual method of proving consistency is to exhibit a model of the axioms. Since every algebraic specification

requires a model, while not every model can be specified algebraically, there seems to be some advantages in rejecting algebraic specifications and working directly with models.

1.2. Our Approach

In developing a model based theory of semantic equivalence, we use the popular approach of defining a core "kernel" language with denotational semantics, and permitting definitional extensions in terms of the basic constructs. See [4,25] for a description of the kernel language. In contrast to other work (for example, [23,26, 27]) we do not use a purely applicative kernel; instead, the concept of state is included, using a *specification statement* which also allows specifications expressed in first order logic as part of the language, thus providing a genuine wide spectrum language.

Fundamental to our approach is the use of infinitary first order logic (see [28]) both to express the weakest preconditions of programs (see [16]) and to define assertions and guards in the kernel language. *Infinitary* logic is an extension of the usual first order logic which allows conjunction and disjunction over infinite sequences of formulae. The particular logic we use, $\mathcal{L}_{\omega_1\omega}$, allows conjunction and disjunction over any *countably* infinite sequence of formulae, and quantification over finite sets of variables.

Engeler [29] was the first to use infinitary logic to describe properties of programs; Back [30] used such a logic to express the weakest precondition of a program as a logical formula. His kernel language was limited to simple iterative programs. We use a different kernel language which includes recursion and guards, so that Back's language can be constructed from a subset of ours. In [4] we show that the introduction of infinitary logic as part of the language (rather than just the metalanguage of weakest preconditions), together with a combination of proof methods using both denotational semantics and weakest preconditions, is a powerful theoretical tool which allows us to prove some general transformations and representation theorems.

The denotational semantics of the kernel language is based on the semantics of infinitary first order logic. Kernel language statements are interpreted as functions which map an initial state to a set of final states (the *set* of final states models the nondeterminacy in the language: for a deterministic program this set will contain a single state). A program $\mathbf{S}_1$ is a refinement of $\mathbf{S}_2$ if, for each initial state, the set of final states for $\mathbf{S}_1$ is a subset of the final states for $\mathbf{S}_2$. Back and von Wright [31] note that the refinement relation can be characterised using weakest preconditions in higher order logic (where quantification over formulae is allowed). For any two programs $\mathbf{S}_1$ and $\mathbf{S}_2$, the program $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$ if the formula $\forall \mathbf{R}.\, \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ is true. This approach to refinement has two problems:

1. It can be difficult to find a finite formula which characterises the weakest precondition of a general loop or recursive statement. Suitable invariants can sometimes provide a sufficiently good approximation to the weakest precondition but, as already noted, these can be difficult to discover for large and complex programs;
2. Second order logic is *incomplete* in the sense that not all true statements are provable. So even if the refinement is true, it may not be possible to prove it.

In [4] we solve both of these problems. Using infinitary logic allows us to give a simple definition of the weakest precondition of *any* statement (including an arbitrary loop) for any postcondition. In addition, we show that for each pair of statements $\mathbf{S}_1$ and $\mathbf{S}_2$ there is a single postcondition $\mathbf{R}$ such that $\mathbf{S}_1$ is a refinement of $\mathbf{S}_2$ iff $\mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ and $\mathrm{WP}(\mathbf{S}_1, \mathbf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{true})$ are both true. Thus, there is no need for the universal quantification over all postconditions. In addition, the infinitary logic we use is complete, so if there is a refinement then there is also guaranteed to be a proof of the refinement. Thus infinitary logic is both necessary and sufficient for proving refinements and transformations. In this paper we give some examples to illustrate the effectiveness of our transformational approach to algorithm derivation and program analysis.

We consider the following criteria to be important for any practical wide-spectrum language and transformation theory:

1. General specifications in any "sufficiently precise" notation should be included in the language. For "sufficiently precise" we will mean anything which can be expressed in terms of mathematical logic with suitable notation. This will allow a wide range of forms of specification, for example **Z** specifications [32] and **VDM** [33] both use the language of mathematical logic and set theory (in different notations) to define specifications. The "Representation Theorem" (see [4]) proves that our specification statement is sufficient to specify *any* WSL program (and therefore any computable function);
2. Nondeterministic programs. We do not want to have to specify *everything* about the program we are working with, and certainly not in the first versions. So we need some way of specifying that some executions will not necessarily result in a particular outcome, but one of an allowed range of outcomes. The implementor can then use this latitude to provide a more efficient implementation which still satisfies the specification;
3. A well-developed catalogue of proven transformations which do not require the user to discharge complex proof obligations before they can be applied. In particular, it should be possible to introduce, analyse and reason about imperative and

recursive constructs without requiring loop invariants;
4. Techniques to bridge the "abstraction gap" between specifications and programs. See Section 8 and [11,34] for examples;
5. Applicable to real programs—not just those in a "toy" programming language with few constructs. This is essential if the theory is to be useful for reverse engineering as well as forward engineering. Our transformation theory, and the FermaT engine which supports it, have been used on some large legacy assembler systems for both program comprehension and migration to high-level languages. See [6,7];
6. Scalable to large programs: this demands a language which is expressive enough to allow automatic translation from existing programming languages, together with the ability to cope with unstructured programs and a high degree of complexity. The largest program we have worked on was a single assembler module which contained 28,000 lines of highly complex source code. This expanded into 37,500 lines of assembler listing and the initial translation into WSL code required over 123,000 lines of WSL. Compare this with a typical assembler module, which contains less than 500 lines of source. See [35] for more examples.

2. NOTATION

In this section we briefly define some of the notation for WSL which we use later in the paper. See [3,4] for a fuller description of WSL.

2.1. Expressions and Conditions in WSL

Expressions in WSL may use all of the usual mathematical operators. For conditions in WSL we may use any formulae of infinitary logic: including "non-computable" formulae such as quantification over infinite sets (for example $\forall n \in \mathbb{N} \ldots$.). For simplicity we assume that all expressions and conditions are everywhere defined (possibly returning a special error value).

2.2. The Specification Statement

Given a formula $\mathbf{Q}$ of infinitary logic, a sequence $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$ of variables, and a corresponding sequence $\mathbf{x}' = \langle x'_1, \ldots, x'_n \rangle$ of "primed" variables, the specification statement is written:

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q}$$

This assigns new values $\mathbf{x}'$ to $\mathbf{x}$ such that the formula $\mathbf{Q}$ is true. $\mathbf{Q}$ defines the relationship between the old values of $\mathbf{x}$ and the new values (represented by $\mathbf{x}'$ in $\mathbf{Q}$). If there are no values which satisfy $\mathbf{Q}$ then the statement does not terminate. For example, the assignment $\langle x \rangle := \langle x' \rangle.(x = 2 * x' \ \wedge \ x \in \mathbb{Z})$ halves x (by binding a new value to x) if it is even and aborts if x is odd.

If $\mathbf{e}$ is a sequence of expressions of the same length as $\mathbf{x}$, then we write $\mathbf{x} := \mathbf{e}$ as an abbreviation for $\mathbf{x} := \mathbf{x}'.(\mathbf{x}' = \mathbf{e})$. If there is one variable in $\mathbf{x}$ and one expression in $\mathbf{e}$, then the sequence brackets can be omitted, i.e. $\langle x \rangle := \langle e \rangle$ can be abbreviated to $x := e$.

2.3. Conditional Statements

WSL includes two kinds of **if** statement, the usual (deterministic) **if** statement:

if $\mathbf{B}_1$ **then** $\mathbf{S}_1$
elsif $\mathbf{B}_2$ **then** $\mathbf{S}_2$
$\ldots$
else $\mathbf{S}_n$ **fi**

which tests each of the conditions $\mathbf{B}_1$, $\mathbf{B}_2$, ... in turn and executes the statement corresponding to the first true condition. The **else** clause is equivalent to **elsif true then** $\mathbf{S}_n$.

The other **if** statement is Dijkstra's guarded command:

if $\mathbf{B}_1 \rightarrow \mathbf{S}_1$
$\Box$ $\mathbf{B}_2 \rightarrow \mathbf{S}_2$
$\Box \ldots$
$\Box$ $\mathbf{B}_n \rightarrow \mathbf{S}_n$ **fi**

Here, *all* the conditions $\mathbf{B}_1$, ... $\mathbf{B}_n$ are evaluated. If any condition is true, then one of the statements corresponding to a true condition is selected (in a nondeterministic way) for execution. If none of the conditions evaluates to true, then the statement aborts. Note that with a guarded command, the order of the clauses is irrelevant.

2.4. Loops

As well as the usual **for** and **while** loops, there is a notation for unbounded loops. Statements of the form **do S od**, where **S** is a statement, are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form **exit**(n) which causes the program to exit the n enclosing loops. We use **exit** as an abbreviation for **exit**(1). To simplify the language we disallow **exit**s which leave a block or a loop other than an unbounded loop. We also insist that n be an integer, *not* a variable or expression—this ensures that we can always determine the target of the **exit**. This type of structure is described in [1] and more recently in [36].

Dijkstra's guarded command loop:

do $\mathbf{B}_1 \rightarrow \mathbf{S}_1$
$\Box$ $\mathbf{B}_2 \rightarrow \mathbf{S}_2$
$\Box \ldots$
$\Box$ $\mathbf{B}_n \rightarrow \mathbf{S}_n$ **od**

is equivalent to the following **while** loop:

while $\mathbf{B}_1 \vee \mathbf{B}_2 \vee \cdots \vee \mathbf{B}_n$ **do**

if $\mathbf{B}_1 \to \mathbf{S}_1$
$\square$ $\mathbf{B}_2 \to \mathbf{S}_2$
$\square$ ...
$\square$ $\mathbf{B}_n \to \mathbf{S}_n$ **fi od**

2.5. Local Variables

The structure **var** $\langle x := e, y := f \rangle$: **S end** introduces a block with new local variables x and y with initial values e and f respectively. If x and y are already present in the state space, then their original values are saved (on a stack) and restored at the end of the block.

2.6. Procedures and Functions with Parameters

We use the following notation for procedures with parameters:

begin $\mathbf{S}_1$
where
proc $F(x, y) \equiv \mathbf{S}_2$.
end

where $\mathbf{S}_1$ is a program containing calls to the procedure F which has parameters x and y. The body $\mathbf{S}_2$ of the procedure may contain recursive procedure calls to F. We use a similar notation (with **funct** instead of **proc**) for function calls:

begin $\mathbf{S}_1$
where
funct $G(x, y) \equiv$
  var $\langle v_1 := e_1, v_2 := e_2 \rangle$:
  $\mathbf{S}_2$;
  (e).
end

Where $\mathbf{S}_1$ is a program containing calls to the function G which has parameters x and y. The body of G also has local variables v_1 and v_2 which are initialised to e_1 and e_2 respectively. The final expression e (which may contain references to local variables and parameters) gives the returned value of the function. The local variables and/or the statement $\mathbf{S}_2$ may be omitted.

2.7. Action Systems

An *action system* is a set of parameterless mutually recursive procedures [37–39]. If the end of the body of an action is reached, then control is returned to the calling action, or to the statement following the action system if there was no calling action, rather than "falling through" to the next action. The exception to this is a special action called the terminating action, usually denoted Z, which when called results in the immediate termination of the whole action system, with execution continuing after the action system. A program written using labels and jumps translates directly into an action system by translating each labelled block of code as an action, and each **goto** as an action call. Where one block of code "falls through" to the next, we add an explicit action call. At the end of the program we add a **call** Z to terminate the action system.

An action system is written as follows, with the first action to be executed (A_1 below) named at the beginning:

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 particular action system, execution of an action body must lead to an action call, so the system can only terminate by calling the Z action (which causes immediate termination). Such action systems are called *regular*. Note that an action system is itself a statement and may occur as a component of another statement—including another action system. Also, note that since each action body either returns to the caller or explicitly calls another action body, it does not matter in what order the action bodies are listed.

For the rest of this paper, all action systems will be *regular* and can be thought of as labelled blocks of code with each block of code explicitly calling the next block to be executed.

2.8. Sequences

$s = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequence, the ith element a_i is denoted $s[i]$, $s[i\,..\,j]$ is the subsequence $\langle s[i], s[i+1], \ldots, s[j] \rangle$, where $s[i\,..\,j] = \langle\rangle$ (the empty sequence) if $i > j$. The length of sequence s is denoted $\ell(s)$, so $s[\ell(s)]$ is the last element of s. We use $s[i\,..]$ as an abbreviation for $s[i\,..\,\ell(s)]$. The concatenation of s_1 and s_2 is defined: $s_1 +\!\!+ s_2 = \langle s_1[1], \ldots, s_1[\ell(s_1)], s_2[1], \ldots, s_2[\ell(s_2)] \rangle$.

Sequences are used to implement stacks and queues. For a sequence s and variable x, the notation: $x \xleftarrow{\text{pop}} s$ means $x := s[1]$; $s := s[2\,..]$ which pops an element off the stack into variable x. To push the value of the expression e onto stack s we use: $s \xleftarrow{\text{push}} e$ which represents: $s := \langle e \rangle +\!\!+ s$. The statement $x \xleftarrow{\text{last}} s$ removes the last element of s and stores its value in the variable x. It is equivalent to $x := s[\ell(s)]$; $s := s[1\,..\,\ell(s) - 1]$. The statement $x \xleftarrow{\text{pick}} s$ removes an arbitrary element from

the sequence s and stores it in x. It is equivalent to **var** $i := i'.(1 \leqslant i' \leqslant \ell(s))$; $x := s[i]$; $s := s[1\,..\,i-1] +\!\!+ s[i+1\,..]$ **end**. Statements $x \xleftarrow{\text{pop}} s$ and $x \xleftarrow{\text{last}} s$ are both valid refinements of $x \xleftarrow{\text{pick}} s$.

3. REFINEMENT IN WSL

A program $\mathbf{S}$ is a piece of formal text, i.e. a sequence of formal symbols which includes logical formulae in some logical language $\mathcal{L}$ as components. There are two ways in which we interpret (give meaning to) these texts:

1. Given a set of values and an interpretation of the symbols of $\mathcal{L}$ as functions and relations on the set of values, and given a final state space (from which we can construct a suitable initial state space), we can interpret a program as a function f (a *state transformation*) which maps each initial state s to the set of possible final states for s. By itself therefore, we can interpret a program as a function from structures to state transformations;
2. Given any formula $\mathbf{R}$ (which represents a condition on the final state), we can construct the formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$, the *weakest precondition* of $\mathbf{S}$ on $\mathbf{R}$. This is the weakest condition on the initial state such that the program $\mathbf{S}$ is guaranteed to terminate in a state satisfying $\mathbf{R}$ if it is started in a state satisfying $\mathrm{WP}(\mathbf{S}, \mathbf{R})$.

These interpretations give rise to two different notions of refinement: *semantic refinement* and *proof-theoretic refinement*.

3.1. Semantic Refinement

A *state* is a collection of variables (the *state space*) with values assigned to them; thus a state is a function which maps from a (finite, non-empty) set V of variables to a set $\mathcal{H}$ of values. There is a special extra state $\perp$ which is used to represent nontermination or error conditions. (It does not give values to any variables). A state transformation f maps each initial state s in one state space, to the set of possible final states $f(s)$, which may be in a different state space. If $\perp$ is in $f(s)$ then so is every other state, also $f(\perp)$ is the set of all states (including $\perp$).

Semantic refinement is defined in terms of these state transformations. A state transformation f is a refinement of a state transformation g if they have the same initial and final state spaces and $f(s) \subseteq g(s)$ for every initial state s. Note that if $\perp \in g(s)$ for some s, then $f(s)$ can be anything at all. In other words we can correctly refine an "undefined" program to do anything we please. If f is a refinement of g (equivalently, g is refined by f) we write $g \leq f$. A *structure* for a logical language $\mathcal{L}$ consists of a set of values, plus a mapping between constant symbols, function symbols and relation symbols of $\mathcal{L}$ and elements, functions and relations on the set of values. If the interpretation of statement $\mathbf{S}_1$ under the structure M is refined by the interpretation of statement $\mathbf{S}_2$ under the same structure, then we write $\mathbf{S}_1 \leq_M \mathbf{S}_2$. A *model* for a set of sentences (formulae with no free variables) is a structure for the language such that each of the sentences is interpreted as true. If $\mathbf{S}_1 \leq_M \mathbf{S}_2$ for every model M of a countable set Δ of sentences of $\mathcal{L}$ then we write $\Delta \models \mathbf{S}_1 \leq \mathbf{S}_2$.

3.2. Proof-Theoretic Refinement

If there exists a proof of a formula $\mathbf{Q}$ using a countable set Δ of sentences (formulae with no free variable) as assumptions, then we write $\Delta \vdash \mathbf{Q}$. Given two statements $\mathbf{S}_1$ and $\mathbf{S}_2$, and a formula $\mathbf{R}$, we can express the weakest preconditions $\mathrm{WP}(\mathbf{S}_1, \mathbf{R})$ and $\mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ as formulae in infinitary logic. (See [4]). Let $\mathbf{x}$ be a sequence of all variables assigned to in either $\mathbf{S}_1$ or $\mathbf{S}_2$ and let $\mathbf{x}'$ be a sequence of new variables. If the formulae $\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}')$ and $\mathrm{WP}(\mathbf{S}_1, \mathbf{true}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{true})$ are provable from the set Δ of sentences, then we say that $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ and write: $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$.

A fundamental result, proved by Ward [4] which generalises a theorem of Back's [30] is that these two notions of refinement are equivalent. More formally:

THEOREM 3.1. For any statements $\mathbf{S}_1$ and $\mathbf{S}_2$, and any countable set Δ of sentences of $\mathcal{L}$:

$$\Delta \models \mathbf{S}_1 \leq \mathbf{S}_2 \text{ if and only if } \Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$$

These two equivalent definitions of refinement give rise to two very different proof methods for proving the correctness of refinements. Both methods are exploited in [4]—weakest preconditions and infinitary logic are used to develop the induction rule for recursion and the recursive implementation theorem (Theorem 5.1), while state transformations are used to prove the representation theorem [4].

Two programs are *equivalent*, written $\Delta \vdash \mathbf{S}_1 \approx \mathbf{S}_2$ if and only if $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$ and $\Delta \vdash \mathbf{S}_2 \leq \mathbf{S}_1$. We write $\mathbf{S}_1 \leq \mathbf{S}_2$ as a shorthand for $\varnothing \vdash \mathbf{S}_1 \leq \mathbf{S}_2$ (where the set Δ is empty), and $\mathbf{S}_1 \approx \mathbf{S}_2$ for $\varnothing \vdash \mathbf{S}_1 \approx \mathbf{S}_2$.

4. THE INDUCTION RULE FOR RECURSION

A recursive statement such as **proc** $X \equiv \mathbf{S}.$ consists of a name (in this case the symbol X) and a body (in this case the statement $\mathbf{S}$) which may contain references to X. An occurrence of X as a component of $\mathbf{S}$ represents a recursive call to the procedure. We define the meaning of a recursive statement to be the "limit" of the infinite sequence of *truncations* of the recursion.

DEFINITION 4.1. The *nth truncation* of a recursive statement **proc** $X \equiv \mathbf{S}.$ is defined for $n < \omega$:

$$(\textbf{proc } X \equiv \mathbf{S}.)^0 =_{\text{DF}} \textbf{abort}$$
$$(\textbf{proc } X \equiv \mathbf{S}.)^{n+1} =_{\text{DF}} \mathbf{S}[(\textbf{proc } X \equiv \mathbf{S}.)^n / X]$$

The weakest precondition for recursion is defined as an infinite disjunction of all the finite truncations (cf [30] and [40]):

$$\mathrm{WP}(\textbf{proc } X \equiv \mathbf{S}., \mathbf{R}) =_{\mathrm{DF}} \bigvee_{n<\omega} \mathrm{WP}((\textbf{proc } X \equiv \mathbf{S}.)^n, \mathbf{R})$$

Each truncation is an approximation to the full recursion, the zeroth truncation is the poorest approximation (it provides no information at all), with later approximations refining the previous ones to approach the full recursion. A fundamental result of our transformation theory is that the sequence of truncations is sufficient to prove a refinement. More formally:

LEMMA 4.2. The Induction Rule for Recursion: If **proc** $X \equiv \mathbf{S}.$ is any recursive statement, and $\mathbf{S}'$ is any other statement such that $\Delta \vdash (\textbf{proc } X \equiv \mathbf{S}.)^n \leq \mathbf{S}'$ for all $n < \omega$, then $\Delta \vdash \textbf{proc } X \equiv \mathbf{S}. \leq \mathbf{S}'$.

This lemma forms the basis of a much more general (and useful) result. We can define the nth truncation $\mathbf{S}^n$ of an arbitrary statement $\mathbf{S}$ by replacing each recursive statement in $\mathbf{S}$ by its nth truncation. Then if $\mathbf{S}$ has bounded nondeterminacy (see [4]) we have:

LEMMA 4.3. The General Induction Rule for Recursion: If $\mathbf{S}$ is any statement with bounded nondeterminacy, and $\mathbf{S}'$ is another statement such that $\Delta \vdash \mathbf{S}^n \leq \mathbf{S}'$ for all $n < \omega$, then $\Delta \vdash \mathbf{S} \leq \mathbf{S}'$.

This lemma has proved extremely valuable in proving the correctness of many transformations involving recursion. Since iteration is defined using recursion, the result also applies to iterative statements. See [3,4] for the formal proof and many applications.

5. INTRODUCING RECURSION

Our next 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 [3,9] and [41].

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 5.1. 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

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

Proof: See [3].

6. GENERAL RECURSION REMOVAL

The following general purpose recursion removal transformation was presented in [25]. The proof may be found in [42].

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

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

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

Our aim is to remove the recursion by introducing a local stack L which records "postponed" operations:

When a recursive call is required we "postpone" it by pushing the pair $\langle 0, e\rangle$ onto L (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 $\mathbf{S}_{jk}$ and the current value of x, by pushing $\langle\langle j,k\rangle, x\rangle$ onto L. Where the procedure body would normally terminate (by calling Z) we instead call a new action $\hat{F}$ which pops the top item off L 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 6.1. A recursive procedure in the form given above is equivalent to the following iterative procedure which uses a new local stack L and a new local variable m:

proc $F'(x)$ $\equiv$
  var $L := \langle\rangle, m := 0$:
  actions A_1:
  A_1 $\equiv$
    $\mathbf{S}_1$[**call** $\hat{F}$/**call** Z].
  $\ldots A_i$ $\equiv$
    $\mathbf{S}_i$[**call** $\hat{F}$/**call** Z].
  $\ldots B_j$ $\equiv$
    $\mathbf{S}_{j0}$;
    $L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j,1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle,$
          $\ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle +\!\!+ L$;
    call $\hat{F}$.
  $\ldots \hat{F}$ $\equiv$
    if $L = \langle\rangle$
      then call Z
      else $\langle m, x\rangle \xleftarrow{\text{pop}} L$;
          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** $\hat{F}$
          ... **fi fi. endactions end.**

where the substitutions $\mathbf{S}_i$[**call** $\hat{F}$/**call** Z] are, of course, not applied to nested action systems which are components of the $\mathbf{S}_i$.

Note that any procedure $F(x)$ can be restructured into the required form; in fact there may be several different ways of structuring $F(x)$ which meet the requirements of the theorem.

The standard stack-based method of recursion removal [1,43] is the special case of this transformation where each B_j action contains a *single* function call. In other words, $n_j = 1$ for each j.

Corollary 6.1. By unfolding some calls to $\hat{F}$ in B_j and pruning, we get a slightly more efficient version:

B_j $\equiv$
  $\mathbf{S}_{j0}$;
  $L := \langle\langle\langle j,1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle,$
        $\ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle +\!\!+ L$;
  $x := g_{j1}(x)$; **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. In the extreme case where there is only one j value, the stack reduces to a sequence of ones, and can therefore be represented by an integer which simply records the length of the stack.

The power and generality of this transformation 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, see Section 11 for an example.

The only real disadvantage is that the recursive and iterative versions must carry out the same sequence of actions: the theorem only modifies the way in which the sequence of actions is defined. This is unavoidable when no restrictions are placed on the form of the recursive procedure: each statement can be modified to record (in a new variable) the exact sequence of operations carried out. The transformed version would have to preserve this variable, and therefore also preserve the sequence of operations. In the next section we show that by introducing some further restructions on the recursion, more radical transformations are possible.

7. CASCADE RECURSION REMOVAL

More restricted forms of recursive procedure can be transformed in more radical ways. A typical restriction is to a *cascade recursion* where the procedure carries out some processing followed by a sequence of zero or more recursive calls (with precomputed arguments) with no processing between the recursive calls. Such a procedure can be expressed in the following form:

proc $F(x)$ $\equiv$
  $\mathbf{S}_0$;
  if $\mathbf{B}_0 \rightarrow$ **skip**
  $\square \ldots \square$ $\mathbf{B}_j \rightarrow F(g_{j1}(x));\ F(g_{j2}(x));\ \ldots;\ F(g_{jn_j}(x))$
  ... **fi.**

where $\mathbf{S}_0$ is any statement which does not affect the value of x and does not call F. We will be considering various iterative versions of this procedure. We use the "procedure map" notation $F * L$ (where F is a procedure and L a list of values) to represent the sequence of calls $F(L[1]);\ F(L[2]);\ \ldots;\ F(L[\ell(L)])$. Using this notation we can write F more succinctly as:

proc $F(x)$ $\equiv$ $\mathbf{S}_0$; $F * G(x)$.

where $G(x)$ returns the (possibly empty) list of arguments for the inner recursive calls of $F(x)$:

funct $G(x)$ $\equiv$
  if $\mathbf{B}_0 \to \langle\rangle$
  $\square \ldots \square$ $\mathbf{B}_j \to \langle g_{j1}(x), g_{j2}(x), \ldots, g_{jn_j}(x)\rangle$
  $\ldots$ **fi.**

In the next three sections we consider three different iterative implementations of this recursion, and determine the conditions under which they are equivalent.

7.1. Depth-First Execution

A straightforward application of Theorem 6.1 to F will preserve the sequence of operations. In this case it will fully execute the first sub-call of F before starting on the second. Hence we term this method "depth-first execution".

First, we express the body of F as an action system:

proc $F(x)$ $\equiv$
  actions A_1 :
  A_1 $\equiv$
    $\mathbf{S}_0$;
    if $\mathbf{B}_0 \to$ **skip**
    $\square \ldots \square$ $\mathbf{B}_j \to$ **call** B_j
    $\ldots$ **fi.**
  $\ldots B_j$ $\equiv$
    $F(g_{j1}(x))$; $F(g_{j2}(x))$;
    $\ldots$; $F(g_{jn_j}(x))$; **call** Z.
  $\ldots$ **endactions.**

and then apply Theorem 6.1:

proc $F(x)$ $\equiv$
  var $L := \langle\rangle$:
  actions A_1:
  A_1 $\equiv$
    $\mathbf{S}_1$;
    if $\mathbf{B}_0 \to$ **skip**
    $\square \ldots \square$ $\mathbf{B}_j \to$ **call** B_j
    $\ldots$ **fi.**
  $\ldots B_j$ $\equiv$
    $L := \langle g_{j1}(x), g_{j2}(x), \ldots, g_{jn_j}(x)\rangle +\!\!+ L$;
    call $\hat{F}$.
  $\ldots \hat{F}$ $\equiv$
    if $L = \langle\rangle$ **then call** Z
      else $x \xleftarrow{\text{pop}} L$; **call** A_1 **fi.**
  endactions end.

Now unfold everything into $\hat{F}$. Instead of calling A_1 as the first action, we initialise L to $\langle x\rangle$ and call $\hat{F}$ as the first action. $\hat{F}$ is now a simple tail-recursion which can be replaced by a **while** loop:

proc $F(x)$ $\equiv$
  var $L := \langle x\rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \xleftarrow{\text{pop}} L$; $\mathbf{S}_0$;
    if $\mathbf{B}_0 \to$ **skip**
    $\square \ldots \square$ $\mathbf{B}_j \to L := \langle g_{j1}(x), g_{j2}(x),$
      $\ldots, g_{jn_j}(x)\rangle +\!\!+ L$
    $\ldots$ **fi od end.**

So F is equivalent to F_D where:

proc $F_D(x)$ $\equiv$
  var $L := \langle x\rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \xleftarrow{\text{pop}} L$; $\mathbf{S}_0$; $L := G(x) +\!\!+ L$ **od end.**

A typical application is searching a tree, where each branch is fully explored before we move to the next, and the sequence L records the branches which have yet to be explored.

7.2. Random First Execution

An alternative evaluation method is to pick a random element from the list L and call F with it. If this execution results in further recursive calls, these are not executed immediately, but pushed back onto L. The procedure terminates as soon as L becomes empty. A reasonable requirement on F is that it should be commutative, in other words, for any valid procedure arguments x and y the two statements $F(x)$; $F(y)$ and $F(y)$; $F(x)$ are equivalent. More formally:

DEFINITION 7.1. A procedure F is *commutative* if for any distinct variables x and y:

$$\Delta \vdash F(x);\ F(y) \ \approx\ F(y);\ F(x)$$

For example, the procedure

proc $F(x)$ $\equiv$ **if** $x = 0$ **then** $r := 1$ **fi**

is commutative since a sequence of calls $F(x)$; $F(y)$ is equivalent to the statement

if $x = 0 \ \vee\ y = 0$ **then** $r := 1$ **fi**

The next theorem shows that, with the addition of a well-foundedness condition on the domain, random-first execution is equivalent to depth-first execution:

THEOREM 7.1. Let F be a commutative cascade recursion. Suppose that there exists an irreflexive well-founded order $\prec$ on the domain of F such that $\mathbf{B}_j \Rightarrow g_{jk}(x) \prec x$ for all x in the domain, and all $1 \leqslant j \leqslant N$ and $1 \leqslant k \leqslant n_j$.
Then F is equivalent to F_R where:

proc $F_R(x)$ $\equiv$
  var $L := \langle x\rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \xleftarrow{\text{pick}} L$; $\mathbf{S}_0$; $L := G(x) +\!\!+ L$ **od end.**

The statement $x \xleftarrow{\text{pick}} L$ chooses a random element from the list L which is removed and assigned to x, i.e.

$x \xleftarrow{\text{pick}} L$ $=_{\text{DF}}$
  var $i := i'.(1 \leqslant i' \leqslant \ell(L))$:
  $x := L[i]$; $L := L[1\,..\,i-1] +\!\!+ L[i+1\,..]$ **end**

Proof: See Appendix. ■

The restriction that F be commutative means that the order in which sequences of F calls are evaluated can be changed. The list L records the list of arguments for which F has yet to be evaluated. We pick one at random and start to evaluate it (by executing $\mathbf{S}_0$). If this results in further calls these are not evaluated at this stage, but simply added to the list L. Since we always pick elements from L at random, the order in which elements are listed in L is irrelevant. So we could represent L as a bag, or a partial function (which records the number of occurrences of each element): the proof is slightly easier if we use a list.

Note that we can refine $x \xleftarrow{\text{pick}} L$ to $x \xleftarrow{\text{pop}} L$ which gives an iterative program equivalent to F (by Theorem 6.1) so we have proved $F_R(x) \leq F(x)$ without any restrictions on F or $\prec$. For the other direction, the requirement on $\prec$ cannot be dispensed with; consider the following example:

proc $F(x) \equiv$
  if even(x) **then** done $:= 1$ **fi**;
  if even$(x) \vee$ done $= 1 \rightarrow$ **skip**
  $\Box$ odd$(x) \rightarrow F(2 * x)$; $F(2 * x + 1)$ **fi.**

By unfolding the first few calls it is easy to see that this procedure terminates for any x: eventually an even argument will be evaluated and done will be set to 1, this causes all subsequent F calls to terminate immediately. So $F(x)$ is equivalent to done $:= 1$, and for every x and y therefore $F(x)$; $F(y) \approx F(y)$; $F(x)$. The transformed version is:

proc $F_R(x) \equiv$
  var $L := \langle x \rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \xleftarrow{\text{pick}} L$;
    if even(x) **then** done $:= 1$ **fi**;
    if even$(x) \vee$ done $= 1 \rightarrow$ **skip**
    $\Box$ odd$(x) \rightarrow L := \langle 2 * x, 2 * x + 1 \rangle +\!\!+ L$
    ... **fi od end.**

We can refine $x \xleftarrow{\text{pick}} L$ to pick an odd integer whenever one is available. If x is odd and done $= 0$ initially, then each iteration of the loop will remove one odd integer from L and add another one, without changing done. So L will never become empty and the loop will never terminate and is equivalent to **abort**. So $F_R(1) \leq$ **abort** in which case it must be equivalent to **abort**; since the only program which **abort** refines, is **abort**.

Theorem 7.1 has a simple corollary:

Corollary 7.2. If for all x and y, $F(x)$; $F(y) \approx F(y)$; $F(x)$ and there exists an irreflexive well-founded order $\prec$ on the domain of F such that $g_{jk}(x) \prec x$ for all x in the domain, and all $1 \leqslant j \leqslant N$ and $1 \leqslant k \leqslant n_j$, then:

proc $F_D(x) \equiv$
  var $L := \langle x \rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \xleftarrow{\text{pop}} L$; $\mathbf{S}_0$;
    $L := G(x) +\!\!+ L$ **od end.**
$\approx$
proc $F_R(x) \equiv$
  var $L := \langle x \rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \xleftarrow{\text{pick}} L$; $\mathbf{S}_0$;
    $L := G(x) +\!\!+ L$ **od end.**

Note that a "random-first" execution may well be refined to a "best-first" execution, where the element selected from L is chosen for efficiency, or some other reason. See Section 10 for an example. With our tree searching example, random-first execution maintains a "fringe" of the tree (a list of subtrees which have yet to be explored fully), an element on the fringe is selected and executed. If the element was not a leaf node then its daughters are added to the fringe.

7.3. Breadth First Execution

Another execution method is to append the new elements to the *end* of the sequence L (which therefore becomes a queue rather than a stack). This will explore all the nodes at a particular depth before moving to the nodes at the next depth.

proc $F_B(x) \equiv$
  var $L := \langle x \rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \xleftarrow{\text{pop}} L$; $\mathbf{S}_0$; $L := L +\!\!+ G(x)$ **od end.**

Our next theorem will show that the well-founded order is not required to prove $F_B(x) \leq F_D(x)$. In other words, we can refine any breadth-first execution of a commutative procedure to a depth first execution, without needing the well-foundedness constraint.

Theorem 7.2. If F and F_B are defined as above, and for all x and y: $F(x)$; $F(y) \approx F(y)$; $F(x)$, then $F_B(x) \leq F(x)$.

Proof: See Appendix.

This proves that $F_B(x) \leq F_D(x)$ under the commutativity condition. The next example shows that the two execution orders are not necessarily equivalent if the well-foundedness constraint is dropped. Define:

proc $F(x) \equiv$ **if** $d(x) > m$ **then** $m := d(x)$ **fi**;
  if $\exists y \in \mathbb{N}.\, x = 3 * y \wedge m > d(x) \rightarrow$ **skip**
  $\Box\ \exists y \in \mathbb{N}.\, x = 3 * y \wedge m \leqslant d(x) \rightarrow F(x+1);$
    $F(x+3)$
  $\Box\ \exists y \in \mathbb{N}.\, x = 3 * y + 1 \rightarrow F(x+1)$
  $\Box\ \exists y \in \mathbb{N}.\, x = 3 * y + 2 \rightarrow$ **skip fi.**
where
funct $d(x) \equiv \lfloor x/3 \rfloor + (x \bmod 3)$.

The function $d(x)$ measures the "depth" of node x. The global variable m records the depth of the deepest node seen so far. Suppose x is a multiple of three. For

such nodes, the first arm of the second **if** statement truncates the search if a "deeper" node has been seen previously. Depth-first execution always expands nodes $x+1$ and $x+2$ before node $x+3$ so, since $x+2$ is "deeper" than $x+3$, the $x+3$ node will not be expanded and the call to $F(x+3)$ will terminate. With breadth-first execution however, the $x+3$ node is always seen before the $x+2$ node and is therefore always expanded. The breadth-first execution therefore never terminates.

8. FIBONACCI NUMBERS

In this section we use cascade recursion removal to derive an efficient iterative algorithm from an inefficient recursive algorithm. Our example makes use of the implementation dependent execution order provided by a "random first" execution, in order to improve the efficiency of a recursive function. We illustrate the method by using the familiar Fibonacci numbers series.

The sequence of Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... is defined:

$$F_0 = 0, \qquad F_1 = 1, \qquad F_{n+2} = F_{n+1} + F_n$$

The specification of a function to calculate Fibonacci numbers is:

funct $f(n) \equiv F_n$.

Transform to take out special cases:

funct $f(n) \equiv$
  if $n = 0 \vee n = 1$ **then** n
    else $F_{n-1} + F_{n-2}$ **fi.**

Apply the theorem in [4] to get the recursive version:

funct $f(n) \equiv$
  if $n = 0 \vee n = 1$ **then** n
    else $f(n-1) + f(n-2)$ **fi.**

A procedural equivalent of this is the following, which returns the result in the global variable r:

proc $f(n) \equiv$
  if $n = 0 \vee n = 1$ **then** $r := n$
    else $f(n-1)$;
      var $r_1 := r$:
      $f(n-2)$; $r := r + r_1$ **end fi.**

To avoid the need for the local variable r_1 we use a different procedure which adds the result of the function to the value in r:

proc $F(n) \equiv$
  if $n = 0 \vee n = 1$ **then** $r := r + n$
    else $F(n-1)$; $F(n-2)$ **fi.**

so $F(n) \approx$ **var** $r_1 := r$: $f(n)$; $r := r + r_1$ **end** and $f(n) \approx r := 0$; $F(n)$. The proof of these equivalences uses the General Induction Rule for Recursion, Lemma 4.3.

The depth-first implementation of this recursion will be hopelessly inefficient, taking $O(2^n)$ steps to execute $F(n)$. The random-first execution will be equally inefficient: we aim to apply some efficiency-improving transformations to reach an $O(n)$ algorithm, without using any properties of F_n.

Clearly $F(x)$; $F(y) \approx r := r + F_x + F_y \approx r := r + F_y + F_x \approx F(y)$; $F(x)$. So the commutativity condition is satisfied. For positive integer arguments, "$<$" on $\mathbb{N}$ is a well-founded order with $n-1 < n$ and $n-2 < n$. So we can apply Theorem 7.1 to give a random-first implementation of the recursion:

proc $F_R(n) \equiv$
  var $L := \langle n \rangle$:
  while $L \neq \langle\rangle$ **do**
    $n \xleftarrow{\text{pick}} L$;
    if $n = 0 \vee n = 1$
      then $r := r + n$
      else $L := \langle n-1, n-2 \rangle +\!\!+ L$ **fi od end.**

Note that, in general, when the element $n-1$ in L is processed it will result in a second $n-2$ value appearing in L. So L is likely to end up with many copies of identical elements. We want to combine the effect of processing m copies of element x into processing a single copy of x and multiplying the result by m. First, we represent L as an array $L'[0\,..\,n_0]$ where for each i, $L'[i]$ is the number of occurrences of i in L.

With a small amount of restructuring we get:

proc $F_R(n) \equiv$
  var $L' := \langle\rangle$:
  $L'[0\,..\,n-1] := 0$; $L'[n] := 1$;
  while $\exists n.\, L'[n] > 0$ **do**
    $n := n'.(L'[n'] > 0)$;
    $L'[n] := L'[n] - 1$;
    if $n = 0$ **then skip**
    elsif $n = 1$
      then $r := r + n$
      else $L'[n-1] := L'[n-1] + 1$;
        $L'[n-2] := L'[n-2] + 1$ **fi od end.**

refine $n := n'.(L'[n'] > 0)$ to pick the largest value of n such that $L'[n] > 0$ (this refinement must result in an equivalent program, since the original specification is deterministic). Then restructure:

proc $F_R(n) \equiv$
  var $L' := \langle\rangle$:
  $L'[0\,..\,n-1] := 0$; $L'[n] := 1$;
  while $\exists n.\, L'[n] > 0$ **do**
    $n := \max\{\, i \mid L'[i] \neq 0 \,\}$;
    for $i := 1$ **to** $L'[n]$ **do**
      $L'[n] := L'[n] - 1$;
      if $n = 0 \vee n = 1$
        then $r := r + n$
        else $L'[n-1] := L'[n-1] + 1$;
          $L'[n-2] := L'[n-2] + 1$ **fi od od end.**

The **for** loop processes all the values in L equal to n. It sets $L'[n]$ to zero and carries out $L'[n]$ simple additions. So we can replace the **for** loop by the statement:

if $n = 0 \vee n = 1$ **then** $r := r + L'[n]$
else $L'[n-1] := L'[n-1] + L'[n]$;
$L'[n-2] := L'[n-2] + L'[n]$ **fi**;
$L'[n] := 0$

Split the cases $n = 0$ and $n = 1$, convert the **while** loop to a **do** loop, unroll on these cases using the fact that $n = \max\{\, i \mid L'[i] \neq 0 \,\}$:

proc $F_R(n) \equiv$
var $L' := \langle\rangle$:
$L'[0\,..\,n-1] := 0$; $L'[n] := 1$;
do if $\neg\exists n.\, L'[n] > 0$ **then exit fi**;
$n := \max\{\, i \mid L'[i] \neq 0 \,\}$;
if $n = 0$ **then** $L'[n] := 0$; **exit**
elsif $n = 1$
then $r := r + L'[n]$; $L'[n] := 0$;
if $L'[n-1] \neq 0$
then $n := n - 1$; $L'[n] := 0$ **fi**;
exit
else $L'[n-1] := L'[n-1] + L'[n]$;
$L'[n-2] := L'[n-2] + L'[n]$;
$L'[n] := 0$ **fi od end.**

At the end of the loop we will have $L'[n-1] > 0$ and $L'[m] = 0$ for all $m \geqslant n$ so the test $\exists n.\, L'[n] > 0$ is always true (since it is true before the loop). If we insert the statement $n := n + 1$ before the loop then we have $\max\{\, i \mid L'[i] \neq 0 \,\} = n + 1$ at the top of the loop, so we can transform $n := \max\{\, i \mid L'[i] \neq 0 \,\}$ to $n := n - 1$:

proc $F_R(n) \equiv$
var $L' := \langle\rangle$:
$L'[0\,..\,n-1] := 0$; $L'[n] := 1$;
$n := n + 1$
do $n := n - 1$;
if $n = 0$ **then** $L'[n] := 0$; **exit**
elsif $n = 1$
then $r := r + L'[n]$; $L'[n] := 0$;
if $L'[n-1] \neq 0$
then $n := n - 1$; $L'[n] := 0$ **fi**; **exit**
else $L'[n-1] := L'[n-1] + L'[n]$;
$L'[n-2] := L'[n-2] + L'[n]$;
$L'[n] := 0$ **fi od end.**

Convert to a **while** loop:

proc $F_R(n) \equiv$
var $L' := \langle\rangle$:
$L'[0\,..\,n-1] := 0$; $L'[n] := 1$;
while $n > 1$ **do**
$L'[n-1] := L'[n-1] + L'[n]$;
$L'[n-2] := L'[n-2] + L'[n]$;
$L'[n] := 0$;
$n := n - 1$ **od**;
if $n = 0$ **then** $L'[n] := 0$
elsif $n = 1$
then $r := r + L'[n]$; $L'[n] := 0$;
if $L'[n-1] \neq 0$
then $n := n - 1$; $L'[n] := 0$ **fi fi end.**

It is clear that only the top two elements of L' are non-zero, because at each stage we clear the top element and increment the two elements underneath it. Also, the assignment to $L'[n-2]$ is equivalent to $L'[n-2] := L'[n]$. So we add variables a and b with assignments such that $a = L'[n]$ and $b = L'[n-1]$. Then we can replace references to L' by references to a and b and remove L' from the program:

proc $F_R(n) \equiv$
var $a := 1, b := 0$:
while $n > 1$ **do**
$\langle a, b\rangle := \langle a + b, a\rangle$; $n := n - 1$ **od**;
if $n = 1$ **then** $r := r + a$ **fi end.**

Substitute in f and simplify to get:

funct $f(n) \equiv$
var $\langle a := 1, b := 0\rangle$:
while $n > 1$ **do**
$\langle a, b\rangle := \langle a + b, a\rangle$; $n := n - 1$ **od**;
if $n = 0$ **then** 0 **else** a **fi**$\rfloor$.

This *linear* factorial function has been derived by applying simple optimising transformations to the exponential, iterative factorial function obtained by removing the recursion in the definition of factorial.

9. TREE SEARCHING ALGORITHMS

In this section we consider recursion removal under different conditions, which relate to tree searching algorithms. Consider the following recursive procedure:

proc $F(x) \equiv$ **if B then** $\mathbf{S}_0$; $F * G(x)$ **fi.**

where the condition **B** does not depend on x and $G(x)$ is a pure function (i.e. a function with no side effects) which depends only on x. As above, $*$ is a "procedure map" operator: $F * G(x)$ means "call procedure F once for each element of $G(x)$, using the element as its argument". The condition **B** is called a "pruning condition", since as soon as it becomes false, no further processing is carried out. $F(x)$ is a typical depth-first search algorithm, Theorem 6.1 gives the following iterative equivalent:

proc $F_D(x) \equiv$
var $L := \langle x\rangle$:
while $L \neq \langle\rangle$ **do**
$x \xleftarrow{\text{pop}} L$;
if B then $\mathbf{S}_0$; $L := G(x) +\!\!+ L$ **fi od end.**

Instead of a commutativity condition on F we have a condition on **B** and $\mathbf{S}_0$: for all values x_1 and x_2:

var $x := x_1$: **if B then** $\mathbf{S}_0$ **fi end**;
var $x := x_2$: **if B then** $\mathbf{S}_0$ **fi end**
$\approx$
var $x := x_2$: **if B then** $\mathbf{S}_0$ **fi end**;
var $x := x_1$: **if B then** $\mathbf{S}_0$ **fi end**

Under these conditions we can prove the following:

THEOREM 9.1. $F(x) \leq F_B(x)$ where:

proc $F_B(x) \equiv$
  var $L := \langle x \rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \xleftarrow{\text{pop}} L$;
    if B then $\mathbf{S}_0$; $L := L +\!\!+ G(x)$ **fi od end.**

Proof: See Appendix ■

The converse does not hold in general, i.e. under the condition of Theorem 9.1 it is not always the case that $F_B(x) \leq F_D(x)$. Consider the following procedures:

proc $F_D(x) \equiv$ **if** done $= 0$
    then if even(x) **then** done $:= 1$ **fi**;
      $F * G(x)$ **fi.**

where
funct $G(x) \equiv \langle x, x+1 \rangle$.

and

proc $F_B(x) \equiv$
  var $L := \langle x \rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \xleftarrow{\text{pop}} L$;
    if done $= 1$
      then if even(x) **then** done $:= 1$ **fi**;
        $L := L +\!\!+ \langle x, x+1 \rangle$ **fi od end.**

If done $= 0$ initially then $F_D(1)$ never terminates (it leads to another call of $F_D(1)$) while $F_B(1)$ first sets $L := \langle 1, 2 \rangle$, then pops 1 off L and sets it to $\langle 2, 1, 2 \rangle$, then pops 2 off L, sets done $:= 1$ and sets L to $\langle 1, 2, 1, 2 \rangle$. Now done $= 1$ so all the elements are popped off L and the procedure terminates.

So in this situation we see that breadth-first execution is the more general form, in the sense that any depth first execution is refined by the corresponding breadth first execution, but the converse does not hold.

10. A HYBRID SORTING ALGORITHM

The following specification statement is a very concise specification for a program which sorts the array A:

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

where for any sequences s and t:

$$\mathsf{sorted}(s) =_{\text{DF}} \forall i.\, 1 \leqslant i < \ell(s).s[i] \leqslant s[i+1]$$

denotes that the sequence s is sorted, and

$$\mathsf{perm}(s,t) =_{\text{DF}} \exists \pi \in \mathsf{perms}(\ell(s)).\forall i.\ 1 \leqslant i \leqslant \ell(s).s[i] = t[\pi(i)]$$

denotes that the sequence t is a permutation of sequence s. The function perms(n) returns the set of *permutations* (bijections) mapping $\{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$.

Many sorting algorithms use a "divide and conquer" strategy which involves sorting part of an array. A trivial generalisation of the SORT specification treats sorting the array segment $A[a\,..\,b]$:

$$\mathsf{SORT}(a,b) =_{\text{DF}} A[a\,..\,b] := A'[a\,..\,b].(\mathsf{sorted}(A'[a\,..\,b]) \wedge \mathsf{perm}(A[a\,..\,b], A'[a\,..\,b]))$$

In [9] we refine this specification into an insertion sort and a quicksort (see [45]). Darlington [46] refines this specification into several different sorting algorithms. The recursive quicksort algorithm takes the following form:

proc QSORT$(a,b) \equiv$
  var $p := 0$:
  if $b > a$
    then partition;
      QSORT$(a, p-1)$; QSORT$(p+1, b)$ **fi end.**

where the partition procedure assigns a value to p such that $a \leqslant p \leqslant b$ and also permutes $A[a\,..\,b]$ such that $A[a\,..\,p-1] \leqslant A[p] \leqslant A[p+1\,..\,b]$, in other words, the elements of A before $A[p]$ are less than or equal to $A[p]$, while the elements after $A[p]$ are greater than or equal to $A[p]$. Sedgewick [47] suggests a suitable method for selecting p which he calls "median of three partitioning", in [9] we give a transformational refinement of the partition procedure, according to this method. Simple recursion removal leads to the following algorithm which is reasonably time-efficient:

proc QSORT$_1(a,b) \equiv$
  var $L := \langle\langle a, b \rangle\rangle, p := 0$:
  while $L \neq \langle\rangle$ **do**
    $\langle a, b \rangle \xleftarrow{\text{pop}} L$;
    if $b > a$
      then partition;
        $L := \langle\langle a, p-1 \rangle, \langle p+1, b \rangle\rangle +\!\!+ L$ **fi od end.**

This algorithm has one major drawback in that the stack L could require storage for up to $b-a$ pairs of elements. This requirement can be reduced to $\lceil \lg(b-a) \rceil$ pairs, provided the smaller of the two ranges $a\,..\,p-1$ and $p+1\,..\,b$ is treated first. (The proof of this assertion, which is quite simple, is given in [9]). This is clearly a case for a "best-first" execution order. QSORT(a,b) is clearly a commutative procedure, and the expression $b-a$ is reduced for each recursive call and never goes below -1. So "best-first" execution immediately gives the more efficient version:

proc QSORT$_2(a,b) \equiv$
  var $L := \langle\langle a, b \rangle\rangle, p := 0$:
  while $L \neq \langle\rangle$ **do**
    $\langle a, b \rangle \xleftarrow{\text{pop}} L$;
    if $b > a$

```
      then partition;
           if p − a < b − p
             then L := ⟨⟨a, p − 1⟩, ⟨p + 1, b⟩⟩ ++ L
              else L := ⟨⟨p + 1, b⟩, ⟨a, p − 1⟩⟩ ++ L fi
    fi od end.
```

A second source of inefficiency is that on any execution of the algorithm, the vast majority of segments being partitioned will be "small" (for the recursive version, most of the calls to QSORT will be for small segments). It is well known that for small segments, insertion sort is more efficient than quicksort (due to the reduced overhead)[2]. This would suggest that we use QSORT for "large" partitions and insertion sort for the smaller partitions, for example:

```
proc QSORT2(a, b) ≡
  var p := 0 :
  if b > a
    then partition;
         if p − a > K then QSORT(a, p − 1)
                      else ISORT(a, p − 1) fi;
         if b − p > K then QSORT(p + 1, b)
                      else ISORT(p + 1, b) fi fi end.
```

where K is a constant to be determined empirically, and ISORT is an implementation of insertion sort (see [9] for the derivation of ISORT). However, this leads to a large number of calls to ISORT. The same effect can be achieved more efficiently by leaving all the small segments unsorted, and then sorting them with a single call to ISORT(a, b).

To implement this idea, we again make use of the implementation dependence provided by random-first execution. We partition L into two variables L_1 which contains the large segments, and L_2 which contains the small segments. We also modify the algorithm to select from L_1 in preference to L_2 (but still put a smaller segment onto L_1 in front of a larger segment):

```
proc QSORT3(a, b) ≡
  var L1 := ⟨⟩, L2 := ⟨⟩, p := 0 :
  if b − a > K + 1
    then L1 := ⟨⟨a, b⟩⟩
     else L2 := ⟨⟨a, b⟩⟩ fi;
  while L1 ++ L2 ≠ ⟨⟩ do
    if L1 ≠ ⟨⟩ then ⟨a, b⟩ ←pop L1 else ⟨a, b⟩ ←pop L2 fi;
    if b > a
      then partition;
           if p − a < b − p
             then if p − a > K
                    then L1 := ⟨⟨a, p − 1⟩⟩ ++ L1
                     else L2 := ⟨⟨a, p − 1⟩⟩ ++ L2 fi;
                  if b − p > K
                    then L1 := ⟨⟨p + 1, b⟩⟩ ++ L1
                     else L2 := ⟨⟨p + 1, b⟩⟩ ++ L2 fi
              else if b − p > K
                    then L1 := ⟨⟨p + 1, b⟩⟩ ++ L1
                     else L2 := ⟨⟨p + 1, b⟩⟩ ++ L2 fi;
                  if p − a > K
                    then L1 := ⟨⟨a, p − 1⟩⟩ ++ L1
                     else L2 := ⟨⟨a, p − 1⟩⟩ ++ L2 fi
    fi fi od end.
```

The next step is to split the **while** loop into two loops. This uses the following transformation (proved in [3]). For any statement **S** and formulae $\mathbf{B}_1$ and $\mathbf{B}_2$ where $\mathbf{B}_1 \Rightarrow \mathbf{B}_2$:

$$\Delta \vdash \textbf{while } \mathbf{B}_2 \textbf{ do S od} \approx \textbf{while } \mathbf{B}_1 \textbf{ do S od; while } \mathbf{B}_2 \textbf{ do S od}$$

In this case $\mathbf{B}_1$ is $L_1 \neq \langle\rangle$ and $\mathbf{B}_2$ is $L_1 \mathbin{+\!\!+} L_2 \neq \langle\rangle$. After splitting the loop into two, we can simplify the loops by using the fact that each segment is replaced by *smaller* segments. The first loop terminates when $L_1 = \langle\rangle$, which means that the second loop will take elements from L_2, in which case any new elements will always be smaller, and hence will be inserted into L_2. So L_1 remains empty throughout the second loop and can be removed. The second loop becomes:

```
while L2 ≠ ⟨⟩ do
  ⟨a, b⟩ ←pop L2;
  if b > a
    then partition;
         if p − a < b − p
           then L2 := ⟨⟨a, p − 1⟩, ⟨p + 1, b⟩⟩ ++ L2
            else L2 := ⟨⟨p + 1, b⟩, ⟨a, p − 1⟩⟩ ++ L2
         fi fi od
```

This is equivalent to executing QSORT$_2$ on each pair in L_2 (see the proof of Theorem 7.1), which in turn is equivalent to executing ISORT on each pair in L_2 (since both QSORT$_2$ and ISORT are equivalent to SORT):

```
while L2 ≠ ⟨⟩ do
  ⟨a, b⟩ ←pop L2;
  if b > a
    then ISORT(a, b) fi od
```

The **while** loop reduces to a filter followed by a procedure map operation:

$$L_2 := \mathsf{filter}(L_2);\ \mathsf{ISORT} * L_2;\ L_2 := \langle\rangle$$

where filter removes all the pairs $\langle a, b\rangle$ with $b > a$.

Recall that throughout this development we have been working with a program which is provably equivalent to SORT(a_0, b_0) (where a_0 and b_0 are the initial values of a and b), so after the loop the array $A[a_0 \,..\, b_0]$ is sorted. Under this condition, SORT(a_0, b_0) is equivalent to **skip**, and hence so is ISORT(a_0, b_0). So we can insert a call to ISORT after the procedure map:

$$L_2 := \mathsf{filter}(L_2);\ \mathsf{ISORT} * L_2;\ L_2 := \langle\rangle;\ \mathsf{ISORT}(a_0, b_0)$$

[2] *Fancy algorithms are slow when n is small, and n is usually small.*—Rob Pike

For each pair $\langle a, b\rangle$ in the filtered list L_2 we have

$$a_0 \leqslant a \leqslant b \leqslant b_0$$

and under this condition

$$\mathsf{SORT}(a,b);\ \mathsf{SORT}(a_0,b_0) \ \approx\ \mathsf{SORT}(a_0,b_0)$$

The same is true for ISORT, so we can delete the procedure map operation:

$L_2 := \mathsf{filter}(L_2);\ L_2 := \langle\rangle;\ \mathsf{ISORT}(a_0,b_0)$

Now, the only references to L_2 are in statements which assign to L_2. So L_2 is a redundant variable and can be removed from the program. The whole of the second loop has been absorbed by the inserted call to ISORT. So the algorithm is:

```
proc QSORT4(a,b) ≡
  var L1 := ⟨⟩, a0 := a, b0 := b, p := 0 :
  if b − a > K + 1 then L1 := ⟨⟨a, b⟩⟩ fi;
  while L1 ≠ ⟨⟩ do
    ⟨a, b⟩ ←pop L1;
    if b > a
      then partition;
           if p − a < b − p
             then if p − a > K
                    then L1 := ⟨⟨a, p − 1⟩⟩ ++ L1 fi;
                  if b − p > K
                    then L1 := ⟨⟨p + 1, b⟩⟩ ++ L1 fi
              else if b − p > K
                    then L1 := ⟨⟨p + 1, b⟩⟩ ++ L1 fi;
                  if p − a > K
                    then L1 := ⟨⟨a, p − 1⟩⟩ ++ L1 fi
           fi fi od;
  ISORT(a0, b0) end.
```

The test $b > a$ is redundant since for every element in L_1 we have $b - a > K + 1 > 0$, so the program simplifies to:

```
proc QSORT5(a,b) ≡
  var L1 := ⟨⟩, a0 := a, b0 := b, p := 0 :
  if b − a > K + 1
    then do partition;
            if p − a < b − p
              then if p − a > K
                     then L1 := ⟨⟨a, p − 1⟩⟩ ++ L1 fi;
                   if b − p > K
                     then L1 := ⟨⟨p + 1, b⟩⟩ ++ L1 fi
               else if b − p > K
                     then L1 := ⟨⟨p + 1, b⟩⟩ ++ L1 fi;
                   if p − a > K
                     then L1 := ⟨⟨a, p − 1⟩⟩ ++ L1 fi fi;
            if L1 = ⟨⟩ then exit fi;
            ⟨a, b⟩ ←pop L1 od;
         ISORT(a0, b0) end.
```

11. PROGRAM ANALYSIS

The recursion removal theorem is a transformation which can be applied equally well in either direction. Because it places so few restrictions on the form of the program, there are many iterative programs which can be cast in the form required by the transformation. Hence it can be used as a program analysis and reverse engineering tool. It will make explicit the control structure of programs which use a stack in a particular way. For example, consider the following function:

```
funct A(m,n) ≡
  var ⟨d := 0, stack := ⟨⟩⟩:
  do do if m = 0 then n := n + 1; exit
        elsif n = 0 then stack := ⟨1⟩ ++ stack;
                         m := m − 1; n := 1
                    else stack := ⟨0⟩ ++ stack;
                         n := n − 1 fi od;
     do if stack = ⟨⟩ then exit(2) fi;
        d ← stack;
        if d = 0 then stack := ⟨1⟩ ++ stack;
                      m := m − 1; exit fi;
        m := m + 1 od od end;
  (n).
```

This program was analysed by the REDO team at the Programming Research Group in Oxford to test their proposed methods for formal reverse engineering of source code. Their paper [48] required eight pages of careful reasoning plus some "inspiration" to uncover the specification this short program. With the aid of our theorem the analysis breaks down into three steps:

1. Restructure into the right form for application of the theorem;
2. Apply the theorem;
3. Restructure the resulting recursive procedure in a functional form.

If we examine the operations carried out on the stack we see that the program terminates when the stack becomes empty, and when the stack is not empty, a value is popped off the stack and used to determine the control flow. Any program which carries out these sorts of operations on a stack is a suitable candidate for the recursion removal theorem applied in the reverse direction. The first step is to restructure the loops into an action system and collect together the "stack push" operations into separate actions. More correctly, we "deconstruct" the loops and **if** statements into separate actions. This is a simple, mechanical operation, similar to the construction of a flow graph of "basic blocks" carried out by many optimising compilers:

```
var d := 0, stack := ⟨⟩ :
actions A1 :
A1 ≡
  if m = 0 then n := n + 1; call Â
  elsif n = 0 then call B1
```

```
            else call B_2 fi.
B_1 ≡
  m := m − 1; n := 1;
  stack := ⟨1⟩ ++ stack; call A_1.
B_2 ≡
  n := n − 1; stack := ⟨0⟩ ++ stack; call A_1.
Â ≡
  if stack = ⟨⟩
    then call Z
     else d ← stack;
          if d = 0 then call B_3
                   else m := m + 1; call Â fi fi.
B_3 ≡
  m := m − 1; stack := ⟨1⟩ ++ stack; call A_1.
endactions end
```

Apply the transformation in Corollary (6.1) to get the recursive version:

```
proc F ≡
  actions A_1 :
  A_1 ≡
    if m = 0 then n := n + 1; call Z
    elsif n = 0 then call B_1
                else call B_2 fi.
  B_1 ≡
    m := m − 1; n := 1;
    F; m := m + 1; call Z.
  B_2 ≡
    n := n − 1; F; call B_3.
  B_3 ≡
    m := m − 1; F; m := m + 1; call Z.
  endactions
```

Unfold all the actions into A_1 to get:

```
proc F ≡ if m = 0 then n := n + 1
  elsif n = 0 then m := m − 1; n := 1;
                   F; m := m + 1
              else n := n − 1; F; m := m − 1;
                   F; m := m + 1 fi.
```

We can turn the global variables m and n into parameters if we add a variable r to record the final value of n (the value of m is unchanged):

```
var r := 0: F(n, m); n := r
where
proc F(m, n) ≡
  if m = 0 then r := n + 1
  elsif n = 0 then F(m − 1, 1)
              else F(m, n − 1); F(m − 1, r) fi. end
```

This procedure can be written in a functional form and substituted into the original function to get:

```
funct A(m, n) ≡
  if m = 0 then n + 1
  elsif n = 0 then A(m − 1, 1)
              else A(m − 1, A(m, n − 1)) fi.
```

where we have replaced calls to F by equivalent calls to A (an example of alpha-conversion).
This is the famous Ackermann function [49].

12. A LARGER REVERSE ENGINEERING EXAMPLE

Consider the following program which takes four positive integer arrays l, r, c and m and a positive integer x and modifies some of the elements of m, where m is assumed to be initially an array of zeros:

```
var ⟨L := ⟨⟩, d := 0⟩ :
proc F(x) ≡
  do do if x ≠ 0
          then if m[l[x]] = 0
                 then L ⟵push ⟨1, x⟩; x := l[x]
                  else exit fi
           else do if L = ⟨⟩ then exit(3) fi;
                   ⟨d, x⟩ ⟵pop L;
                   if d = 0 → exit(2)
                   □ d = 1 → exit
                   □ d = 2 → skip fi od fi;
         do m[x] := 1;
            if m[c[x]] = 0 ∧ m[r[x]] = 1
              then L ⟵push ⟨2, x⟩; x := c[x]; exit(2)
            elsif m[c[x]] = 1 ∧ m[r[x]] = 0
                then L ⟵push ⟨2, x⟩; x := r[x]; exit(2)
            elsif m[c[x]] = 0 ∧ m[r[x]] = 0
                then L ⟵push ⟨2, x⟩; L ⟵push ⟨0, r[x]⟩;
                     x := c[x]; exit(2)
                 else do if L = ⟨⟩ then exit(4) fi;
                         ⟨d, x⟩ ⟵pop L;
                         if d = 0 → exit(3)
                         □ d = 1 → exit
                         □ d = 2 → skip fi
                      od fi od od od. end
```

Despite its small size, this program has a fairly complex control structure, with a quadruple-nested loop and exits which terminate from the middle of one, two, three and four nested loops. Finding suitable invariants for all the loops seems to be a difficult task, while finding a variant function is impossible! For suppose $x \neq 0$, $m[l[x]] = 0$ and $l[x] = x$ initially. With this initial state, it is easy to see that the program will never terminate. Therefore there is *no* variant function which works over the whole initial state space. To determine the conditions under which the program terminates, basically involves determining the behaviour of the entire program: not a helpful requirement for the first stage of a reverse engineering task!

There are certain features which suggest that the recursion removal theorem might be usefully applied: in particular the presence of a local array which is used as a stack and which starts empty and finishes empty. One problem is that there are two places in the program where L is tested and an element popped off. However,

if we restructure the program as an action system, then there are some powerful transformations which can be used to merge the two actions which test L and convert the program into the right structure for the recursion introduction theorem.

The first step therefore is to restructure the program as an action system. This basically involves implementing the loops and **exit**s as action calls (i.e. **goto**s)—an unusual step in a reverse engineering process!

```
var ⟨L := ⟨⟩, d := 0⟩ :
proc F(x) ≡
  actions A1 :
  A1 ≡
    if x ≠ 0 then if m[l[x]] = 0 then call B1
                                 else call A2 fi
            else call F̂1 fi.
  A2 ≡
    m[x] := 1;
    if m[c[x]] = 0 ∧ m[r[x]] = 1
      then call B2
    elsif m[c[x]] = 1 ∧ m[r[x]] = 0
        then call B3
    elsif m[c[x]] = 0 ∧ m[r[x]] = 0
        then call B4
         else call F̂2 fi.
  B1 ≡
    L ⟵push ⟨1, x⟩; x := l[x]; call A1.
  B2 ≡
    L ⟵push ⟨2, x⟩; x := c[x]; call A1.
  B3 ≡
    L ⟵push ⟨2, x⟩; x := r[x]; call A1.
  B4 ≡
    L ⟵push ⟨2, x⟩; L ⟵push ⟨0, r[x]⟩; x := c[x]; call A1.
  F̂1 ≡
    if L = ⟨⟩ then call Z fi;
    ⟨d, x⟩ ⟵pop L;
    if d = 0 → call A1
    □ d = 1 → call A2
    □ d = 2 → call F̂1 fi.
  F̂2 ≡
    if L = ⟨⟩ then call Z fi;
    ⟨d, x⟩ ⟵pop L;
    if d = 0 → call A1
    □ d = 1 → call A2
    □ d = 2 → call F̂2 fi. endactions. end
```

The actions $\hat{F}_1$ and $\hat{F}_2$ are identical (apart from calls to $\hat{F}_1$ and $\hat{F}_2$) so they can be merged using a transformation in [3]. The result will be in the right form for Corollary 6.1. (Incidentally, the resulting action call graph is irreducible, but this causes no difficulties for our methods). Applying Corollary 6.1 gives:

```
proc F(x) ≡
  actions A1 :
  A1 ≡
    if x ≠ 0 then if m[l[x]] = 0 then call B1
                                 else call A2 fi
            else call Z fi.
  A2 ≡
    m[x] := 1;
    if m[c[x]] = 0 ∧ m[r[x]] = 1
      then call B2
    elsif m[c[x]] = 1 ∧ m[r[x]] = 0
        then call B3
    elsif m[c[x]] = 0 ∧ m[r[x]] = 0
        then call B4
         else call Z fi.
  B1 ≡
    F(l[x]); call A2.
  B2 ≡
    F(c[x]); call Z.
  B3 ≡
    F(r[x]); call Z.
  B4 ≡
    F(c[x]); F(r[x]); call Z. endactions.
```

Restructuring to remove the action system to give a recursive version of the program:

```
proc F(x) ≡
  if x ≠ 0
    then if m[l[x]] = 0 then F(l[x]) fi;
         m[x] := 1;
         if m[c[x]] = 0 ∨ m[r[x]] = 1
           then F(c[x])
         elsif m[c[x]] = 1 ∨ m[r[x]] = 0
             then F(r[x])
         elsif m[c[x]] = 0 ∨ m[r[x]] = 0
             then F(c[x]); F(r[x]) fi fi.
```

We have reduced the 22 line original program, with its quadruple-nested loops, to a ten line recursive procedure with no loops and only simple **if** statements, by a purely mechanical application of general purpose transformations.

12.1. Change Data Representation

Observe that the arrays $l[]$, $c[]$ and $r[]$ are never modified, while array $m[]$ starts out with all zeros and during the course of the program's execution, certain elements of $m[]$ are set to 1. Since x is a parameter of the procedure, the only effect of the procedure is to set these elements of $m[]$: we want to determine which elements of $m[]$ are modified. Let the set $\mathcal{N}$ represent the domain of $F()$ and of the arrays l, c, r and m. The array $m[]$ is equivalent to a subset of $\mathcal{N}$ (those elements x for which $m[x] = 1$). We define the set M of *marked elements* as follows:

$$M =_{\text{DF}} \{\, x \in \mathcal{N} \mid m[x] = 1 \,\}$$

For each $x \in \mathcal{N}$, either $x = 0$ and the function returns immediately, or there are three other elements of $\mathcal{N}$ which may be used as parameters of F: the values

$l[x]$, $c[x]$ and $r[x]$. We therefore define a function $D : \mathcal{N} \to \mathcal{N}^*$ as follows:

$$D(x) =_{\text{DF}} \begin{cases} \langle\rangle & \text{if } x = 0 \\ \langle l[x], c[x], r[c]\rangle & \text{otherwise} \end{cases}$$

With this data representation our program becomes:

proc $F(x)$ $\equiv$
  if $D(x) \neq \langle\rangle$
    then if $D(x)[1] \notin M$ **then** $F(D(x)[1])$ **fi**;
      $M := M \cup \{x\}$;
      if $D(x)[2] \notin M \ \wedge \ D(x)[3] \in M$
        then $F(D(x)[2])$
      elsif $D(x)[2] \in M \ \wedge \ D(x)[3] \notin M$
        then $F(D(x)[3])$
      elsif $D(x)[2] \notin M \ \wedge \ D(x)[3] \notin M$
        then $F(D(x)[2])$; $F(D(x)[3])$ **fi fi.**

We can simplify the **if** statement by using an iteration over a sequence (see [41] for the formal definition):

proc $F(x)$ $\equiv$
  if $D(x) \neq \langle\rangle$
    then if $D(x)[1] \notin M$ **then** $F(D(x)[1])$ **fi**;
      $M := M \cup \{x\}$;
      for $y \xleftarrow{\text{POP}} D(x)[2..] \setminus M$ **do** $F(y)$ **od fi.**

where for a sequence X and set M, the expression $X \setminus M$ denotes the subsequence of elements of X which are not in M.

This version of the program can be generalised for *any* function $D : \mathcal{N} \to \mathcal{N}^*$, so from now on we will ignore the "trinary" nature of the original D function. This step is an "abstraction" in the sense that our original program will implement a special case of the specification we derive.

With the recursive and abstract version of the program it is clear that the effect of a call to $F(x)$ is to add certain elements to the set M. Since all the recursive calls to $F(x)$ ensure that $x \notin M$, we will assume that this is the case for external calls also. Hence we can assume that the assertion $x \notin M$ holds at the beginning of the body of F. The arguments for the recursive calls are all elements of $D(x)$, so all the elements added to M will be reached by zero or more applications of D. In fact, the function D defines a *directed graph* on $\mathcal{N}$ with edges $\langle x, y\rangle$ where $y \in D(x)$. For any set $X \subseteq \mathcal{N}$ we define $R(X)$ to be the set of nodes in the graph *reachable* from X via zero or more applications of D. This is called the *Transitive Closure* of D:

$$R(X) =_{\text{DF}} \bigcup_{n<\omega} R^n(X)$$

where

$$R^0(X) =_{\text{DF}} X$$

and

$$R^{n+1}(X) =_{\text{DF}} \bigcup \{\, \mathsf{set}(D(y)) \mid y \in R^n(X) \,\}$$

We are also interested in the nodes reachable via unmarked nodes. We define $D_M(x) =_{\text{DF}} D(x) \setminus M$ which is the sequence $D(x)$ with elements of M deleted. We extend D_M to its transitive closure R_M in the same way as for D and R.

12.2. Abstraction Assumptions

To simplify the abstraction process we make two assumptions. The first is that all unmarked reachable nodes are reachable via unmarked nodes, i.e. $M \cup R(X) = M \cup R_M(X)$ initially for all $X \subseteq \mathcal{N}$. Since $M = \varnothing$ initially for our original program, this assumption is in fact a further generalisation of that program. Our second assumption is that no unmarked node is reachable from its first daughter node, i.e. $\forall x \in \mathcal{N} \setminus M.\ x \notin R(D(x)[1])$. This is an essential assumption since if $x \in R(D(x)[1])$ and $x \notin M$, then $x \in R_M(D(x)[1])$ and it is easy to see that $F(x)$ will not terminate.

In [8] we prove the following *Reachability Theorem*:

Theorem 12.1. Let M and X be sets of nodes such that $M \cup R(X) = M \cup R_M(X)$ and let $x \in X \setminus M$. Let A and B be any subsets of $R_M(\{x\})$ such that $R_M(\{x\}) \setminus A \subseteq R_{M \cup A}(B)$. Then:

$$\begin{aligned} M \cup R_M(X) &= M \cup A \cup R_{M \cup A}((X \setminus \{x\}) \cup B) \\ &= M \cup A \cup R((X \setminus \{x\}) \cup B) \end{aligned}$$

Two obvious choices for A are $\{x\}$ and $R_M(\{x\})$. In the former case, a suitable choice for B is $D(x) \setminus (M \cup \{x\})$ and in the latter case, the only choice for B is $\varnothing$. So we have two corollaries:

Corollary 12.1. If $M \cup R(X) = M \cup R_M(X)$ and $x \in X \setminus M$ then:

$$\begin{aligned} M \cup R_M(X) &= M \cup \{x\} \cup R_{M \cup \{x\}}(X') \\ &= M \cup \{x\} \cup R(X') \end{aligned}$$

where $X' = (X \setminus \{x\}) \cup (D(x) \setminus (M \cup \{x\}))$.

Corollary 12.2. If $M \cup R(X) = M \cup R_M(X)$ and $x \in X \setminus M$ then:

$$\begin{aligned} M \cup R_M(X) &= M \cup R(\{x\}) \cup R_{M \cup R(\{x\})}(X \setminus \{x\}) \\ &= M \cup R(\{x\}) \cup R(X \setminus \{x\}) \end{aligned}$$

12.3. The Specification

We claim that $F(x)$ is a refinement of $\mathsf{SPEC}(\{x\})$ where for $X \subseteq \mathcal{N}$:

$$\mathsf{SPEC}(X) =_{\text{DF}} I(X);\ M := M \cup R(X)$$

where

$$I(X) = \{M \cup R(X) = M \cup R_M(X)\}$$

To prove the claim, we will use the recursive implementation theorem Theorem 5.1. First, we replace the

recursive calls in $F(x)$ by copies of the specification and add an assertion (from the abstraction assumptions):

$\mathbf{S} = I(\{x\});$
  if $D(x) \neq \langle\rangle$
    then if $D(x)[1] \notin M$ **then** $\mathsf{SPEC}(D(x)[1])$ **fi**;
      $M := M \cup \{x\}$;
      for $y \xleftarrow{\text{pop}} D(x)[2..] \setminus M$ **do**
        $\mathsf{SPEC}(y)$ **od**
    else $M := M \cup \{x\}$ **fi**

This expands to:

$\mathbf{S} \approx I(\{x\});$
  if $D(x) \neq \langle\rangle$
    then if $D(x)[1] \notin M$
        then $I(\{D(x)[1]\})$;
          $M := M \cup R(\{D(x)[1]\})$ **fi**;
      $M := M \cup \{x\}$;
      for $y \xleftarrow{\text{pop}} D(x)[2..] \setminus M$ **do**
        $I(\{y\})$; $M := M \cup R(\{y\})$ **od**
    else $M := M \cup \{x\}$ **fi**

If $D(x)[1] \notin M$ then $D(x)[1] \notin (M \cup \{x\})$ since our abstraction assumption $x \notin R(D(x)[1])$ implies $x \neq D(x)[1]$, so adding x to M does not affect the test. So the assignment $M := M \cup \{x\}$ can be moved back past the preceding **if** statement:

$\mathbf{S} \approx I(\{x\});$
  if $D(x) \neq \langle\rangle$
    then $M := M \cup \{x\}$;
      if $D(x)[1] \notin M \setminus \{x\}$
        then $I(\{D(x)[1]\})$;
          $M := M \cup R(\{D(x)[1]\})$ **fi**;
      for $y \xleftarrow{\text{pop}} D(x)[2..] \setminus M$ **do**
        $I(\{y\})$; $M := M \cup R(\{y\})$ **od**
    else $M := M \cup \{x\}$ **fi**

Now we roll the **if** statement into the **for** loop and factor $M := M \cup \{x\}$ out of the outer **if** statement. The test $D(x) \neq \langle\rangle$ then becomes redundant since **for** $y \xleftarrow{\text{pop}} D(x)$ **do** ... **od** is equivalent to **skip** when $D(x) = \langle\rangle$.

$\mathbf{S} \approx I(\{x\});$
  $M := M \cup \{x\}$;
  for $y \xleftarrow{\text{pop}} D(x) \setminus M$ **do**
    $I(\{y\})$; $M := M \cup R(\{y\})$ **od**

By Corollary 12.2 and the general induction rule for iteration we can prove that for any $X \subseteq \mathcal{N}$ such that $M \cup R(X) = M \cup R_M(X)$:

for $y \in X \setminus M$ **do**
  $I(\{y\})$; $M := M \cup R(\{y\})$ **od**
$\approx$
$M := M \cup R_M(X)$

So we have:

$\mathbf{S} \approx I(\{x\})$; $M := M \cup \{x\}$; $M := M \cup R_M(D(x))$
  $\approx I(\{x\})$; $M := M \cup \{x\} \cup R_M(D(x))$

So by Corollary 12.1:

$$\mathbf{S} \approx \mathsf{SPEC}(\{x\})$$

Finally, note that before each copy of $\mathsf{SPEC}(\{x\})$ in **S**, *either* M has been increased (and hence the finite set $\mathcal{N} \setminus M$ reduced) from its initial value, *or* M remains the same, but $R(\{x\})$ has been reduced (the abstraction assumption that $x \notin R(D(x)[1])$ shows that $R(D(x)[1]) \subset R(\{x\})$. So we can apply the recursive implementation theorem (Theorem 5.1) in reverse to prove:

$\mathsf{SPEC}(\{x\}) \approx$
proc $F(x) \equiv$
  if $D(x) \neq \langle\rangle$
    then if $D(x)[1] \notin M$ **then** $F(D(x)[1])$ **fi**;
      $M := M \cup \{x\}$;
      for $y \xleftarrow{\text{pop}} D(x)[2..] \setminus M$ **do** $F(y)$ **od fi.**

So, given the abstraction assumptions of Section 12.2, our original program is a correct implementation of $\mathsf{SPEC}(\{x\})$.

13. CONCLUSION

In this paper we have briefly described our approach to algorithm derivation and reverse engineering, which relies on formal transformations in a wide-spectrum language, based in infinitary first order logic. We presented a general-purpose recursion removal/introduction theorem and, for the case of commutative procedures, we proved the following results:

- For a cascade recursion, any breadth first execution is refined by the corresponding depth-first execution, but the converse is not generally true;
- For a cascade recursion with a pruning condition (eg. a tree searching algorithm), any depth-first execution is refined by a breadth-first execution, but the converse is not generally true;
- If a well-founded order exists on the domain, such that all recursive calls have smaller arguments, then depth-first and breadth-first executions are equivalent, and equivalent to random-first execution.

We illustrated the practical application of these results with several algorithm derivations, and some sample reverse-engineering problems. The transformation approach has provided some powerful tools for algorithm derivation and reverse engineering.

14. APPENDIX: PROOFS OF VARIOUS THEOREMS

14.1. Theorem 7.1

To prove Theorem 7.1, we actually need a well-founded order on *sequences* of domain elements (i.e. on values of L rather than values of x). Let $\mathcal{D}$ be the domain set

and $\mathcal{D}^*$ be the set of finite sequences of elements of $\mathcal{D}$. The next lemma shows that a well-founded order on $\mathcal{D}$ induces a suitable order on $\mathcal{D}^*$.

DEFINITION 14.1. The extension $\prec^*$ of $\prec$ to $\mathcal{D}^*$ is defined as follows: For all $K, L \in \mathcal{D}^*$ we define $L \prec^* K$ to be true iff L is not a permutation of K, and L is of the form $L_1 +\!\!+ L_2 +\!\!+ \cdots +\!\!+ L_n$ where $n = \ell(K)$ and for each $i \in \{1, 2, \ldots, n\}$, either $L_i = \langle K[\pi(i)] \rangle$ or $\forall j, 1 \leqslant j \leqslant \ell(L_i).\, L_i[j] \prec K[\pi(i)]$. (Here π is some permutation of $\{1, 2, \ldots, n\}$ which depends on K and L). In other words, each element of K either appears somewhere in L, or is represented by a sequence of strictly smaller elements. Note that some or all of the L_i may be empty.

LEMMA 14.2. If $\prec$ is a well-founded partial order on $\mathcal{D}$ then $\prec^*$ as defined above, is a well-founded partial order on $\mathcal{D}^*$.

Proof: It is easy to see that $\prec^*$ is a partial order. To prove well-foundedness, suppose for contradiction that we have an infinite descending chain $L_1 \succ^* L_2 \succ^* L_3 \succ^* \ldots$ of elements of $\mathcal{D}^*$. From this chain we construct a tree $\mathfrak{T}_k$ of elements of $\mathcal{D}$ where the root of each $\mathfrak{T}_k$ is x_k, the kth element of L_1. (L_1 cannot of course be empty, since the empty sequence is smaller than any other). The daughters of x_k (if any) are those elements of L_2 which are equal to or smaller than x_k. At each level i in the tree $\mathfrak{T}_k$, the nodes are the elements of the corresponding L_i. From the fact that $L_i \succ^* L_{i+1}$ we have $L_{i+1} = L_{i1} +\!\!+ L_{i2} +\!\!+ \cdots +\!\!+ L_{in}$ where $\ell(L_{ij}) \geqslant 1$ and either $L_{ij} = \langle L_i[\pi(j)] \rangle$ or each element of L_{ij} is strictly less than $L_i[\pi(j)]$. As above, π is a permutation of $\{1, 2, \ldots, \ell(L_i)\}$ which depends on L_i and L_{i+1}. We define the daughters of $L_i[\pi(j)]$ in $\mathfrak{T}_k$ to be the elements of L_{ij}. So each element in $\mathfrak{T}_k$ either has one daughter (which is equal to it), or a list of daughters, each of which is strictly smaller than it.

We want to get an infinite descending chain in $\mathcal{D}$ from one of the trees, so we construct a new tree $\mathfrak{T}'_k$ from $\mathfrak{T}_k$ by pruning all the branches which end in an infinite sequence of identical elements. Some of the trees $\mathfrak{T}_k$ may be finite, but at least one of the trees must be infinite, since for each i, each element of L_i must appear in one of the trees. Therefore, at least one of the pruned trees $\mathfrak{T}'_k$ must also be infinite, since otherwise if all the trees were finite, say of maximum depth N, then the list of daughters at level N of $\mathfrak{T}$ is the same as at level $N + 1$, which contradicts $L_N \succ^* L_{N+1}$.

Let $\mathfrak{T}'$ be one of the infinite pruned trees and apply Koenig's Lemma to this finitely branching infinite tree to get an infinite path of elements $\langle x_0, x_1, \ldots \rangle$, where each element is less than or equal to the previous one. There must be an infinite number of $\succ$ relationships in this sequence since otherwise, at some point in the sequence all the elements are equal, but we pruned all such branches from the tree. Hence we have an infinite descending chain $x_0 \succ x_{n_1} \succ x_{n_2} \succ \ldots$ in $\mathcal{D}$ where for each i, $n_i < n_{i+1}$. This contradicts the well-foundedness of $\prec$. So the assumption that $\prec^*$ has an infinite descending chain is false. So $\prec^*$ is well founded. ■

The proof of the main theorem uses well-founded induction on the value of L using the well-founded order $\prec^*$ induced from $\prec$.

Proof: To prove Theorem 7.1, let:

$$\begin{aligned}\mathbf{DO} = \ &\textbf{while } L \neq \langle\rangle \textbf{ do}\\ &x \xleftarrow{\text{pop}} L;\ F(x) \textbf{ od}\end{aligned}$$

and

$$\begin{aligned}\mathbf{DO'} = \ &\textbf{while } L \neq \langle\rangle \textbf{ do}\\ &x \xleftarrow{\text{pick}} L;\ \mathbf{S}_0;\\ &L := G(x) +\!\!+ L \textbf{ od}\end{aligned}$$

So it is sufficient to prove **DO** $\approx$ **DO'** (ignoring the final value of x), since $F(x) \approx$ **var** $L := \langle x \rangle$: **DO end**. The proof is by well-founded induction on the value of L using the well-founded order $\prec^*$ induced from $\prec$.

The minimal element is clearly the empty sequence (since a sequence can always be made smaller by removing elements), and if $L = \langle\rangle$ then both **DO** and **DO'** are **skip**. So suppose **DO** $\approx$ **DO'** for all $L \prec \lambda$ for some $\lambda \neq \langle\rangle$ and let $L = \lambda$ initially. Unroll the first step of **DO'**:

$$\begin{aligned}\mathbf{DO'} \ \approx \ &x \xleftarrow{\text{pick}} L;\ \mathbf{S}_0;\\ &L := G(x) +\!\!+ L;\\ &\mathbf{DO'}\end{aligned}$$

This removes an element from L and adds a sequence of elements $\langle g_{j1}(x), g_{j2}(x), \ldots, g_{jn_j}(x) \rangle$ where x was the element removed. Since $g_{ij}(x) \prec x$ the value of L just before **DO'** is less than λ in the $\prec^*$ order. So we can apply the induction hypothesis:

$$\begin{aligned}\mathbf{DO'} \ \approx \ &x \xleftarrow{\text{pick}} L;\ \mathbf{S}_0;\\ &L := G(x) +\!\!+ L;\\ &\mathbf{DO}\end{aligned}$$

Unfold $G(x)$ and absorb **DO** into the **if** statement:

$$\begin{aligned}\mathbf{DO'} \ \approx \ &x \xleftarrow{\text{pick}} L;\ \mathbf{S}_0;\\ &\textbf{if } \mathbf{B}_0 \to \textbf{skip};\ \mathbf{DO}\\ &\square \ldots \square\ \mathbf{B}_j \to L := \langle g_{j1}(x), g_{j2}(x),\\ &\qquad\qquad \ldots, g_{jn_j}(x) \rangle +\!\!+ L;\ \mathbf{DO}\\ &\ldots \textbf{ fi}\end{aligned}$$

Now:

$$\begin{aligned}L := \ &\langle g_{j1}(x), g_{j2}(x), \ldots, g_{jn_j}(x) \rangle +\!\!+ L;\ \mathbf{DO}\\ \approx \ &L \xleftarrow{\text{push}} g_{jn_j}(x);\ \ldots;\ L \xleftarrow{\text{push}} g_{j2}(x);\\ &L \xleftarrow{\text{push}} g_{j1}(x);\ \mathbf{DO}\\ \approx \ &L \xleftarrow{\text{push}} g_{jn_j}(x);\ \ldots;\ L \xleftarrow{\text{push}} g_{j2}(x);\\ &x := g_{j1}(x);\ F(x);\ \mathbf{DO}\end{aligned}$$

by unrolling the first step of **DO**

$$\approx\ L \overset{\text{push}}{\longleftarrow} g_{jn_j}(x);\ \ldots;\ L \overset{\text{push}}{\longleftarrow} g_{j2}(x);\ F(g_{j1}(x));\ \textbf{DO}$$

since **DO** overwrites x before reading it

$$\approx\ F(g_1(x));\ L \overset{\text{push}}{\longleftarrow} g_{jn_j}(x);\ \ldots;\ L \overset{\text{push}}{\longleftarrow} g_{j2}(x);\ \textbf{DO}$$

since $F(g_{j1}(x)$ doesn't use x or L. Repeating these steps gives:

$$\approx\ F(g_{j1}(x));\ F(g_{j2}(x));\ \ldots;\ F(g_{jn_j}(x));\ \textbf{DO}$$

So we have:

$$\begin{aligned}\textbf{DO}'\ \approx\ & x \overset{\text{pick}}{\longleftarrow} L;\ \mathbf{S}_0;\\ & \textbf{if } \mathbf{B}_0 \to \textbf{skip};\ \textbf{DO}\\ & \square \ldots \square\ \mathbf{B}_j \to F(g_{j1}(x));\ F(g_{j2}(x));\\ & \qquad\qquad \ldots;\ F(g_{jn_j}(x));\ \textbf{DO}\\ & \ldots \textbf{ fi}\end{aligned}$$

Separate **DO** from the **if** statement again and use the fact that $\mathbf{S}_0;\ F * G(x)\ \approx\ F(x)$ to get:

$$\textbf{DO}'\ \approx\ x \overset{\text{pick}}{\longleftarrow} L;\ F(x);\ \textbf{DO}$$

We can express $x \overset{\text{pick}}{\longleftarrow} L$ as a nondeterministic choice (depending on which element is picked):

$$x \overset{\text{pick}}{\longleftarrow} L\ \approx\ \prod_{i=1}^{\ell(L)} \big(x := L[i];\ L := L[1\,..\,i-1] +\!\!+ L[i+1\,..]\big)$$

this is a nondeterministic choice over $\ell(L)$ statements: i is replaced by the integers 1, 2, 3, etc. in each of the statements. We claim that for each i:

$$\begin{aligned}\textbf{DO}\ \approx\ & x := L[i];\ L := L[1\,..\,i-1] +\!\!+ L[i+1\,..];\\ & F(x);\ \textbf{DO}\end{aligned}$$

First, re-write **DO** as a pair of **for** loops, and delay updating L until the last moment:

$x := L[i];$
$L := L[1\,..\,i-1] +\!\!+ L[i+1\,..];$
$F(x);$
DO
$\approx$
$x := L[i];\ F(x);$
for $j := 1$ **to** $i-1$ **do** $F(L[j])$ **od**;
for $j := i+1$ **to** $\ell(L)$ **do** $F(L[j])$ **od**;
$L := \langle\rangle$

Use the fact that $F(x);\ F(y)\ \approx\ F(y);\ F(x)$ and induction on i to move the $F(L[i])$ past the first **for** loop, and delete the assignment to x (since x is not used):

for $j := 1$ **to** $i-1$ **do** $F(L[j])$ **od**;
$F(L[i]);$
for $j := i+1$ **to** $\ell(L)$ **do** $F(L[j])$ **od**;
$L := \langle\rangle$

Now $F(L[i])\ \approx$ **for** $j := i$ **to** i **do** $F(L[j])$ **od** so we can merge the three **for** loops to get

for $j := 1$ **to** $\ell(L)$ **do** $F(L[j])$ **od**; $L := \langle\rangle$

which is equivalent to **DO** as required. ■

14.2. Theorem 7.2

proc $F_B(x)\ \equiv$
  var $L := \langle x\rangle$:
  while $L \neq \langle\rangle$ **do**
    $x \overset{\text{pop}}{\longleftarrow} L;\ \mathbf{S}_0;\ L := L +\!\!+ G(x)$ **od end.**

We have that for all x and y: $F(x);\ F(y)\ \approx\ F(y);\ F(x)$, and claim that $F_B(x)\ \leq\ F(x)$. Let:

DO =**while** $L \neq \langle\rangle$ **do**
    $x \overset{\text{pop}}{\longleftarrow} L;\ F(x)$ **od**
and
$\textbf{DO}'$ =**while** $L \neq \langle\rangle$ **do**
    $x \overset{\text{pop}}{\longleftarrow} L;\ \mathbf{S}_0;\ L := L +\!\!+ G(x)$ **od**

It is sufficient to show that $\textbf{DO}' \leq \textbf{DO}$. We will use induction on n and to prove that $\textbf{DO}'^n \leq \textbf{DO}$ and appeal to the induction rule for iteration. Here

$$\textbf{DO}'^0\ =_{\text{DF}}\ \textbf{abort}$$

and

$$\begin{aligned}\textbf{DO}'^{n+1}\ =_{\text{DF}}\ & \textbf{if } L \neq \langle\rangle\\ & \quad \textbf{then } x \overset{\text{pop}}{\longleftarrow} L;\ \mathbf{S}_0;\\ & \qquad L := L +\!\!+ G(x);\ \textbf{DO}'^n\ \textbf{fi}\end{aligned}$$

So suppose $\textbf{DO}'^n \leq \textbf{DO}$. We assume $L \neq \langle\rangle$ since the result is trivial when $L = \langle\rangle$:

$$\begin{aligned}\textbf{DO}'^{n+1}\ \approx\ & x \overset{\text{pop}}{\longleftarrow} L;\ \mathbf{S}_0;\ L := L +\!\!+ G(x);\ \textbf{DO}'^n\\ \leq\ & x \overset{\text{pop}}{\longleftarrow} L;\ \mathbf{S}_0;\ L := L +\!\!+ G(x);\ \textbf{DO}\end{aligned}$$

by the induction hypothesis. Unfold $G(x)$ and push **DO** inside the **if** statement:

$$\begin{aligned}\approx\ & x \overset{\text{pop}}{\longleftarrow} L;\ \mathbf{S}_0;\\ & \textbf{if } \mathbf{B}_0 \to L := L +\!\!+ \langle\rangle;\ \textbf{DO}\\ & \square \ldots \square\ \mathbf{B}_j \to L := L +\!\!+ \langle g_{j1}(x), g_{j2}(x),\\ & \qquad\qquad \ldots, g_{jn_j}(x)\rangle;\\ & \qquad\qquad \textbf{DO}\\ & \ldots \textbf{ fi}\end{aligned}$$

Note that $\textbf{DO}\ \approx\ F * L;\ L := \langle\rangle$ so we have:

$$\begin{aligned}\approx\ & x \overset{\text{pop}}{\longleftarrow} L;\ \mathbf{S}_0;\\ & \textbf{if } \mathbf{B}_0 \to L := L +\!\!+ \langle\rangle;\ \textbf{DO}\\ & \square \ldots \square\ \mathbf{B}_j \to F * (L +\!\!+ \langle g_{j1}(x), g_{j2}(x),\\ & \qquad\qquad \ldots, g_{jn_j}(x)\rangle);\\ & \qquad\qquad L := \langle\rangle\\ & \ldots \textbf{ fi}\end{aligned}$$

Since $F(x);\ F(y)\ \approx\ F(y);\ F(x)$ for any x, y we can re-order the list we map F to:

$$\approx\ x \xleftarrow{\text{pop}} L;\ \mathbf{S}_0;$$
$$\textbf{if } \mathbf{B}_0 \to L := L +\!\!+ \langle\rangle;\ \textbf{DO}$$
$$\square \ldots \square\ \mathbf{B}_j \to F * (\langle g_{j1}(x), g_{j2}(x),$$
$$\ldots, g_{jn_j}(x)\rangle +\!\!+ L);$$
$$L := \langle\rangle$$
$$\ldots \textbf{ fi}$$

$$\approx\ x \xleftarrow{\text{pop}} L;\ \mathbf{S}_0;$$
$$\textbf{if } \mathbf{B}_0 \to L := L +\!\!+ \langle\rangle;\ \textbf{DO}$$
$$\square \ldots \square\ \mathbf{B}_j \to F * \langle g_{j1}(x), g_{j2}(x),$$
$$\ldots, g_{jn_j}(x)\rangle;$$
$$\textbf{DO}$$
$$\ldots \textbf{ fi}$$

Take **DO** out of the **if** statement and fold a call to $G(x)$:

$$\approx\ x \xleftarrow{\text{pop}} L;\ \mathbf{S}_0;$$
$$F * G(x);$$
$$\textbf{DO}$$

We have $F(x)\ \approx\ \mathbf{S}_0;\ F * G(x)$, so we can fold a call to F:

$$\approx\ x \xleftarrow{\text{pop}} L;\ F(x);\ \textbf{DO}$$

and roll up a loop to get:

$$\approx\ \textbf{DO}$$ ■

14.3. Theorem 9.1

To prove that the tree searching program $F(x)$ is refined by $F_B(x)$ we first note that $F_B(x)$ can be expressed as follows:

proc $F_B(x)\ \equiv$
var $L := \langle x\rangle, L' := \langle\rangle$:
while $L' +\!\!+ L \neq \langle\rangle$ **do**
if $L' = \langle\rangle$ **then** $L' := L;\ L := \langle\rangle$ **fi**;
$x \xleftarrow{\text{pop}} L'$;
if B then $\mathbf{S}_0;\ L := L +\!\!+ G(x)$ **fi od end.**

where we represent L by $L' +\!\!+ L$. Now we selectively unroll the loop under the condition $L' \neq \langle\rangle$ to get:

proc $F_B(x)\ \equiv$
var $L := \langle x\rangle, L' := \langle\rangle$:
while $L' +\!\!+ L \neq \langle\rangle$ **do**
if $L' = \langle\rangle$ **then** $L' := L;\ L := \langle\rangle$ **fi**;
$x \xleftarrow{\text{pop}} L'$;
if B then $\mathbf{S}_0;\ L := L +\!\!+ G(x)$ **fi**;
while $L' \neq \langle\rangle$ **do**
if $L' = \langle\rangle$ **then** $L' := L;\ L := \langle\rangle$ **fi**;
$x \xleftarrow{\text{pop}} L'$;
if B then $\mathbf{S}_0;\ L := L +\!\!+ G(x)$ **fi od od end.**

Now we have $L' = \langle\rangle$ at the beginning of the outer loop, so this is simplified to:

proc $F_B(x)\ \equiv$
var $L := \langle x\rangle, L' := \langle\rangle$:
while $L \neq \langle\rangle$ **do**
$L' := L;\ L := \langle\rangle$;
while $L' \neq \langle\rangle$ **do**
$x \xleftarrow{\text{pop}} L'$;
if B then $\mathbf{S}_0;\ L := L +\!\!+ G(x)$ **fi od od end.**

where we have also rolled one step into the inner loop. Let:

$$\mathsf{DFS} = \textbf{while } L \neq \langle\rangle \textbf{ do}$$
$$x \xleftarrow{\text{pop}} L;$$
$$\textbf{if B then } \mathbf{S}_0;\ L := G(x) +\!\!+ L \textbf{ fi od}$$

and

$$\mathsf{BFS} = \textbf{while } L \neq \langle\rangle \textbf{ do}$$
$$x \xleftarrow{\text{pop}} L;$$
$$\textbf{if B then } \mathbf{S}_0;\ L := L +\!\!+ G(x) \textbf{ fi od}$$

and

$$\mathsf{BFS}' = \textbf{while } L \neq \langle\rangle \textbf{ do}$$
$$L' := L;\ L := \langle\rangle;$$
$$\textbf{while } L' \neq \langle\rangle \textbf{ do}$$
$$x \xleftarrow{\text{pop}} L';$$
$$\textbf{if B then } \mathbf{S}_0;\ L := L +\!\!+ G(x) \textbf{ fi od od}$$

Note that $\mathsf{BFS}\ \approx\ \mathsf{BFS}'$ as above. We will show by induction $\mathsf{DFS}^n\ \leq\ \mathsf{BFS}$ for all $n < \omega$ whence $F_D(x)\ \leq\ F_B(x)$ follows. We may assume that $L \neq \langle\rangle$:

$$\mathsf{DFS}^{n+1} = x \xleftarrow{\text{pop}} L;$$
$$\textbf{if B then } \mathbf{S}_0;\ L := G(x) +\!\!+ L \textbf{ fi};$$
$$\mathsf{DFS}^n$$
$$\leq\ x \xleftarrow{\text{pop}} L;$$
$$\textbf{if B then } \mathbf{S}_0;\ L := G(x) +\!\!+ L \textbf{ fi};$$
$$\mathsf{BFS}$$

by the induction hypothesis

$$\approx\ x \xleftarrow{\text{pop}} L;$$
$$\textbf{if B then } \mathbf{S}_0;\ L := G(x) +\!\!+ L;\ \mathsf{BFS}$$
$$\textbf{else } \mathsf{BFS} \textbf{ fi}$$
$$\leq\ x \xleftarrow{\text{pop}} L;$$
$$\textbf{if B then } \mathbf{S}_0;\ L := L +\!\!+ G(x);\ \mathsf{BFS}$$
$$\textbf{else } \mathsf{BFS} \textbf{ fi}$$

see below

$$\approx\ x \xleftarrow{\text{pop}} L;$$
$$\textbf{if B then } \mathbf{S}_0;\ L := L +\!\!+ G(x) \textbf{ fi};$$
$$\mathsf{BFS}$$
$$\approx\ \mathsf{BFS}$$

So all that remains is to prove $L := G(x) +\!\!+ L;\ \mathsf{BFS}\ \leq\ L := L +\!\!+ G(x);\ \mathsf{BFS}$, or equivalently $L := L_1 +\!\!+ L_2;\ \mathsf{BFS}'\ \leq\ L := L_2 +\!\!+ L_1;\ \mathsf{BFS}'$. We do this by induction on the length of $G(x)$, moving the elements from the end of L_2 to the front of L_1:

$$L := L_1 +\!\!+ \langle x_1\rangle;\ \mathsf{BFS}'^{n+1}$$
$$\approx\ L := L_1 +\!\!+ \langle x_1\rangle;\ L' := L;\ L := \langle\rangle;$$
$$\textbf{while } L' \neq \langle\rangle \textbf{ do}$$
$$x \xleftarrow{\text{pop}} L';$$
$$\textbf{if B then } \mathbf{S}_0;\ L := L +\!\!+ G(x) \textbf{ fi od};$$
$$\mathsf{BFS}'^n$$

Unroll the last step of the **while** loop:

$\approx$ $L' := L_1;\ L := \langle\rangle;$
while $L' \neq \langle\rangle$ **do**
$x \xleftarrow{\text{pop}} L';$
if B then $\mathbf{S}_0;\ L := L +\!\!+ G(x)$ **fi od;**
$x := x_1;$ **if B then** $\mathbf{S}_0;\ L := L +\!\!+ G(x_1)$ **fi;**
BFS'^n

Push BFS'^n inside and apply the induction hypothesis:

$\leq$ $L' := L_1;\ L := \langle\rangle;$
while $L' \neq \langle\rangle$ **do**
$x \xleftarrow{\text{pop}} L';$
if B then $\mathbf{S}_0;\ L := L +\!\!+ G(x)$ **fi od;**
$x := x_1;$
if B then $\mathbf{S}_0;\ L := G(x_1) +\!\!+ L;\ \mathsf{BFS}'$
else BFS' **fi**

Re-arrange:

$\leq$ $L' := L_1;\ L := \langle\rangle;$
while $L' \neq \langle\rangle$ **do**
$x \xleftarrow{\text{pop}} L';$
if B then $\mathbf{S}_0;\ L := L +\!\!+ G(x)$ **fi od;**
if B then $\mathbf{S}_0[x_1/x];\ L := G(x_1) +\!\!+ L$ **fi;**
BFS'

We want to move the statement $\mathbf{S}_0[x_1/x]$ to the beginning. Note that if $\mathbf{B}$ ever becomes false then $\mathsf{BFS}' \approx L := \langle\rangle$ so we can take the assignments to L out of the tests of $\mathbf{B}$:

$\leq$ $L' := L_1;\ L := \langle\rangle;$
while $L' \neq \langle\rangle$ **do**
$x \xleftarrow{\text{pop}} L';$
if B then $\mathbf{S}_0$ **fi;** $L := L +\!\!+ G(x)$ **od;**
if B then $\mathbf{S}_0[x_1/x]$ **fi;**
$L := G(x_1) +\!\!+ L;\ \mathsf{BFS}'$

(technically, we unroll all the steps of the **while** loop, push BFS' into the **if** structure and replace BFS' by the equivalent $L := L +\!\!+ G(x);\ \mathsf{BFS}'$ when $\mathbf{B}$ is false). Now we can use the commutativity of $\mathbf{S}_0$ to get:

$\approx$ $L' := L_1;\ L := \langle\rangle;$ **if B then** $\mathbf{S}_0[x_1/x]$ **fi;**
while $L' \neq \langle\rangle$ **do**
$x \xleftarrow{\text{pop}} L';$
if B then $\mathbf{S}_0$ **fi;** $L := L +\!\!+ G(x)$ **od;**
$L := G(x_1) +\!\!+ L;\ \mathsf{BFS}'$

Since the **while** loop appends to L we can move the assignment $L := G(x_1) +\!\!+ L$ to the beginning. Since $L = \langle\rangle$ we can write it as $L := L +\!\!+ G(x_1)$. Also, as above, if $\mathbf{B}$ is not true then the value of L is immaterial, so we only need to assign to L when $\mathbf{B}$ is true:

$\approx$ $L' := L_1;\ L := \langle\rangle;$
if B then $\mathbf{S}_0[x_1/x];\ L := L +\!\!+ G(x_1)$ **fi;**
while $L' \neq \langle\rangle$ **do**
$x \xleftarrow{\text{pop}} L';$
if B then $\mathbf{S}_0$ **fi;** $L := L +\!\!+ G(x)$ **od;**
BFS'

re-arrange:

$\approx$ $L' := \langle x_1\rangle +\!\!+ L_1;\ L := \langle\rangle;\ x \xleftarrow{\text{pop}} L';$
if B then $\mathbf{S}_0;\ L := L +\!\!+ G(x)$ **fi;**
while $L' \neq \langle\rangle$ **do**
$x \xleftarrow{\text{pop}} L';$
if B then $\mathbf{S}_0$ **fi;** $L := L +\!\!+ G(x)$ **od;**
BFS'

Finally, roll up the **while** loop and absorb it into BFS':

$\approx$ $L := \langle x_1\rangle +\!\!+ L_1;\ \mathsf{BFS}'$

This completes the proof of the theorem. ■

REFERENCES

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

[2] R. Bird, "Notes on Recursion Removal," *Comm. ACM* 20 (June, 1977), 434–439.

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

[4] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," Durham University, Technical Report, 1994, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/foundation2-t.ps.gz⟩.

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

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

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

[8] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," *IEEE Trans. Software Eng.* 22 (Sept., 1996), 665–686, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/sw-alg.ps.gz⟩.

[9] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/sorting-t.ps.gz⟩.

[10] M. Ward, "Iterative Procedures for Computing Ackermann's Function," Durham University, Technical Report 89-3, Feb., 1989, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/ack-t.ps.gz⟩.

[11] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/prog-spec.ps.gz⟩.

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

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

[14] C. A. R. Hoare, I. J. Hayes, H. E. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey & B. A. Sufrin, "Laws of Programming," *Comm. ACM* 30 (Aug., 1987), 672–686.

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

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

[17] C. B. Jones, K. D. Jones, P. A. Lindsay & R. Moore, *mural: A Formal Development Support System*, Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[18] M. Neilson, K. Havelund, K. R. Wagner & E. Saaman, "The RAISE Language, Method and Tools," *Formal Aspects of Computing* 1 (1989), 85–114 .

[19] J. R. Abrial, S. T. Davis, M. K. O. Lee, D. S. Neilson, P. N. Scharbach & I. H. Sørensen, *The B Method*, BP Research, Sunbury Research Centre, U.K., 1991.

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

[21] F. L. Bauer & The CIP Language Group, *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, Lect. Notes in Comp. Sci. #183, Springer-Verlag, New York–Heidelberg–Berlin, 1985.

[22] F. L. Bauer & The CIP System Group, *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, Lect. Notes in Comp. Sci. #292, Springer-Verlag, New York–Heidelberg–Berlin, 1987.

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

[24] M. E. Majester, "Limits of the 'Algebraic' Specification of Abstract Data Types," *SIGPLAN Notices* 12 (Oct., 1977), 37–42.

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

[26] R. Bird, "Lectures on Constructive Functional Programming," in *Constructive Methods in Computing Science*, M. Broy, ed., NATO ASI Series #F55, Springer-Verlag, New York–Heidelberg–Berlin, 1989, 155–218.

[27] H. Partsch, "The CIP Transformation System," in *Program Transformation and Programming Environments* Report on a Workshop directed by F. L. Bauer and H. Remus, P. Pepper, ed., Springer-Verlag, New York–Heidelberg–Berlin, 1984, 305–323.

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

[29] E. Engeler, *Formal Languages: Automata and Structures*, Markham, Chicago, 1968.

[30] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, Amsterdam, 1980.

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

[32] I. J. Hayes, *Specification Case Studies*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[33] C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[34] E. J. Younger & M. Ward, "Inverse Engineering a simple Real Time program," *J. Software Maintenance: Research and Practice* 6 (1993), 197–234, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/eddy-t.ps.gz⟩.

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

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

[37] J. Arsac, *An Interactive Program Manipulation System for Non-Naïve Users*, LITP Res. Rep. Institut de Programmation, Paris, 1978.

[38] J. Arsac, "Syntactic Source to Source Program Transformations and Program Manipulation," *Comm. ACM* 22 (Jan., 1982), 43–54.

[39] J. Arsac, "Transformation of Recursive Procedures," in *Tools and Notations for Program Construction*, D. Neel, ed., Cambridge University Press, Cambridge, 1982, 211–265.

[40] B. A. Davey & H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, Cambridge, 1990.

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

[42] M. Ward, "A Recursion Removal Theorem—Proof and Applications," Durham University, Technical Report, 1991, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/rec-proof-t.ps.gz⟩.

[43] R. M. Burstall & J. A. Darlington, "A Transformation System for Developing Recursive Programs," *J. Assoc. Comput. Mach.* 24 (Jan., 1977), 44–67.

[44] E. W. Dijkstra, "On the Interplay Between Mathematics and Programming," in *Program Construction*, G. Goos & H. Hartmanis, eds., Lect. Notes in Comp. Sci. #69, Springer-Verlag, New York–Heidelberg–Berlin, 1979, 35–46.

[45] C. A. R. Hoare, "Quicksort," *Comput. J.* 5 (1962), 10–15.

[46] J. Darlington, "A Synthesis of Several Sort Programs," *Acta Informatica* 11 (1978), 1–30.

[47] R. Sedgewick, *Algorithms*, Addison Wesley, Reading, MA, 1988.

[48] P. T. Breuer, K. Lano & J. Bowen, "Understanding Programs through Formal Methods," Oxford University, Programming Research Group, Apr., 1991.

[49] W. Ackermann, "Zum Hilbertschen Aufbau der reellen Zahlen," *Math. Ann.* 99 (1928), 118–133.