A Recursion Removal Theorem—Proof and Applications

Martin Ward
Computer Science Dept
Science Labs
South Rd
Durham DH1 3LE

February 23, 1999

Abstract

In this paper we briefly introduce a Wide Spectrum Language and its transformation theory and describe a recent success of the theory: a general recursion removal theorem. This theorem includes as special cases the two techniques discussed by Knuth [12] and Bird [7]. We describe some applications of the theorem to cascade recursion, binary cascade recursion, Gray codes, the Towers of Hanoi problem, and an inverse engineering problem.

1 Introduction

In this paper we briefly introduce some of the ideas behind the transformation theory we have developed over the last eight years at Oxford and Durham Universities and describe a recent result: a general recursion removal theorem.

We use a Wide Spectrum Language (called WSL), developed in [19,20,21] which includes low-level programming constructs and high-level abstract specifications within a single language. Working within a single language means that the proof that a program correctly implements a specification, or that a specification correctly captures the behaviour of a program, can be achieved by means of formal transformations in the language. We don't have to develop transformations between the "programming" and "specification" languages. An added advantage is that different parts of the program can be expressed at different levels of abstraction, if required.

Refinement is defined in terms of the denotational semantics of the language: the semantics of a program $\mathbf{S}$ is a function which maps from an initial state to a final set of states. The set of final states represents all the possible output states of the program for the given input state. Using a set of states enables us to model nondeterministic programs and partially defined (or incomplete) specifications. For programs $\mathbf{S}_1$ and $\mathbf{S}_2$ we say $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ (or $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$) and write $\mathbf{S}_1 \leqslant \mathbf{S}_2$ if $\mathbf{S}_2$ is more defined and more deterministic than $\mathbf{S}_1$. If $\mathbf{S}_1 \leqslant \mathbf{S}_2$ and $\mathbf{S}_2 \leqslant \mathbf{S}_1$ then we say $\mathbf{S}_1$ is equivalent to $\mathbf{S}_2$ and write $\mathbf{S}_1 \approx \mathbf{S}_2$. A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an equivalent program. Equivalence is thus defined in terms of the external "black box" behaviour of the program. See [19] and [20] for a description of the semantics of WSL and the methods used for proving the correctness of transformations.

In developing a model based theory of semantic equivalence, we use the popular approach of defining a core "kernel" language with denotational semantics, and permitting definitional extensions in terms of the basic constructs. In contrast to other work, we do not use a purely applicative kernel; instead, the concept of state is included, using the "atomic description" construct of Back [4] which also allows specifications expressed in first order logic as part of the language, thus providing a genuine "wide spectrum language". Unlike the CIP project [5] and others (eg [6,8]) our kernel language will have state introduced right from the start so that it can cope easily with imperative

programs. Our experience is that an imperative kernel language with functional extensions is more tractable than a functional kernel language with imperative extensions. Unlike Bird [8] we did not want to be restricted to a purely functional language since this is incompatible with the aims of a true wide spectrum language.

This approach has proved highly successful, our successes to date include:

- Deriving complex algorithms in a systematic way from their specifications;
- Improving the efficiency of programs;
- Deriving the specification of an unstructured program from the source code ("Inverse Engineering");
- Discovering bugs in a program by attempting to transform it into a specification.

2 The Wide Spectrum Language

Our kernel language has two primitive statements: the atomic specification and the guard statement. The atomic specification is based on Back's atomic description [4]; it is written $\mathbf{x}/\mathbf{y}.\mathbf{Q}$, where $\mathbf{Q}$ is a formula of first order logic and $\mathbf{x}$ and $\mathbf{y}$ are sets of variables. Its effect is to add the variables in $\mathbf{x}$ to the state space, assign new values to them such that $\mathbf{Q}$ is satisfied, remove the variables in $\mathbf{y}$ from the state and terminate. If there is no assignment to the variables in $\mathbf{x}$ which satisfies $\mathbf{Q}$ then the atomic specification does not terminate. The guard statement is written $[\mathbf{P}]$, where $\mathbf{P}$ is a formula of first order logic. The statement $[\mathbf{P}]$ always terminates, it enforces $\mathbf{P}$ to be true at this point in the program. In effect it restricts previous nondeterminism to those cases which leave $\mathbf{P}$ true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore all possible final states will satisfy any desired condition. Hence the "null guard", $[\mathsf{false}]$, is a "correct refinement" of *any* specification whatsoever. Clearly guard statements cannot be directly implemented but they are nonetheless a useful theoretical tool.

Morgan and others [13,14,15,16] use a different specification statement, written $\mathbf{x}$: $[\mathsf{Pre}, \mathsf{Post}]$. This statement is guaranteed to terminate for all initial states which satisfy Pre and will terminate in a state which satisfies Post while only assigning to variables in the list $\mathbf{x}$. It is thus a combination of an assignment and a guard statement. In our notation an equivalent statement is $\{\mathsf{Pre}\}$; $[\exists \mathbf{x}.\, \mathsf{Post}]$; $\mathbf{x}/\langle\rangle.\mathsf{Post}$. We find it more natural to separate guard statements from assignments, otherwise it is easy to (correctly) refine a specification into an (unimplementable) null statement.

The kernel language is constructed from these two primitive statements, a set of *statement variables* (these are symbols which will be used to represent the recursive calls of recursive statements) and the following three compounds:

1. **Sequential Composition**: $(\mathbf{S}_1;\ \mathbf{S}_2)$
   First $\mathbf{S}_1$ is executed and then $\mathbf{S}_2$.
2. **Choice:** $(\mathbf{S}_1\ \sqcap\ \mathbf{S}_2)$
   One of the statements $\mathbf{S}_1$ or $\mathbf{S}_2$ is chosen for execution.
3. **Recursive Procedure:** $(\mu X.\mathbf{S}_1)$
   Within the body $\mathbf{S}_1$, occurrences of the statement variable X represent recursive calls to the procedure.

The kernel language we have developed is particularly elegant and tractable but is too primitive to form a useful wide spectrum language for the transformational development of programs. For this purpose we need to extend the language by defining new constructs in terms of the existing ones using "definitional transformations". A series of new "language levels" is built up, with the language at each level being defined in terms of the previous level: the kernel language is the "level zero" language which forms the foundation for all the others. Each new language level automatically inherits the transformations proved at the previous level, these form the basis of a new transfor-

mation catalogue. Transformations of the new language construct are proved by appealing to the definitional transformation of the construct and carrying out the actual manipulation in the previous level language. This technique has proved extremely powerful in the development of a practical transformation system which currently implements over three hundred transformations, accessible through a simple user interface [10].

Within expressions we use the following notation:

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 $s[i\,..\,\ell(s)]$.

Sequence concatenation: $s_1 +\!\!+ s_2 = \langle s_1[1], \ldots, s_1[\ell(s_1)], s_2[1], \ldots, s_2[\ell(s_2)] \rangle$.

Stacks: Sequences are also used to implement stacks, for this purpose we have the following notation: For a sequence s and variable x: $x \leftarrow s$ means $x := s[1];\ s := s[2\,..]$ which pops an element of the stack into variable x. To push the value of the expression e onto stack s we use:$s := \langle e \rangle +\!\!+ s$.

Sets: We have the usual set operations $\cup$ (union), $\cap$ (intersection) and $-$ (set difference), $\subseteq$ (subset), $\in$ (element), $\mathcal{P}$ (powerset). $\{\, x \in A \mid P(x) \,\}$ is the set of all elements in A which satisfy predicate P. For the sequence s, $\mathsf{set}(s)$ is the set of elements of the sequence, i.e. $\mathsf{set}(s) = \{\, s[i] \mid 1 \leqslant i \leqslant \ell(s) \,\}$.

Substitution: The expression $\mathbf{S}[\mathbf{S}_2/\mathbf{S}_1]$ where $\mathbf{S}$, $\mathbf{S}_1$ and $\mathbf{S}_2$ are statements means the result of replacing all occurrences of $\mathbf{S}_1$ in $\mathbf{S}$ by $\mathbf{S}_2$.

2.1 Language Extensions

The first set of language extensions are as follows. These go to make up the "first level" language. Subsequent extensions will be defined in terms of the first level language. For the purposes of this paper we will describe only a subset of the language extensions. See [20] and [19] for a more complete definition.

- Sequential composition: The sequencing operator is associative so we can eliminate the brackets:

$$\mathbf{S}_1;\ \mathbf{S}_2;\ \mathbf{S}_3;\ \ldots;\ \mathbf{S}_n \quad =_{\mathrm{DF}} \quad (\ldots((\mathbf{S}_1;\ \mathbf{S}_2);\ \mathbf{S}_3);\ \ldots;\ \mathbf{S}_n)$$

- Deterministic Choice: We can use guards to turn a nondeterministic choice into a deterministic choice:

$$\textbf{if } \mathbf{B} \textbf{ then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi} \quad =_{\mathrm{DF}} \quad (([\mathbf{B}];\ \mathbf{S}_1)\ \sqcap\ ([\neg\mathbf{B}];\ \mathbf{S}_2))$$

- Assertion: An assertion is a partial **skip** statement, it aborts if the condition is false but does nothing if the condition is true. It can be defined using an atomic specification which changes no variables:

$$\{\mathbf{B}\} \quad =_{\mathrm{DF}} \quad \langle\rangle/\langle\rangle.\mathbf{B}$$

- Assignment: We can express a general assignment using a pair of atomic specifications:

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q} \quad =_{\mathrm{DF}} \quad \mathbf{x}'/\langle\rangle.\mathbf{Q};\ \mathbf{x}/\mathbf{x}'.(\mathbf{x} = \mathbf{x}')$$

- Simple Assignment: If $\mathbf{Q}$ is of the form $\mathbf{x}' = \mathbf{t}$ where $\mathbf{t}$ is a list of terms and $\mathbf{x}''$ is a list of new variables, then:

$$\mathbf{x} := \mathbf{t} \quad =_{\mathrm{DF}} \quad \mathbf{x}''/\langle\rangle.(\mathbf{x} = \mathbf{t});\ \mathbf{x}'/\mathbf{x}''.(\mathbf{x}' = \mathbf{x}'')$$

- Nondeterministic Choice: The "guarded command" of Dijkstra [11]:

$$\begin{array}{lcl} \textbf{if } \mathbf{B}_1 \rightarrow \mathbf{S}_1 & =_{\text{DF}} & (((\ldots(([\mathbf{B}_1];\ \mathbf{S}_1) \sqcap \\ \square\ \mathbf{B}_2 \rightarrow \mathbf{S}_2 & & \quad ([\mathbf{B}_2];\ \mathbf{S}_2)) \sqcap \\ \quad \ldots & & \quad \ldots) \sqcap \\ \square\ \mathbf{B}_n \rightarrow \mathbf{S}_n \textbf{ fi} & & \quad ([\mathbf{B}_n];\ \mathbf{S}_n)) \sqcap \\ & & \quad ([\neg(\mathbf{B}_1 \vee \mathbf{B}_2 \vee \cdots \vee \mathbf{B}_n)];\ \textbf{abort})) \end{array}$$

- Deterministic Iteration: We define a **while** loop using a new recursive procedure X which does not occur free in $\mathbf{S}$:

$$\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od} \quad =_{\text{DF}} \quad (\mu X.(([\mathbf{B}];\ \mathbf{S};\ X) \sqcap [\neg\mathbf{B}]))$$

- Nondeterministic Iteration:

$$\begin{array}{lcl} \textbf{do } \mathbf{B}_1 \rightarrow \mathbf{S}_1 & =_{\text{DF}} & \textbf{while } (\mathbf{B}_1 \vee \mathbf{B}_2 \vee \cdots \vee \mathbf{B}_n) \textbf{ do} \\ \square\ \mathbf{B}_2 \rightarrow \mathbf{S}_2 & & \quad \textbf{if } \mathbf{B}_1 \rightarrow \mathbf{S}_1 \\ \quad \ldots & & \quad \square\ \mathbf{B}_2 \rightarrow \mathbf{S}_2 \\ \square\ \mathbf{B}_n \rightarrow \mathbf{S}_n \textbf{ od} & & \quad \ldots \\ & & \quad \square\ \mathbf{B}_n \rightarrow \mathbf{S}_n \textbf{ fi od} \end{array}$$

- Initialised local Variables:

$$\textbf{var } \mathbf{x} := \mathbf{t}\colon \mathbf{S} \textbf{ end} \quad =_{\text{DF}} \quad \mathbf{x}/\langle\rangle.(\mathbf{x} = \mathbf{t};\ \mathbf{S};\ \langle\rangle/\mathbf{x}.\textbf{true}$$

- Counted Iteration:

$$\begin{array}{lcl} \textbf{for } i := b \textbf{ to } f \textbf{ step } s \textbf{ do } \mathbf{S} \textbf{ od} & =_{\text{DF}} & \textbf{var } i := b\colon \\ & & \quad \textbf{while } i \leqslant f \textbf{ do} \\ & & \qquad \mathbf{S};\ i := i + s \textbf{ od end} \end{array}$$

- Procedure call:

$$\textbf{proc } X \ \equiv\ \mathbf{S}. \quad =_{\text{DF}} \quad (\mu X.\mathbf{S})$$

- Block with local procedure:

$$\textbf{begin } \mathbf{S}_1 \textbf{ where proc } X \ \equiv\ \mathbf{S}_2. \textbf{ end} \quad =_{\text{DF}} \quad \mathbf{S}_1[\textbf{proc } X \ \equiv\ \mathbf{S}_2./X]$$

2.2 Exit Statements

Our programming language will include statements of the form **exit**(n), where n is an integer, (*not* a variable) which occur within loops of the form **do S od** where **S** is a statement. These were described in [12] and more recently in [18]. They 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. To simplify the language we disallow **exit**s which leave a block or a loop other than an unbounded loop).

Previously, the only formal treatments of **exit** statements have treated them in the same way as unstructured **goto** statements by adding "continuations" to the denotational semantics of all the other statements. This adds greatly to the complexity of the semantics and also means that all the results obtained prior to this modification will have to be re-proved with respect to the new semantics. The approach taken in our work, which does not seem to have been tried before, is to express every program which uses **exit** statements and unbounded loop in terms of the first

level language *without* changing the language semantics. This means that the new statements will not change the denotational semantics of the kernel so all the transformations developed without reference to **exit** statements will still apply in the more general case. In fact we make much use of the transformations derived without reference to **exit**s in the derivation of transformations of statements which use the **exit** statement.

The interpretation of these statements in terms of the first level language is as follows:

We have an integer variable depth which records the current depth of nesting of loops. At the beginning of the program we have depth := 0 and each **exit** statement **exit**(k) is translated: depth := depth $- k$ since it changes the depth of "current execution" by moving out of k enclosing loops. To prevent any more statements at the current depth being executed after an **exit** statement has been executed we surround all statements by "guards" which are **if** statements which will test depth and only allow the statement to be executed if depth has the correct value. Each unbounded loop **do S od** is translated:

$$\mathsf{depth} := n;\ \textbf{while}\ \mathsf{depth} = n\ \textbf{do}\ \mathsf{guard}_n(\mathbf{S})\ \textbf{od}$$

where n is an integer constant representing the depth of the loop (1 for an outermost loop, 2 for double nested loops etc.) and $\mathsf{guard}_n(\mathbf{S})$ is the statement **S** with each component statement guarded so that if the depth is changed by an **exit** statement then no more statements in the loop will be executed and the loop will terminate. The important property of a guarded statement is that it will only be executed if depth has the correct value. Thus: $\{\mathsf{depth} \neq n\};\ \mathsf{guard}_n(\mathbf{S}) \approx \textbf{skip}$. So for example, the program:

```
do do last := item[i];
        i := i + 1;
        if i = n + 1 then write(count); exit(2) fi;
        if item[i] ≠ last then write(count); exit(1)
                          else count := count + number[i] fi od;
   count := number[i] od
```

translates to the following:

```
depth := 1;
while depth = 1 do
      depth := 2;
      while depth = 2 do
            last := item[i];
            i := i + 1;
            if i = n + 1 then write(count); depth := depth − 2 fi;
            if depth = 2
              then if item[i] ≠ last then write(count); depth := depth − 1
                                    else count := count + number[i] fi od;
              if depth = 1 then count := number[i] fi od
```

2.3 Action Systems

This section will introduce the concept of an *Action System* as a set of parameterless mutually recursive procedures. A program written using labels and jumps translates directly into an action system. Note however that if the end of the body of an action is reached, then control is passed to the action which called it (or to the statement following the action system) rather than "falling through" to the next label. 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.

Our recursive statement does not directly allow the definition of mutually recursive procedures (since all calls to a procedure must occur within the procedure body). However we can define a

set of mutually recursive procedures by putting them all within a single procedure. For example suppose we have two statements, $\mathbf{S}_1$ and $\mathbf{S}_2$ both containing statement variables X_1 and X_2 (where we intend $\mathbf{S}_1$ to be the body of X_1 and $\mathbf{S}_2$ to be the body of X_2). We can represent these by a single recursive program:

$x := 1;$
proc $A \equiv$ **if** $x = 1 \rightarrow \mathbf{S}_1[x := 1;\ A/X_1][x := 2;\ A/X_2]$
$\quad \square\ x = 2 \rightarrow \mathbf{S}_2[x := 1;\ A/X_1][x := 2;\ A/X_2]$ **fi.**

where an additional variable x records which procedure is required when the composite procedure A is called.

Arsac [2,3] uses a restricted definition of actions together with deterministic assignments, the binary **if** statement and **do** loops with **exit**s so there is no place for nondeterminism in his results. The main differences between our action systems and Arsac's are: (i) that we use a much more powerful language (including general specifications) , (ii) we give a formal definition (ultimately in terms of denotational semantics), and (iii) our action systems are simple statements which can form components of other constructs. This last point is vitally important in this application since it gives us a way to restructure the body of a recursive procedure as an action system. It is this restructuring which gives the recursion removal theorem much of its power and generality.

Definition 2.1 An *action* is a parameterless procedure acting on global variables (cf [2,3]). It is written in the form $A \equiv \mathbf{S}$ where A is a statement variable (the name of the action) and $\mathbf{S}$ is a statement (the action body). A set of (mutually recursive) actions is called an *action system*. There may sometimes be a special action (usually denoted Z), execution of which causes termination of the whole action system even if there are unfinished recursive calls. An occurrence of a statement **call** X within the action body refers to a call of another action.

The action system:

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

(where statements $\mathbf{S}_1, \ldots, \mathbf{S}_n$ must have no **exit**(n) statements within less than n nested loops) is defined as follows:

var action := "A_1";
proc $A \equiv$ **if** action = "A_1" $\rightarrow$ action := "O"; $\mathsf{guard}_Z(\mathbf{S}_1)$[action := "$A_i$"; A/**call** A_i]
$\square$ action = "A_2" $\rightarrow$ action := "O"; $\mathsf{guard}_Z(\mathbf{S}_2)$[action := "$A_i$"; A/**call** A_i]
...
$\square$ action = "A_n" $\rightarrow$ action := "O"; $\mathsf{guard}_Z(\mathbf{S}_n)$[action := "$A_i$"; A/**call** A_i]. **end**

Here action is a new variable which contains the name of the next action to be invoked and $\mathsf{guard}_Z(\mathbf{S})$ is defined in a similar way to $\mathsf{guard}_n(\mathbf{S})$ so that:

$$\mathsf{guard}_Z(\textbf{call } Z) =_{\mathrm{DF}} \mathsf{action} := \text{"}Z\text{"}$$
$$\mathsf{guard}_Z(v := e) =_{\mathrm{DF}} \textbf{if } \mathsf{action} = \text{"}O\text{"} \textbf{ then } v := e \textbf{ fi} \qquad \text{etc.}$$

and as soon as action is set to "Z" no further statements will be executed. This ensures the correct operation of the "halting" action. Here "A_1", ..., "A_n", "O" and "Z" represent a suitable set of $n+2$ distinct constant values.

The procedure A is never called with action equal to "Z" (or in fact anything other than "A_1", ..., "A_n"). The assignment action := "O" is not really needed because the variable action will be assigned again before its value is tested; it is added so that we can distinguish the following three cases depending on the value of action:

1. Currently executing an action: action = "O";
2. About to call another (or the same) action (other than the terminating action): action = one of "A_1", ..., "A_n";
3. Have called the terminating action, all outstanding recursive calls are terminated without any statements being executed: action = "Z".

Definition 2.2 An action is *regular* if every execution of the action leads to an action call. (This is similar to a regular rule in a Post production system [17]).

Definition 2.3 An action system is regular if every action in the system is regular. Any algorithm defined by a flowchart, or a program which contains labels and **goto**s but no procedure calls in non-terminal positions, can be expressed as a regular action system.

2.4 Procedures and Functions with Parameters

For simplicity we will only consider procedures with parameters which are called by value or by value-result. Here the value of the actual parameter is copied into a local variable which replaces the formal parameter in the body of the procedure. For result parameters, the final value of this local variable is copied back into the actual parameter. In this case the actual parameter must be a variable or some other object (eg an array element) which can be assigned a value. Such objects are often denoted as "L-values" because they can occur on the left of assignment statements.

The reason for concentrating on value parameters is that they avoid some of the problems caused by "aliasing" where two variable names refer to the same object. For example if a global variable of the procedure is also used as a parameter, or if the same variable is uses for two actual parameters then with other forms of parameter passing aliasing will occur but with value parameters the aliasing is avoided (unless the same variable is used for two result parameters and the procedure tries to return two different values). This means that procedures with value parameters have simpler semantics.

In most cases the different methods of parameter passing produce the same result, though there may be differences in efficiency. For this reason the language Ada allows the compiler to choose between call by value and call by reference and requires all programs to give the same result whatever method is used: programs which would give different results are technically illegal, although no compiler could determine which programs are legal and which are illegal. It is generally better to specify that a compiler *rejects* certain specific constructs as erroneous rather than simply leaving the result "undefined". (For example: making it an error to access the value of a loop variable after the loop has terminated rather than leaving the value undefined). This prevents programmers making use of the effect produced by a particular compiler and so writing programs which may give different results at a different installation, or with a different version of the compiler.

Other languages (eg Modula) default to passing simple variables by value (to avoid repeated recomputation of expressions) and passing structures and arrays by reference (to avoid copying the whole structure when only part of it may be accessed).

Our "definitional transformation" for a procedure with formal parameters and local variables will replace them both by global stacks. Consider the following piece of code, which contains a call to the recursive procedure F. This procedure uses a local variable a which must be preserved over recursive calls to F:

begin ...; $F(t, v)$; ...
where
proc $F(x, \mathsf{var} : y) \equiv$
  var $a := d$:
    S end.
end

where t is an expression, v a variable, x is a value parameter, v a value-result parameter and a a local variable which is assigned the initial value d. This is defined as:

begin
$x := \langle\rangle;\ y := \langle\rangle;\ a := \langle\rangle;$
$\ldots;$
$x \overset{\text{push}}{\longleftarrow} t;\ y \overset{\text{push}}{\longleftarrow} v;$
$F;$
$v \overset{\text{pop}}{\longleftarrow} y;\ x := x[2\,..];$
$\ldots$
where
proc $F\ \equiv$
$a \overset{\text{push}}{\longleftarrow} d;$
$\mathbf{S}[x[1]/x][y[1]/y][a[1]/a]$
$[x \overset{\text{push}}{\longleftarrow} t';\ y \overset{\text{push}}{\longleftarrow} v';\ F;\ v' \overset{\text{pop}}{\longleftarrow} y;\ x := x[2\,..]/F(t', v')];$
$a := a[2\,..].$
end

Here the substitution of $x[1]$ for x etc. ensures that the body of the procedure only accesses and updates the tops of the stacks which replace the parameters and local variables. This means that any call of F will only affect the values at the tops of the stacks x, y and a so an inner recursive call of F, which takes the form: $x \overset{\text{push}}{\longleftarrow} t';\ y \overset{\text{push}}{\longleftarrow} v';\ F;\ v' \overset{\text{pop}}{\longleftarrow} y;\ x := x[2\,..]$, will only affect the value of v (and global variables in $\mathbf{S}$) and will not affect the stacks. The proof is by the theorems on invariant maintenance for recursive statements [19].

To allow side effects in expressions and conditions we introduce the new notation of "expression brackets", $\lceil$ and $\rfloor$. These allow us to include statements as part of an expression, for example the following are valid expressions:

$\lceil x := x + 1;\ x \rfloor$
$\lceil x := x + 1;\ x - 1 \rfloor$
if $x > 0$ **then** x **else** $-x$ **fi**

The first and second are equivalent to C's `++x` and `x++` respectively, the third is a conditional expression which returns the absolute value of x.

Note that expression brackets may be nested, for example the assignment:
$a := \lceil \mathbf{S}_1;\ b :=$ **if** $\lceil \mathbf{S}_2;\ \mathbf{Q} \rfloor$ **then** $\lceil \mathbf{S}_3;\ \mathbf{t}_1 \rfloor$ **else** $\mathbf{t}_2$ **fi**$;\ b.b \rfloor$

is represented as:
$\mathbf{S}_1;\ \mathbf{S}_2;$ **if** $\mathbf{Q}$ **then** $\mathbf{S}_3;\ b := \mathbf{t}_1$ **else** $b := t_2$ **fi**$;\ a := b.b$

Definition 2.4 *Function calls:* The definitional transformation of a function call will replace the function call by a call to a procedure which assigns the value returned by the function to a variable. This variable then replaces the function call in the expression. Several calls in one expression are replaced by the same number of procedure calls and new variables. Boolean functions are treated as functions which return one of the values "tt" or "ff" (representing true and false). So a boolean function call is replaced by a formula ($b =$ "tt") where b is a new local variable. The statement in which the function call appeared is preceded by a procedure call which sets b to "tt" or "ff", depending on the result of the corresponding boolean function.

For example, the statement with function calls:
begin $a := F(x) + F(y)$
where
funct $F(x)\ \equiv$ **if** $\mathbf{B}$ **then** $\mathbf{t}_1$ **else** $\mathbf{t}_2$ **fi. end**

is interpreted:

```
begin var r1, r2 :
        F(x); r1 := r; F(y); r2 := r;
        a := r1 + r2
  where
  proc F(x) ≡ if B then r := t1 else r := t2 fi. end
```

The statement:

```
begin
  a := ⌈while B(x) do x := F(x) od; x + c⌋
where
funct B(x) ≡ ⌈S; x > y⌋.
funct F(x) ≡ if B then t1 else t2 fi.
```

is interpreted:

```
begin
  do B(x); if r = "ff" then exit fi;
     F(x); x := r od;
  a := x + c where
  proc B(x) ≡ S; if x > y then r := "tt" else r := "ff" fi,
    proc F(x) ≡ if B then r := t1 else r := t2 fi. end
```

See [19] for the formal definition of generalised expressions and generalised conditions and their interpretation functions.

3 Example Transformations

In this section we describe a few of the transformations we will use later:

3.1 Expand IF statement

The **if** statement:

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

can be expanded over the following statement to give:

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

3.2 Loop Inversion

If the statement $\mathbf{S}_1$ contains no **exit**s which can cause termination of an enclosing loop (i.e. in the notation of [19] it is a *proper sequence*) then the loop:

$$\textbf{do } \mathbf{S}_1; \ \mathbf{S}_2 \textbf{ od}$$

can be inverted to:

$$\mathbf{S}_1; \ \textbf{do } \mathbf{S}_2; \ \mathbf{S}_1 \textbf{ od}$$

This transformation may be used in the forwards direction to move the termination test of a loop to the beginning, prior to transforming it into a **while** loop, or it may be used in the reverse direction to merge two copies of the statement $\mathbf{S}_1$.

3.3 Loop Unrolling

The next three transformations concern various forms of loop unrolling. They play an important role in the proofs of other transformations as well as being generally useful.

Lemma 3.1 *Loop Unrolling:*

$$\textbf{while B do S od} \approx \textbf{if B then S; while B do S od fi}$$

Lemma 3.2 *Selective unrolling of* **while** *loops:* For any condition **Q** we have:

$$\textbf{while B do S od} \approx \textbf{while B do S; if B} \wedge \textbf{Q then S fi od}$$

Lemma 3.3 *Entire Loop Unfolding:* if $\mathbf{B}' \Rightarrow \mathbf{B}$ then:

$$\textbf{while B do S od} \approx \textbf{while B do S; if Q then while B}' \textbf{ do S od fi od}$$

Each of these transformation has a generalisation in which instead of inserting the "unrolled" part after **S** it is copied after an arbitrary selection of the terminal statements in **S**.

3.4 Absorption

Definition 3.4 A *primitive statement* is any statement other than a conditional, a **do** ... **od** loop or a sequence of statements. The *depth* of a component of a statement is the number of enclosing **do** ... **od** loops around the component. A *terminal statement* is a primitive statement which is either

(i) in a terminal position, or

(ii) is an **exit**(n) at depth less than n, or

(iii) is an **exit**(n) at depth n where the outermost **do** ... **od** loop is in a terminal position.

The *terminal value* of a terminal statement **exit**(n) is n minus the depth. *Incrementing* a statement by k means adding **exit**(k) after each non-**exit** terminal statement with terminal value zero, and replacing each terminal statement **exit**(n) with terminal value zero by **exit**$(n+k)$.

A sequence **S**; **S**$'$ of two statements can be merged together by the *absorption* The statement **S**$'$ following **S** is "absorbed" into it by replacing all of the terminal statements of **S** which would lead to **S**$'$ by a copy of **S**$'$ incremented by the depth of the terminal statement. For example:

```
do do if y > x then exit fi;
        x := x − 1;
        if x = 0 then exit(2) fi od;
    if z > x then exit fi od;
if z = x then exit fi
```

after absorption becomes:

```
do do if y > x then exit fi;
        x := x − 1;
        if x = 0 then if z = x then exit(3) else exit(2) fi fi od;
    if z > x then if z = x then exit(2) else exit fi od
```

4 The Theorem

Theorem 4.1 Suppose we have a recursive procedure whose body is an action system in the following form, in which the body of the procedure is an action system. (A **call** Z in the action system will therefore terminate only the current invocation of the procedure):

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. There are $M+N$ actions in total: $A_1, \ldots, A_M, B_1, \ldots, B_N$.

We claim that this 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** $/F$/**call** Z].
    $\ldots A_i \equiv \mathbf{S}_i$[**call** $/F$/**call** Z].
    $\ldots B_j \equiv \mathbf{S}_{j0};\ L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j,1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;$
      call $/F$.
    $\ldots /F \equiv$ **if** $L = \langle\rangle$ **then call** Z
      else $\langle m, x\rangle \leftarrow L;$
        if $m = 0 \rightarrow$ **call** A_1
        $\square \ldots \square\ m = \langle j,k\rangle \rightarrow \mathbf{S}_{jk};$ **call** $/F$
        $\ldots$ **fi fi. endactions**
  end.

Note that any procedure $F(x)$ can be restructured into the required form; in fact (as we shall see later) there may be several different ways of structuring $F(x)$ which meet these criteria.

We will assume that the action system is *regular*, i.e. every execution of an action body leads to the call of another action. This means that the action body (and hence the current invocation of F) can only be terminated by a call to action Z. Transformations are presented in [19] to convert any action system into a regular one, perhaps with the aid of a stack. We will also assume for simplicity that all the action calls appear in terminal positions in an action body, regularity then implies that the statement at every terminal position is an action call. Any regular action system can be put into this form by repeated application of the absorption transformation of [19].

Any regular action system can be transformed into a double-nested loop using a transformation proved in [19] so we have the following equivalent for $F(x)$ (where a is a new local variable):

proc $F(x)$ $\equiv$
  var $a :=$ "A_1" :
    do do if $a =$ "A_1" $\rightarrow \mathbf{S}_1[a :=$ "Z"; **exit**(2)/**call** $Z][a :=$ "X"; **exit**/**call** $X]$
      $\square \ldots \square\ a =$ "A_i" $\rightarrow \mathbf{S}_i[a :=$ "Z"; **exit**(2)/**call** $Z][a :=$ "X"; **exit**/**call** $X]$
      $\square \ldots \square\ a =$ "B_j" $\rightarrow \mathbf{S}_{j0};\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x});\ \ldots;\ F(g_{jn_j}(x));$
        $\mathbf{S}_{jn_j}[a :=$ "Z"; **exit**(2)/**call** $Z][a :=$ "X"; **exit**/**call** $X]$
      $\ldots$ **fi od od end.**

The substitutions replace each action call by **exit**(2) (if it is **call** Z) or by an assignment to a followed by an **exit**. Because all calls appear in terminal positions only the last element of the sequence in the body of B_j needs to have the substitution applied.

Since we know that a is only assigned the value "Z" just before the loop is terminated, we can replace the **exit**(2)'s by **exit**'s inside the inner loop and test a outside to see if the outer loop is terminated. Formally this transformation is an inverse of absorption:

proc $F(x)$ $\equiv$
  var $a :=$ "A_1" :
    do do if $a =$ "A_1" $\rightarrow \mathbf{S}_1[a :=$ "Z"; **exit**/**call** $Z][a :=$ "X"; **exit**/**call** $X]$
      $\square \ldots \square\ a =$ "A_i" $\rightarrow \mathbf{S}_i[a :=$ "Z"; **exit**/**call** $Z][a :=$ "X"; **exit**/**call** $X]$
      $\square \ldots \square\ a =$ "B_j" $\rightarrow \mathbf{S}_{j0};\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x});\ \ldots;\ F(g_{jn_j}(x));$
        $\mathbf{S}_{jn_j}[a :=$ "Z"; **exit**/**call** $Z][a :=$ "X"; **exit**/**call** $X]$
      $\ldots$ **fi od**;

if $a =$ "Z" **then exit fi od end.**

Since all calls appear in terminal positions the body of the inner loop is reducible. Also, since the action system is regular, each arm of the inner **if** statement will result in an **exit** so the body of the loop is improper. This means that the inner loop is a false loop which can be removed. The outer loop can be transformed to a **while** loop since the termination test is initially false. We get:

```
proc F(x) ≡
  var a := "A_1" :
    while a ≠ "Z" do
        if a = "A_1" → S_1[a := "X"/call X]
        □ ... □ a = "A_i" → S_i[a := "X"/call X]
        □ ... □ a = "B_j" → S_j0; F(g_j1(x)); S_j1; F(g_j2(x)); ...; F(g_jn_j(x));
                            S_jn_j[a := "X"/call X]
        ... fi od end.
```

We claim that this is equivalent to the following iterative procedure which uses a stack L and another local variable m:

```
proc F'(x) ≡
  var a := "A_1", L := ⟨⟩, m := 0 :
    while a ≠ "Z" do
        if a = "/F" →
            if L = ⟨⟩
              then a := "Z"
               else ⟨m, x⟩ ← L;
                    if m = 0 → a := "A_1"
                    □ ... □ m = ⟨j, k⟩ → S_jk[a := "/F"/call Z][a := "X"/call X]
                    ... fi fi
        □ a = "A_1" → S_1[a := "/F"/call Z][a := "X"/call X]
        □ ... □ a = "A_i" → S_i[a := "/F"/call Z][a := "X"/call X]
        □ ... □ a = "B_j" → S_j0; L := ⟨⟨0, g_j1(x)⟩, ⟨⟨j, 1⟩, x⟩, ⟨0, g_j2(x)⟩,
                                       ..., ⟨⟨j, n_j + 1⟩, x⟩⟩ ++ L;
                            a := "/F"
        ... fi od end.
```

To prove the claim, let **DO**$'$ be the **while** loop above and let **DO** be:

```
while a ≠ "Z" do
    if a = "/F" →
        if L = ⟨⟩
          then a := "Z"
           else ⟨m, x⟩ ← L;
                if m = 0 → a := "A_1"
                □ ... □ m = ⟨j, k⟩ → S_jk[a := "/F"/call Z][a := "X"/call X]
                ... fi fi
    □ a = "A_1" → S_1[a := "X"/call X]
    □ ... □ a = "A_i" → S_i[a := "X"/call X]
    □ ... □ a = "B_j" → S_j0; F(g_j1(x)); S_j1; F(g_j2(x)); ...; F(g_jn_j(x));
                        S_jn_j[a := "X"/call X]
    ... fi od
```

this is the same as the loop in $F(x)$ above with an extra (redundant) test added to the guarded command.

4.1 Preliminary Lemmas

To carry through the proof we require the following two lemmas:

Lemma 4.2 $a :=$ "A_1"; **DO** $\approx$ $F(x)$; $a :=$ "$/F$"; **DO**

Proof: Apply entire loop unrolling to $a :=$ "A_1"; **DO** to get:

```
a := "A1";
while a ≠ "Z" do
    if a = "/F" →
        if L = ⟨⟩
          then a := "Z"
           else ⟨m, x⟩ ← L;
                if m = 0 → a := "A1"
                □ ... □ m = ⟨j, k⟩ → S_jk[a := "/F"/call Z][a := "X"/call X]
                ... fi;
                DO fi
    □ a = "A1" → S_1[a := "X"/call X]
    □ ... □ a = "A_i" → S_i[a := "X"/call X]
    □ ... □ a = "B_j" → S_j0; F(g_j1(x)); S_j1; F(g_j2(x)); ...; F(g_jn_j(x));
                        S_jn_j[a := "X"/call X]
    ... fi od
```

By unrolling **DO** we see that:

```
⟨m, x⟩ ← L;                                                    ≈    a := "/F"; DO
if m = 0 → a := "A1"
□ ... □ m = ⟨j, k⟩ → S_jk[a := "/F"/call Z][a := "X"/call X]
... fi;
DO
```

when $L \neq \langle\rangle$. We also have $a :=$ "Z" $\approx$ $a :=$ "$/F$"; **DO** when $L = \langle\rangle$. So we can simplify the loop body to:

```
a := "A1";
while a ≠ "Z" do
    if a = "/F" →
        if L = ⟨⟩
          then a := "/F"; DO
           else a := "/F"; DO fi
    □ a = "A1" → S_1[a := "X"/call X]
    □ ... □ a = "A_i" → S_i[a := "X"/call X]
    □ ... □ a = "B_j" → S_j0; F(g_j1(x)); S_j1; F(g_j2(x)); ...; F(g_jn_j(x));
                        S_jn_j[a := "X"/call X]
    ... fi od
```

The test of $L = \langle\rangle$ is clearly redundant. We know that $a =$ "Z" is true after **DO** so we can take it out of the loop to get:

```
a := "A1";
while a ≠ "Z" do
    if a = "/F" → a := "Z"
    □ a = "A1" → S_1[a := "X"/call X]
    □ ... □ a = "A_i" → S_i[a := "X"/call X]
    □ ... □ a = "B_j" → S_j0; F(g_j1(x)); S_j1; F(g_j2(x)); ...; F(g_jn_j(x));
                        S_jn_j[a := "X"/call X]
    ... fi od;
```

$a :=$ "$/F$"; **DO**

If we convert this loop into an action system and replace the action $/F$ by the equivalent action Z then we get something identical to the body of $F(x)$. So we can fold it into a procedure call to get:

$F(x)$; $a :=$ "$/F$"; **DO**

as required. This completes the lemma.

The lemma we need for **DO**$'$ is more tricky. Let **IF**^{n} be defined as follows:

$$
\begin{array}{l}
\textbf{IF}^n \;=_{\mathrm{DF}}\; \textbf{if } a = \text{"}/F\text{"} \rightarrow \\
\qquad\qquad \textbf{if } L = \langle\rangle \textbf{ then } a := \text{"}Z\text{"} \\
\qquad\qquad\qquad \textbf{else } \langle m, x\rangle \leftarrow L; \\
\qquad\qquad\qquad\quad \textbf{if } m = 0 \rightarrow a := \text{"}A_1\text{"} \\
\qquad\qquad\qquad\quad \Box \ldots \Box\; m = \langle j, k\rangle \rightarrow \mathbf{S}_{jk}[a := \text{"}/F\text{"}/\textbf{call } Z][a := \text{"}X\text{"}/\textbf{call } X] \\
\qquad\qquad\qquad\qquad\qquad\qquad \ldots \textbf{ fi fi} \\
\qquad \Box\; a = \text{"}A_1\text{"} \rightarrow \mathbf{S}_1[a := \text{"}/F\text{"}/\textbf{call } Z][a := \text{"}X\text{"}/\textbf{call } X] \\
\qquad \Box \ldots \Box\; a = \text{"}A_i\text{"} \rightarrow \mathbf{S}_i[a := \text{"}/F\text{"}/\textbf{call } Z][a := \text{"}X\text{"}/\textbf{call } X] \\
\qquad \Box \ldots \Box\; a = \text{"}B_j\text{"} \rightarrow \mathbf{S}_{j0};\; F^n(g_{j0}(x));\; \mathbf{S}_{j1};\; F^n(g_{j1}(x)); \\
\qquad\qquad\qquad \ldots F^n(g_{jn_j}(x));\; \mathbf{S}_{jn_j}[a := \text{"}/F\text{"}/\textbf{call } Z][a := \text{"}X\text{"}/\textbf{call } X] \textbf{ fi}
\end{array}
$$

Let **IF** be the corresponding statement with F not truncated. From the definition of F we see that $F^{n+1}(x)$ is equivalent to:

$$
\begin{array}{l}
\textbf{var } a := \text{"}A_1\text{"} : \\
\quad \textbf{while } a \neq \text{"}Z\text{"} \wedge a \neq \text{"}/F\text{"} \textbf{ do} \\
\qquad\quad \textbf{IF}^n \textbf{ od end}
\end{array}
$$

Note that if $a \neq$ "Z" initially then **IF**^{n} cannot set a to "Z", so if $a \neq$ "Z" then the loop **while** $a \neq$ "Z" $\wedge$ $a \neq$ "$/F$" **do IF**^{n} **od** must terminate with $a =$ "$/F$" if it terminates at all.

Lemma 4.3 **while** $a \neq$ *"Z"* $\wedge$ $a \neq$ *"/F"* **do IF od**; **DO**$' \leqslant$ **DO**$'$

Proof: We use the induction rule for recursion and prove by induction on n that **while** $a \neq$ "Z" $\wedge$ $a \neq$ "$/F$" **do IF**^{n} **od**; **DO**$' \leqslant$ **DO**$'$ for all $n < \omega$. For the induction step we use a sub-induction which uses the induction rule for iteration and prove: **while** $a \neq$ "Z" $\wedge$ $a \neq$ "$/F$" **do IF**$^{n+1}$ **od**^{m}; **DO**$' \leqslant$ **DO**$'$ for every $m < \omega$.

From the (sub) induction hypothesis we have:

$$
\begin{array}{l}
\textbf{while } a \neq \text{"}Z\text{"} \wedge a \neq \text{"}/F\text{"} \textbf{ do IF}^{n+1} \textbf{ od}^{m+1};\; \textbf{DO}' \\
\quad = \textbf{if } a \neq \text{"}Z\text{"} \wedge a \neq \text{"}/F\text{"} \\
\qquad\quad \textbf{then IF}^{n+1};\; \textbf{while } a \neq \text{"}Z\text{"} \wedge a \neq \text{"}/F\text{"} \textbf{ do IF}^{n+1} \textbf{ od}^m;\; \textbf{DO}' \\
\qquad\quad \textbf{else DO}' \textbf{ fi} \\
\quad \leqslant \textbf{if } a \neq \text{"}Z\text{"} \wedge a \neq \text{"}/F\text{"} \\
\qquad\quad \textbf{then IF}^{n+1};\; \textbf{DO}' \\
\qquad\quad \textbf{else DO}' \textbf{ fi}
\end{array}
$$

so we need to prove **IF**$^{n+1}$; **DO**$' \leqslant$ **DO**$'$. For this we consider the different cases which the **if** statement tests for—these depend on the initial value of a.

Examining the body of **IF**$^{n+1}$ we see that for most of the cases the clause of the **IF** statement is identical to a clause in the **DO** statement, for example:

$$\{a = \text{"}A_i\text{"}\};\; \textbf{IF}^{n+1};\; \textbf{DO}' \;\approx\; \{a = \text{"}A_i\text{"}\};\; \mathbf{S}_i[a := \text{"}/F\text{"}/\textbf{call } Z][a := \text{"}X\text{"}/\textbf{call } X];\; \textbf{DO}'$$

and

$$\{a = \text{"}A_i\text{"}\};\; \textbf{DO}' \;\approx\; \{a = \text{"}A_i\text{"}\};\; \mathbf{S}_i[a := \text{"}/F\text{"}/\textbf{call } Z][a := \text{"}X\text{"}/\textbf{call } X];\; \textbf{DO}'$$

by unrolling the first step of $\mathbf{DO}'$ and pruning. Hence:

$$\{a = \text{“}A_i\text{”}\};\ \mathbf{IF}^{n+1};\ \mathbf{DO}' \ \approx\ \{a = \text{“}A_i\text{”}\};\ \mathbf{DO}'$$

as required. The case $a = \text{“}/F\text{”}$ follows similarly. So we are left with the cases $a = \text{“}B_j\text{”}$, i.e. all that remains is to prove:

$$\begin{aligned}\{a = \text{“}B_j\text{”}\};\ \mathbf{S}_{j0};\ F^{n+1}(g_{j0}(x));\ \mathbf{S}_{j1};\ F^{n+1}(g_{j1}(x));\\ \ldots F^{n+1}(g_{jn_j}(x));\ \mathbf{S}_{jn_j}[a := \text{“}/F\text{”}/\textsf{call } Z][a := \text{“}X\text{”}/\textsf{call } X];\ \mathbf{DO}'\\ \leqslant \{a = \text{“}B_j\text{”}\};\ \mathbf{DO}' \quad (1)\end{aligned}$$

To prove this we use the fact, noted above, that $F^{n+1}(x)$ can be expressed in terms of **while** $a \neq$ “Z” $\wedge$ $a \neq$ “$/F$” **do IF**^{n} **od** and use the main induction hypothesis to show that **while** $a \neq$ “Z” $\wedge$ $a \neq$ “$/F$” **do IF**^{n} **od**; $\mathbf{DO}' \leqslant \mathbf{DO}'$.

First note that the value of a is not used by $\mathbf{S}_{jk}$ or F^{n+1}. Also, since the original body of F contains a regular action system with all the action calls in terminal places, each terminal statement of $\mathbf{S}_{jn_j}$ must be an action call. So $\mathbf{S}_{jn_j}[a := \text{“}/F\text{”}/\textsf{call } Z][a := \text{“}X\text{”}/\textsf{call } X]$ must assign some value to a. So we can add assignments to a anywhere in the sequence on the LHS. By unfolding $\mathbf{DO}'$ and pruning (as in the previous Lemma) we have:

$$\mathbf{S}_{jn_j}[a := \text{“}/F\text{”}/\textsf{call } Z][a := \text{“}X\text{”}/\textsf{call } X];\ \mathbf{DO}' \quad \approx \quad L := \langle\langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \mathbf{DO}'$$

As in the previous Lemma, move the assignment to L to after $\mathbf{S}_{j0}$. We now claim:

$$F^{n+1}(g_{jn_j}(x));\ a := \text{“}/F\text{”};\ \mathbf{DO}' \ \leqslant\ L := \langle\langle 0, g_{jn_j}(x)\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \mathbf{DO}'$$

To prove this we note that:

$$\begin{aligned}&F^{n+1}(g_{jn_j}(x));\ a := \text{“}/F\text{”};\ \mathbf{DO}'\\ &\quad\approx\quad x := g_{jn_j}(x);\ F^{n+1}(x);\ a := \text{“}/F\text{”};\ \mathbf{DO}'\end{aligned}$$

since, when $a = \text{“}/F\text{”}$, $\mathbf{DO}'$ doesn't use the initial value of x.

$$\approx\quad x := g_{jn_j}(x);\ a := \text{“}A_1\text{”};\ \textsf{while } a \neq \text{“}Z\text{”} \wedge a \neq \text{“}/F\text{”} \textsf{ do } \mathbf{IF}^n \textsf{ od};\ a := \text{“}/F\text{”};\ \mathbf{DO}'$$

by unfolding F^{n+1}.

$$\approx\quad x := g_{jn_j}(x);\ a := \text{“}A_1\text{”};\ \textsf{while } a \neq \text{“}Z\text{”} \wedge a \neq \text{“}/F\text{”} \textsf{ do } \mathbf{IF}^n \textsf{ od};\ \mathbf{DO}'$$

since the **while** loop terminates with $a = \text{“}/F\text{”}$.

$$\approx\quad x := g_{jn_j}(x);\ a := \text{“}A_1\text{”};\ \mathbf{DO}'$$

from the sub-induction result.

$$\approx\quad L := \langle\langle 0, g_{jn_j}(x)\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \mathbf{DO}'$$

by unfolding $\mathbf{DO}'$ and pruning. Finally the assignment to L can be merged with the first assignment to L (which we moved to just after $\mathbf{S}_{j0}$) since F^{n+1} and $\mathbf{S}_{jk}$ do not use L. Continuing in this way we can convert all the calls to F^{n+1} and the intermediate statements into assignments to L to get:

$$\begin{aligned}&\{a = \text{“}B_j\text{”}\};\ \mathbf{S}_{j0};\ F^{n+1}(g_{j0}(x));\ \mathbf{S}_{j1};\ F^{n+1}(g_{j1}(x));\\ &\qquad \ldots F^{n+1}(g_{jn_j}(x));\ \mathbf{S}_{jn_j}[a := \text{“}/F\text{”}/\textsf{call } Z][a := \text{“}X\text{”}/\textsf{call } X];\ \mathbf{DO}'\\ &\leqslant\\ &\{a = \text{“}B_j\text{”}\};\ \mathbf{S}_{j0};\ L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle,\\ &\qquad\qquad \ldots, \langle\langle j, n_j + 1\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \mathbf{DO}'\end{aligned}$$

By unfolding and pruning $\mathbf{DO}'$ we see this is equivalent to $\{a = \text{“}B_j\text{”}\};\ \mathbf{DO}'$ as required.

This completes the proof of the lemma.

From this lemma we deduce the following corollary:

Corollary 4.4 $F(g(x));\ a := \text{“}/F\text{”};\ \textsf{DO}' \leqslant L := \langle\langle 0, g(x)\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \textsf{DO}'$.

Proof: Expanding $F(x)$ in the LHS we have:

$$\begin{aligned}
& F(g(x));\ a := \text{“}/F\text{”};\ \textsf{DO}' \\
& \approx\ x := g(x);\ F(x);\ a := \text{“}/F\text{”};\ \textsf{DO}' \\
& \approx\ x := g(x);\ a := \text{“}A_1\text{”};\ \textsf{while}\ a \neq \text{“}Z\text{”} \wedge a \neq \text{“}/F\text{”}\ \textsf{do IF od};\ a := \text{“}/F\text{”};\ \textsf{DO}' \\
& \approx\ x := g(x);\ a := \text{“}A_1\text{”};\ \textsf{while}\ a \neq \text{“}Z\text{”} \wedge a \neq \text{“}/F\text{”}\ \textsf{do IF od};\ \textsf{DO}'
\end{aligned}$$

since the **while** loop terminates with $a = \text{“}/F\text{”}$.

$$\leqslant x := g(x);\ a := \text{“}A_1\text{”};\ \textsf{DO}'$$

from Lemma(4.3)

$$\leqslant x := g(x);\ L := \langle\langle 0, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \textsf{DO}'$$

by unfolding and pruning $\textsf{DO}'$.

$$\leqslant L := \langle\langle 0, g(x)\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \textsf{DO}'$$

since $\textsf{DO}'$ does not use the value of x when $a = \text{“}/F\text{”}$ initially.

4.2 The Theorem

We now turn our attention to the main theorem. To prove that $F(x) \approx F'(x)$ it is sufficient to prove that $\textsf{DO} \approx \textsf{DO}'$. The proof uses the general induction rule for iteration and the induction rule for recursion.

Lemma 4.5 *For every $n < \omega$:* $\textsf{DO}'^{n} \leqslant \textsf{DO}$:

Proof: By induction on n.

```
DO'^{n+1}
  ⩽ if a ≠ "Z"
      then if a = "/F" →
                if L = ⟨⟩
                  then a := "Z"
                   else ⟨m, x⟩ ← L;
                        if m = 0 → a := "A_1"
                        □ ... □ m = ⟨j, k⟩ → S_jk[a := "/F"/call Z][a := "X"/call X]
                        ... fi fi
            □ a = "A_1" → S_1[a := "/F"/call Z][a := "X"/call X]
            □ ... □ a = "A_i" → S_i[a := "/F"/call Z][a := "X"/call X]
            □ ... □ a = "B_j" → S_j0; L := ⟨⟨0, g_j1(x)⟩, ⟨⟨j, 1⟩, x⟩, ⟨0, g_j2(x)⟩,
                                           ..., ⟨⟨j, n_j + 1⟩, x⟩⟩ ++ L;
                                 a := "/F"
            ... fi;
            DO fi
```

by unfolding and using the induction hypothesis.
Absorb **DO** into the preceding statement. We claim:

$$\begin{aligned}
& L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle\langle j, n_j + 1\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \textsf{DO} \\
& \qquad \approx\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\ F(g_{jn_j}(x));\ \mathbf{S}_{jn_j}[a := \text{“}X\text{”}/\textsf{call}\ X];\ \textsf{DO} \qquad (2)
\end{aligned}$$

Using this claim we can transform each of the $a =$ "B_j" lines to be the same as in **DO**, we then separate **DO** from each arm of the guarded command and use loop rolling to get:

$$\mathsf{DO}'^{n+1} \leqslant \mathsf{DO}$$

as required.

So all that remains is to prove the claim (2). Unroll the first step of **DO** and prune the **if** statements to get:

$$\begin{array}{l}
L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle\langle j, n_j + 1\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”}; \\
\langle m, x\rangle \leftarrow L; \\
\textbf{if}\ m = 0 \rightarrow a := \text{“}A_1\text{”} \\
\Box \ldots \Box\ m = \langle j, k\rangle \rightarrow \mathbf{S}_{jk}[a := \text{“}/F\text{”}/\textbf{call}\ Z][a := \text{“}X\text{”}/\textbf{call}\ X] \\
\ldots\ \textbf{fi}; \\
\textbf{DO}
\end{array}$$

this simplifies to:

$$\begin{array}{l}
L := \langle\langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle\langle j, n_j + 1\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”}; \\
m := 0;\ x := g_{j1}(x); \\
a := \text{“}A_1\text{”}; \\
\textbf{DO}
\end{array}$$

By Lemma (4.2) we get:

$$\begin{array}{l}
L := \langle\langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle\langle j, n_j + 1\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”}; \\
m := 0;\ x := g_{j1}(x); \\
F(x); \\
a := \text{“}/F\text{”}; \\
\textbf{DO}
\end{array}$$

Since $F(x)$ preserves x and does not use L or m we can move the procedure call to the beginning:

$$\begin{array}{l}
F(g_{j1}(x)); \\
L := \langle\langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle\langle j, n_j + 1\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”}; \\
m := 0;\ x := g_{j1}(x); \\
a := \text{“}/F\text{”}; \\
\textbf{DO}
\end{array}$$

Unroll **DO** again and simplify:

$$\begin{array}{l}
F(g_{j1}(x)); \\
L := \langle\langle 0, g_{j2}(x)\rangle, \ldots, \langle\langle j, n_j + 1\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”}; \\
m := \langle j, 1\rangle; \\
\mathbf{S}_{j1}; \\
\textbf{DO}
\end{array}$$

since $\mathbf{S}_{jn_j}$ contains no action calls. Since it also preserves the value of x and doesn't use L or m we can move it to after the procedure call:

$$\begin{array}{l}
F(g_{j1}(x));\ \mathbf{S}_{j1}; \\
L := \langle\langle 0, g_{j2}(x)\rangle, \ldots, \langle\langle j, n_j + 1\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”}; \\
m := \langle j, 1\rangle; \\
a := \text{“}/F\text{”}; \\
\textbf{DO}
\end{array}$$

Continuing in this way we can eliminate all the items pushed onto the front of L. We get:

$$F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \mathbf{S}_{j2};\ \ldots;\ \mathbf{S}_{jn_j};\ F(g_{jn_j}(x));\ \mathbf{S}_{jn_j}[a := \text{“}X\text{”}/\textsf{call } X];$$
$$\textsf{DO}$$

which completes the proof of the lemma.

Hence, by the induction rule for iteration, we have: $\textsf{DO}' \leqslant \textsf{DO}$.

For the converse we prove by induction that $\textsf{DO}^n \leqslant \textsf{DO}'$:

Lemma 4.6 *For every $n < \omega$:* $\textsf{DO}^n \leqslant \textsf{DO}'$

Proof: By induction on n.

$\textsf{DO}^{n+1}$

$\leqslant$ **if** $a \neq \text{“}Z\text{”}$
  then if $a = \text{“}/F\text{”} \rightarrow$
    if $L = \langle\rangle$
      then $a := \text{“}Z\text{”}$
      else $\langle m, x\rangle \leftarrow L;$
        if $m = 0 \rightarrow a := \text{“}A_1\text{”}$
        $\square \ldots \square\ m = \langle j, k\rangle \rightarrow \mathbf{S}_{jk}[a := \text{“}/F\text{”}/\textsf{call } Z][a := \text{“}X\text{”}/\textsf{call } X]$
        $\ldots$ **fi fi**
  $\square\ a = \text{“}A_1\text{”} \rightarrow \mathbf{S}_1[a := \text{“}/F\text{”}/\textsf{call } Z][a := \text{“}X\text{”}/\textsf{call } X]$
  $\square \ldots \square\ a = \text{“}A_i\text{”} \rightarrow \mathbf{S}_i[a := \text{“}/F\text{”}/\textsf{call } Z][a := \text{“}X\text{”}/\textsf{call } X]$
  $\square \ldots \square\ a = \text{“}B_j\text{”} \rightarrow \mathbf{S}_{j0};\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \mathbf{S}_{j2};$
    $\ldots;\ \mathbf{S}_{jn_j};\ F(g_{jn_j}(x));\ \mathbf{S}_{jn_j}[a := \text{“}X\text{”}/\textsf{call } X]$
  $\ldots$ **fi**;
  $\textsf{DO}'$ **fi**

by unfolding and using the induction hypothesis.
Push $\textsf{DO}'$ into each arm of the inner **if** statement. By analogy with the previous part it is sufficient to prove:

$$F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\ F(g_{jn_j}(x));\ \mathbf{S}_{jn_j}[a := \text{“}X\text{”}/\textsf{call } X];\ \textsf{DO}'$$
$$\leqslant L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \textsf{DO}' \quad (3)$$

The statement $\mathbf{S}_{jn_j}[a := \text{“}X\text{”}/\textsf{call } X]$ can be replaced by a push to the stack to get:

$$F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\ F(g_{jn_j}(x));\ L := \langle\langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \textsf{DO}'$$

Move the assignment to L to the beginning:

$$L := \langle\langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\ F(g_{jn_j}(x));\ a := \text{“}/F\text{”};\ \textsf{DO}'$$

Apply Corollary (4.4):

$$L := \langle\langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\ L := \langle\langle 0, g_{jn_j}(x)\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \textsf{DO}'$$

Merge the assignments to L:

$$L := \langle\langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\ a := \text{“}/F\text{”};\ \textsf{DO}'$$

Continuing in this way we can replace the whole sequence of calls and statements by an assignment to L:

$$L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;\ a := \text{“}/F\text{”};\ \textsf{DO}'$$

This completes the proof.

Corollary 4.7 By unfolding some calls to "$/F$" and pruning, we get the following, slightly more efficient, version:

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

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 can be significant for parameterless procedures where the number of j values is small as it can reduce the amount of storage required by the stack. In the extreme case where there is only one j value, the stack reduces to a sequence of identical elements and can therefore be represented by an integer which simply records the length of the stack.

5 Cascade Recursion

This theorem can provide several different iterative equivalents for a given recursive procedure, depending on how the initial restructuring of the procedure body into an action system is carried out. Two extreme cases are:

1. Each action contains no more than one procedure call. This imposes no restrictions on the other statements in the body and is therefore frequently used (for example, many compilers use essentially this approach to deal with recursion). Bird [7] calls this the *direct method*.
2. Each action contains as long a sequence of procedure calls as possible. The resulting iterative program is a simple **while** loop with the stack managing all the control flow. Bird [7] describes this as the *postponed obligations* method: all the sub-invocations arising from a given invocation of the procedure are postponed on the stack before any is fulfilled.

These two special cases of the general transformation will be applied to the following simple cascade recursion schema:

proc $F(x)$ $\equiv$
  if B then T
      else $\mathbf{S}_1;\ F(g_1(x));\ M(x);\ F(g_2(x));\ \mathbf{S}_2$ **fi.**

For the direct method we restructure the body of the procedure into the following action system:

proc $F(x)$ $\equiv$
  actions A_1 :
  $A_1 \equiv$ **if B then T**
            else call B_1 **fi.**
  $B_1 \equiv \mathbf{S}_1;\ F(g_1(x));$ **call** B_2.
  $B_2 \equiv M(x);\ F(g_2(x));$ **call** A_2.
  $A_2 \equiv \mathbf{S}_2;$ **call** Z. **endactions.**

Applying the general recursion removal transformation we get:

proc $F(x)$ $\equiv$
  var $L := \langle\rangle, m := 0$:
    actions A_1 :

```
A1  ≡ if B then T
            else call B1 fi.
B1  ≡ S1; L := ⟨⟨0, g1(x)⟩, ⟨1, x⟩⟩ ++ L; call /F.
B2  ≡ M(x); L := ⟨⟨0, g2(x)⟩, ⟨2, x⟩⟩ ++ L; call /F.
A2  ≡ S2; call /F.
/F  ≡ if L = ⟨⟩ then call Z
                else ⟨m, x⟩ ← L;
                     if m = 0 → call A1
                     □ m = 1 → call B1
                     □ m = 2 → call B2 fi fi. endactions end.
```

The action system is (naturally) regular, so we can apply the transformations in [19] to restructure the action system:

```
proc F(x) ≡
  var L := ⟨⟩, m := 0:
    do while ¬B do S1; L := ⟨⟨1, x⟩⟩ ++ L; x := g1(x) od;
       T;
       do if L = ⟨⟩ then exit(2) fi;
          ⟨m, x⟩ ← L;
          if m = 1 → M(x); L := ⟨⟨2, x⟩⟩ ++ L; x := g2(x); exit
          □ m = 2 → S2 fi od od end.
```

Note that whenever $\langle 0, g_i(x)\rangle$ was pushed onto the stack, it was immediately popped off. So we have avoided pushing $\langle 0, g_i(x)\rangle$ altogether in this version.

For the postponed obligations case we need to structure the initial action system slightly differently:

```
proc F(x) ≡
  actions A :
  A  ≡ if B then T
            else call B fi.
  B  ≡ S1; F(g1(x)); M(x); F(g2(x)); S2; call Z. endactions.
```

Applying the general recursion removal transformation we get:

```
proc F(x) ≡
  var L := ⟨⟩, m := 0:
    actions A :
    A  ≡ if B then T
              else call B fi.
    B  ≡ S1; L := ⟨⟨0, g1(x)⟩, ⟨1, x⟩, ⟨0, g2(x)⟩, ⟨2, x⟩⟩ ++ L; call /F.
    /F  ≡ if L = ⟨⟩ then call Z
                    else ⟨m, x⟩ ← L;
                         if m = 0 → call A
                         □ m = 1 → M(x); call /F
                         □ m = 2 → S2; call /F fi fi. endactions end.
```

This can be expressed as a simple **while** loop thus:

```
proc F(x) ≡
  var L := ⟨⟨0, x⟩⟩, m := 0:
    while L ≠ ⟨⟩ do
          ⟨m, x⟩ ← L;
          if m = 0 → if B then T
                          else S1; L := ⟨⟨0, g1(x)⟩, ⟨1, x⟩, ⟨0, g2(x)⟩, ⟨2, x⟩⟩ ++ L fi
          □ m = 1 → M(x)
```

$\square\ m = 2 \rightarrow \mathbf{S}_2$ **fi od end.**

Alternatively, we can restructure so as to avoid some unnecessary pushes and pops:

proc $F(x)\ \equiv$
  var $L := \langle\rangle, m := 0$:
    do while $\neg\mathbf{B}$ **do** $\mathbf{S}_1;\ L := \langle\langle 1, x\rangle, \langle 0, g_2(x)\rangle, \langle 2, x\rangle\rangle \mathbin{+\!\!+} L;\ x := g_1(x)$ **od**;
      $\mathbf{T}$;
      do if $L = \langle\rangle$ **then exit**(2) **fi**;
        $\langle m, x\rangle \leftarrow L$;
        if $m = 0 \rightarrow$ **exit**
        $\square\ m = 1 \rightarrow M(x)$
        $\square\ m = 2 \rightarrow \mathbf{S}_2$ **fi od od end.**

6 Binary Cascade Recursion

In this section we consider a special case of the cascade recursion above where the functions $g_1(x)$ and $g_2(x)$ return $x - 1$ and the test for a nonrecursive case is simply $n = 0$. Here each invocation of the function leads to either zero or two further invocations, so we use the term *binary* cascade for this schema:

proc $G(n)\ \equiv$
  if $n = 0$ **then** $\mathbf{T}$
      else $\mathbf{S}_1;\ G(n-1);\ M(n);\ G(n-1);\ \mathbf{S}_2$ **fi.**

where $\mathbf{T}$, $\mathbf{S}_1$ and $\mathbf{S}_2$ are statements which do not change the value of n and M is an external procedure.

With this schema, the sequence of statements and calls to M depends only on the initial value of n. We want to determine this sequence explicitly, i.e. we want to determine how many calls of M are executed, what their arguments are and what statements are executed between the calls.

Since the functions g_i are invertable, there is no need to have n as a parameter: we can replace it by a global variable thus:

proc $G\ \equiv$
  if $n = 0$ **then** $\mathbf{T}$
      else $\mathbf{S}_1;\ n := n - 1;\ G;\ M(n+1);\ G;\ n := n + 1;\ \mathbf{S}_2$ **fi.**

It is clear that G preserves the value of n and hence G is equivalent to $G(n)$. We apply the direct method of recursion removal (discussed in the previous Section) to get:

var $L := \langle\rangle, d := 0$:
  do while $n \neq 0$ **do** $\mathbf{S}_1;\ n := n - 1;\ L := \langle 1\rangle \mathbin{+\!\!+} L$ **od**;
    $\mathbf{T}$;
    do if $L = \langle\rangle$ **then exit**(2) **fi**;
      $d \leftarrow L$;
      if $d = 1 \rightarrow M(n+1);\ L := \langle 2\rangle \mathbin{+\!\!+} L;$ **exit**
      $\square\ d = 2 \rightarrow \mathbf{S}_2;\ n := n + 1$ **fi od od end**

Note that since there are no parameters the stack only records control information.

The elements of the stack are either 1 or 2, so we can represent this stack by an integer c whose digits in a binary representation represent the elements of the stack. We need to distinguish an empty stack from a stack of zeros so we use the value 1 to represent the empty stack. The statement $L := \langle 1\rangle \mathbin{+\!\!+} L$ becomes $c := 2.c + 1$, $L := \langle 2\rangle \mathbin{+\!\!+} L$ becomes $c := 2.c$ and $d \leftarrow L$ becomes $\langle d, c\rangle := \langle c \div 2\rangle$. With this representation, the translation of **while** $n \neq 0$ **do** $S_1;\ n := n-1;\ L := \langle 1\rangle \mathbin{+\!\!+} L$ **od** which pushes n 1's onto L has the effect of multiplying c by 2^n and adding $2^n - 1$ to the result. We get:

```
var c := 1, d := 0:
  do for i := n step −1 to 1 do S1[i/n] od;
     c := 2^n.c + 2^n − 1; n := 0;
     T;
     do if c = 1 then exit(2) fi;
        ⟨d, c⟩ := ⟨c ÷ 2⟩;
        if d = 1 → M(n + 1); c := 2.c; exit
        □ d = 0 → S2; n := n + 1 fi od od end
```

The variable d is not really needed as we can simply test whether c is odd or even, if it is odd then the effect of $\langle c, d\rangle := \langle c \div 2\rangle$ followed by $c := 2.c$ is to subtract one from c:

```
var c := 1:
  do for i := n step −1 to 1 do S1[i/n] od;
     c := 2^n(c + 1) − 1; n := 0;
     T;
     do if c = 1 then exit(2) fi;
        if odd(c) then M(n + 1); c := c − 1; exit
                  else S2; c := c/2; n := n + 1 fi od od end
```

Take the first three lines out of the outer loop and merge them into the inner loop, this results in a simple double loop which can be reduced to a single loop:

```
var c := 2^(n+1) − 1:
  for i := n step −1 to 1 do S1[i/n] od;
  n := 0; T;
  do if c = 1 then exit fi;
     if odd(c) then M(n + 1); c := c − 1;
                    for i := n step −1 to 1 do S1[i/n] od;
                    c := 2^n(c + 1) − 1; n := 0;
                    T
               else S2; c := c/2; n := n + 1 fi od end
```

The body of this loop is split into two cases (for odd or even c), where one case (for even c) is much simpler than the other. This suggests a partial entire loop unrolling under the case $\mathsf{even}(c)$:

```
var c := 2^(n+1) − 1:
  for i := n step −1 to 1 do S1[i/n] od;
  n := 0; T;
  do if c = 1 then exit fi;
     if odd(c) then M(n + 1); c := c − 1;
                    for i := n step −1 to 1 do S1[i/n] od;
                    c := 2^n(c + 1) − 1; n := 0;
                    T;
                    while even(c) do S2; c := c/2; n := n + 1 od
               else S2; c := c/2; n := n + 1 fi od end
```

If c is odd then the execution of the loop body will leave c odd. Since c is initially odd ($2^{n+1} - 1$ is odd for all $n \geqslant 0$) we can eliminate the test $\mathsf{odd}(c)$:

```
var c := 2^(n+1) − 1:
  for i := n step −1 to 1 do S1[i/n] od;
  n := 0; T;
  do if c = 1 then exit fi;
     M(n + 1); c := c − 1;
     for i := n step −1 to 1 do S1[i/n] od;
     c := 2^n(c + 1) − 1; n := 0;
```

```
    T;
    while even(c) do S_2; c := c/2; n := n + 1 od od end
```

Since c is initially odd, the **while** loop can be moved to the beginning of the loop. The effect of this **while** loop is to divide c by $2^{\mathrm{ntz}(c)}$ where $\mathrm{ntz}(c)$ is the number of trailing zeros in the binary expansion of c. Since n is set to zero just before the loop, it also sets n to $\mathrm{ntz}(c)$:

```
var c := 2^{n+1} - 1:
  for i := n step -1 to 1 do S_1[i/n] od;
  T;
  do n := ntz(c); c := c/2^n;
      for i := 0 step 1 to n - 1 do S_2[i/n] od;
      if c = 1 then exit fi;
      M(n + 1);
      for i := n step -1 to 1 do S_1[i/n] od;
      c := 2^n.c - 1;
      T od end
```

The test $c = 1$ will be true only if the binary expansion of c at the beginning of the loop contained exactly one 1. If this is not the case then the effect of the loop is to decrement c by one. This means that the loop continues until the most significant bit of the initial value of c is the only bit set, ie $c = 2^{n_0}$ where n_0 is the initial value of n. Thus we can convert the loop to a **while** loop:

```
var c := 2^{n+1} - 1, n_0 := n:
  for i := n step -1 to 1 do S_1[i/n] od;
  T;
  while c ≠ 2^{n_0} do
        n := ntz(c); c := c/2^n;
        for i := 0 step 1 to n - 1 do S_2[i/n] od;
        M(n + 1);
        for i := n step -1 to 1 do S_1[i/n] od;
        c := 2^n.c - 1; n := 0;
        T od;
  n := ntz(c); c := c/2^n;
  for i := 0 step 1 to n - 1 do S_2[i/n] od end
```

The final values of the local variables c and n do not affect the result of the program; so we can simplify this to:

```
var c := 2^{n+1} - 1, n_0 := n:
  for i := n step -1 to 1 do S_1[i/n] od;
  T;
  while c ≠ 2^{n_0} do
        n := ntz(c);
        for i := 0 step 1 to n - 1 do S_2[i/n] od;
        M(n + 1);
        for i := n step -1 to 1 do S_1[i/n] od;
        c := c - 1;
        T od;
  for i := 0 step 1 to n_0 - 1 do S_2[i/n] od end
```

Note that for $2^{n_0} < c < 2^{n_0+1}$, $\mathrm{ntz}(c) = \mathrm{ntz}(2^{n_0+1} - c)$ so we can represent c by c' where $c' = 2^{n_0+1} - c$. Writing the loop as a **for** loop (and putting c back for c') we get the final version:

```
var n_0 := n:
  for i := n step -1 to 1 do S_1[i/n] od;
  T;
```

```
  for c := 1 step 1 to 2^{n_0} - 1 do
      n := ntz(c);
      for i := 0 step 1 to n - 1 do S_2[i/n] od;
      M(n + 1);
      for i := n step -1 to 1 do S_1[i/n] od;
      T od;
  for i := 0 step 1 to n_0 - 1 do S_2[i/n] od end
```

For the case where $\mathbf{S}_1$ and $\mathbf{S}_2$ are both **skip** this simplifies to:

```
T;
for c := 1 step 1 to 2^n - 1 do
    M(ntz(c) + 1);
    T od;
```

7 Example: The Gray Code

An n-bit gray code is a sequence of 2^n n-bit binary numbers (sequences of 0's and 1's of length n) starting from $00\ldots0$ such that each element of the sequence differs from the next in a single bit position (and the 2^nth element has a single bit set). We want to define a function $g(n)$ which returns an n-bit gray code. For $n = 0$ the gray code is the one element sequence $\langle\langle\rangle\rangle$. Note that there are several different n-bit gray codes for $n > 1$: the problem of finding all gray codes of a given length is equivalent to finding all the Hamiltonian cycles of a n-dimensional unit hypercube.

So suppose we have $g(n-1)$ and want to construct $g(n)$. The elements of $g(n-1)$ will be $n-1$ bit codes; hence $(\langle 0\rangle +\!\!+) * g(n-1)$ and $(\langle 1\rangle +\!\!+) * g(n-1)$ are disjoint gray code sequences of length 2^{n-1}. Their corresponding elements differ in only the first bit position, in particular the last element of $(\langle 0\rangle +\!\!+) * g(n-1)$ differs from the last element of $(\langle 1\rangle +\!\!+) * g(n-1)$ in one bit position. Thus if we reverse the sequence $(\langle 1\rangle +\!\!+) * g(n-1)$ and append it to $(\langle 0\rangle +\!\!+) * g(n-1)$ we will form an n-bit gray code. Thus the definition of $g(n)$ is:

```
funct g(n) ≡
  if n = 0 then ⟨⟨⟩⟩
           else (⟨0⟩ ++) * g(n - 1) ++ reverse((⟨0⟩ ++) * g(n - 1)) fi.
```

This function defines $g(n)$ in terms of $g(n-1)$ and $\mathsf{reverse}(g(n-1))$: this suggests we define $g(n)$ in terms of a function $g'(n,s)$ such that $g'(n,0) = g(n)$ and $g'(n,1) = \mathsf{reverse}(g(n))$. Note that $\mathsf{reverse}(g(n)) = (\langle 1\rangle +\!\!+) * g(n-1) +\!\!+ (\langle 0\rangle +\!\!+) * \mathsf{reverse}(g(n-1))$. So we can define $g'(n,s)$ as follows:

```
funct g'(n, s) ≡
  if n = 0 then ⟨⟨⟩⟩
           else (⟨s⟩ ++) * g'(n - 1, 0) ++ (⟨1 - s⟩ ++) * g'(n - 1, 1) fi.
```

Finally, instead of computing $g'(n-1,s)$ and appending either $\langle 0\rangle$ or $\langle 1\rangle$ to each element, we can pass a third argument which is to be appended to each element of the result; i.e. define $g''(L,n,s) = (L +\!\!+) * g'(n,s)$. We get the following definition of g'':

```
funct g''(L, n, s) ≡
  if n = 0 then ⟨L⟩
           else g''(⟨s⟩ ++ L, n - 1, 0) ++ g''(⟨1 - s⟩ ++ L, n - 1, 1) fi.
```

The recursive case of this version simply appends the results of the two recursive calls. This suggests we use a procedural equivalent which appends the result to a global variable r. Thus our gray code function $g(n)$ is equivalent to:

```
funct g(N) ≡
  ⌈r := ⟨⟩:
   begin
```

```
    G(⟨⟩, N, 0)
  where
  proc G(L, n, s) ≡
    if n = 0 then r := r ++ ⟨L⟩
             else G(⟨s⟩ ++ L, n − 1, 0); G(⟨1 − s⟩ ++ L, n − 1, 1) fi. end;
  r⌋.
```

Represent the stack L of bits as an integer c as in Section 6:

```
begin
  G(1, N, 0)
where
proc G(c, n, s) ≡
  if n = 0 then r := r ++ ⟨bits(c)⟩
           else G(2.c + s, n − 1, 0); G(2.c + 1 − s, n − 1, 1) fi. end
```

where bits(c) returns the sequence of bits represented by the integer c. We can combine c and s into one argument c' where $c' = 2.c + s$:

```
begin
  G(2, N)
where
proc G(c', n) ≡
  if n = 0 then r := r ++ ⟨bits(⌊c'/2⌋)⟩
           else G(2.c', n − 1); G(2.(c' ⊕ 1) ⊕ 1, n − 1) fi. end
```

where $a \oplus b$ is a "bitwise exclusive or" operator. Note that we always double c' whenever we decrement n; this suggests representing c' by c where $c = c'.2^n$:

```
begin
  G(2^(N+1), N)
where
proc G(c, n) ≡
  if n = 0 then r := r ++ ⟨bits(⌊c/2⌋)⟩
           else G(c, n − 1); G((c ⊕ 2^n) ⊕ 2^(n−1), n − 1) fi. end
```

We want to replace c by a global variable c'. To do this we add c' as a new ghost variable; we assign values to c' which track the current value of c:

```
begin var c' := 2^(N+1):
        G(2^(N+1), N)
  where
  proc G(c, n) ≡
    if n = 0 then r := r ++ ⟨bits(⌊c/2⌋)⟩; c' := c' ⊕ 1
             else G(c, n − 1); c' := c' ⊕ 2^n; G((c ⊕ 2^n) ⊕ 2^(n−1), n − 1) fi. end end
```

By induction on n we prove: $\{c' = c\};\ G(c, n-1) \leqslant \{c' = c\};\ G(c, n-1);\ \{c' = c \oplus 2^n\}$. Then at every call of G we have $c' = c$ so we can replace the parameter c by the global variable c':

```
begin var c' := 2^(N+1):
        G(N)
  where
  proc G(n) ≡
    if n = 0 then r := r ++ ⟨bits(⌊c'/2⌋)⟩; c' := c' ⊕ 1
             else G(n − 1); c' := c' ⊕ 2^n; G(n − 1) fi. end end
```

Now we have a standard binary cascade recursion for which the transformation of Section 6 gives:

```
begin var c' := 2^(N+1):
        r := r ++ ⟨bits(⌊c'/2⌋)⟩;
```

```
    for i := 1 step 1 to 2^N − 1 do
        c' := c' ⊕ 2^ntz(i)+1;
        r := r ++ ⟨bits(⌊c'/2⌋)⟩ od end end
```

Finally, the least significant bit of c' is always ignored and the most significant bit of c' is always the 2^{N+1} bit so we can represent c' by $c = \lfloor (c' - 2^{N+1})/2 \rfloor$:

```
begin var c := 0:
    r := r ++ ⟨Nbits(c)⟩;
    for i := 1 step 1 to 2^N − 1 do
        c := c ⊕ 2^ntz(i);
        r := r ++ ⟨Nbits(c)⟩ od end end
```

where $\mathsf{Nbits}(c) = \mathsf{bits}(c + 2^{N+1})$.

Thus, the bit which changes between the ith and $(i+1)$th codes is the bit in position $\mathrm{ntz}(i)$. From this result we can prove the following:

Theorem 7.1 *The ith gray code is $i \oplus \lfloor i/2 \rfloor$.*

Proof: The proof is by induction on i. Suppose $c = i \oplus \lfloor i/2 \rfloor$ is the ith gray code. Then from the program above, the $(i+1)$th gray code is $c \oplus 2^{\mathrm{ntz}(i+1)}$. The number of trailing zeros in the binary representation of $i+1$ is simply the number of trailing ones in the binary representation of i. Suppose i is even (i.e. there are no trailing ones) and it's N-bit binary representation is $\langle i_1, i_2, \ldots, i_{n-1}, 0 \rangle$. Then:

$$\begin{array}{rcl}
i & = & \langle\ i_1, \quad i_2, \quad \ldots, \quad i_{n-1}, \quad 0 \quad \rangle \\
\lfloor i/2 \rfloor & = & \langle\ 0, \quad i_1, \quad \ldots, \quad i_{n-2}, \quad i_{n-1} \quad \rangle \\
i \oplus \lfloor i/2 \rfloor & = & \langle\ i_1, \quad i_2 \oplus i_1, \ldots, i_{n-1} \oplus i_{n-2}, \quad i_{n-1} \quad \rangle \\
i+1 & = & \langle\ i_1, \quad i_2, \quad \ldots, \quad i_{n-1}, \quad 1 \quad \rangle \\
\lfloor (i+1)/2 \rfloor & = & \langle\ 0, \quad i_1, \quad \ldots, \quad i_{n-2}, \quad i_{n-1} \quad \rangle \\
(i+1) \oplus \lfloor (i+1)/2 \rfloor & = & \langle\ i_1, \quad i_2 \oplus i_1, \ldots, i_{n-1} \oplus i_{n-2}, \quad i_{n-1} \oplus 1 \ \rangle
\end{array}$$

i.e. the 2^0 bit has changed.

On the other hand, suppose i is odd and has k trailing ones with $k > 0$. Since $(i+1) < 2^n$ we must have $k < n$. So:

$$\begin{array}{rcl}
i & = & \langle\ i_1, \quad i_2, \quad \ldots, \quad i_{n-k-1}, \quad 0, \quad 1, \quad 1, \ldots, 1 \ \rangle \\
\lfloor i/2 \rfloor & = & \langle\ 0, \quad i_1, \quad \ldots, \quad i_{n-k-2}, \quad i_{n-k-1}, \quad 0, \quad 1, \ldots, 1 \ \rangle \\
i \oplus \lfloor i/2 \rfloor & = & \langle\ i_1, \quad i_2 \oplus i_1, \ldots, i_{n-k-1} \oplus i_{n-k-2}, \quad i_{n-k-1}, \quad 1, \quad 0, \ldots, 0 \ \rangle \\
i+1 & = & \langle\ i_1, \quad i_2, \quad \ldots, \quad i_{n-k-1}, \quad 1, \quad 0, \quad 0, \ldots, 0 \ \rangle \\
\lfloor (i+1)/2 \rfloor & = & \langle\ 0, \quad i_1, \quad \ldots, \quad i_{n-k-2}, \quad i_{n-k-1}, \quad 1, \quad 0, \ldots, 0 \ \rangle \\
(i+1) \oplus \lfloor (i+1)/2 \rfloor & = & \langle\ i_1 \quad i_2 \oplus i_1 \ldots, i_{n-k-1} \oplus i_{n-k-2} \quad i_{n-k-1} \oplus 1 \quad 1 \quad 0 \ldots, 0 \ \rangle
\end{array}$$

i.e. the 2^k bit has changed.

Which proves the theorem

From this we derive the following gray code generator:

```
funct g(n) ≡
  ⌈r := ⟨Nbits(0)⟩:
   for i := 1 step 1 to 2^n − 1 do
       r := r ++ ⟨Nbits(i ⊕ ⌊i/2⌋)⟩ od;
   r⌋.
```

While the previous gray code generator only told us which bit changes from one code to the next, this one calculates the ith gray code directly from i without using any previous codes.

8 Towers of Hanoi

The well-known "Towers of Hanoi" algorithm is a simple binary cascade recursion. The procedure $H(n, a, b, c)$ moves a stack of n different-sized disks from peg a to peg b using peg c without ever placing a larger disk upon a smaller(where a, b and c are some permutation of the values $\{0, 1, 2\}$):

proc $H(n, a, b, c)$ $\equiv$
  if $n = 0$ **then skip**
    else $H(n-1, a, c, b)$; $\mathsf{move}(n, a, b)$; $H(n-1, c, b, a)$ **fi.**

where the procedure $\mathsf{move}(n, a, b)$ moves the disk numbered n from disk a to disk b.

The values the parameters a, b and c are always a permutation of their initial values, so we can replace them by global variables:

proc $H(n)$ $\equiv$
  if $n = 0$ **then skip**
    else $\langle b, c\rangle := \langle c, b\rangle$; $H(n-1, a)$; $\langle b, c\rangle := \langle c, b\rangle$;
      $\mathsf{move}(n, a, b)$;
      $\langle a, c\rangle := \langle c, a\rangle$; $H(n-1)$; $\langle a, c\rangle := \langle c, a\rangle$ **fi.**

Applying the binary cascade recursion transformation we get:

proc $H(n)$ $\equiv$
  if $\mathsf{odd}(n)$ **then** $\langle b, c\rangle := \langle c, b\rangle$ **fi**;
  for $c := 1$ **to** $2^n - 1$ **do**
    if $\mathsf{odd}(\mathrm{ntz}(c))$ **then** $\langle a, c\rangle := \langle c, a\rangle$ **fi**;
    $\langle b, c\rangle := \langle c, b\rangle$; $\mathsf{move}(\mathrm{ntz}(c) + 1, a, b)$; $\langle a, c\rangle := \langle c, a\rangle$;
    if $\mathsf{odd}(\mathrm{ntz}(c))$ **then** $\langle b, c\rangle := \langle c, b\rangle$ **fi od**;
  if $\mathsf{odd}(n)$ **then** $\langle a, c\rangle := \langle c, a\rangle$ **fi.**

Hence the sequence in which disks are moved in the Towers of Hanoi problem is the same as the sequence in which bits are changed in the generation of a gray code.

It is often worth trying to simplify a recursive procedure as much as possible *before* removing the recursion. In this case we would like to be able to determine the parameters a and b for the move procedure from c alone since this will greatly simplify the iterative version of the program. Hand simulation for small values of n suggests that each disk always moves in the same direction: either forwards ($0 \to 1 \to 2 \to 0$) or backwards ($0 \to 2 \to 1 \to 0$) with disk k moving forwards if $N - k$ is odd (where N is the initial value of n and $a = 0$, $b = 1$ and $c = 2$ initially). This fact is rather tricky to prove from the iterative version of the program, but for the recursive procedure the proof is a simple induction on n. From the iterative program we know that before move c, disk number k will have moved $m(k, c)$ times where

$$m(k, c) = \#\{\, i \mid 1 \leqslant i < c \ \wedge\ \mathrm{ntz}(i) + 1 = k \,\}$$

Now $\mathrm{ntz}(i) + 1 = k$ iff $i = j.2^{k-1}$ for some odd j, so:

$$\begin{aligned} m(k, c) &= \#\left\{\, j \mid 0 < j.2^{k-1} < c \ \wedge\ \mathsf{odd}(j) \,\right\} \\ &= \#\left\{\, j \mid 0 < j < \lfloor c/2^{k-1} \rfloor \ \wedge\ \mathsf{odd}(j) \,\right\} \\ &= \#\left\{\, j \mid 0 < j < \lfloor (\lfloor c/2^{k-1} \rfloor)/2 \rfloor \,\right\} \\ &= \lfloor (\lfloor c/2^{k-1} \rfloor)/2 \rfloor \end{aligned}$$

So after move c the kth disk will have moved to position $(m(c, k) \bmod 3)$ or $(-m(c, k) \bmod 3)$ depending on whether it is moving forwards or backwards. Thus the position of disk k before move c is $p(c, k, N) = ((-1)^{N-k+1} m(c, k) \bmod 3)$. Thus given c and N we can immediately determine the positions of all the disks, the next disk to be moved (it is number $\mathrm{ntz}(c) + 1$), and the peg it moves to.

So our final version of the program is:

```
proc H(n) ≡
  for c := 1 to 2^n − 1 do
      move(ntz(c) + 1, p(c, ntz(c) + 1, n), p(c + 1, ntz(c) + 1, n)); od.
```

9 Program Analysis

Since the recursion removal theorem can be applied in either direction, and because it places so few restrictions on the form of the program, it can be applied in the reverse direction as a program analysis or reverse engineering tool to 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) ≡
  ⌈begin 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 sent by the author as a "challenge" to the REDO group at the Programming Research group in Oxford to test their proposed methods for formal reverse engineering of source code. Their paper [9] required eight pages of careful reasoning plus some "inspiration" to analyse 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 (this stage could easily be automated);
2. Apply the theorem;
3. Restructure the resulting recursive procedure in a functional form (this stage could also be automated).

If we examine the operations carried out on the stack we see that only constant elements are pushed onto the stack, the program terminated when the stack becomes empty, and the value popped off the stack is used to determine the control flow. This suggests that we may be able to remove the stack and re-express the control flow explicitly using our theorem. The first step is to restructure the loops into an action system and collect together the "stack push" operations into separate actions:

```
var d := 0, stack := ⟨⟩:
  actions A1 :
  A1 ≡ if m = 0 then n := n + 1; call /A
       elsif n = 0 then call B1
                   else call B2 fi.
  B1 ≡ stack := ⟨1⟩ ++ stack; m := m − 1; n := 1; call A1.
  B2 ≡ stack := ⟨0⟩ ++ stack; n := n − 1; call A1.
  /A ≡ if stack = ⟨⟩ then call Z
                  else d ← stack;
                       if d = 0 then call B3
                                else m := m + 1; call /A fi fi.
  B3 ≡ stack := ⟨1⟩ ++ stack; m := m − 1; call A1.
  endactions end
```

This is already almost in the right form to apply the Corollary (4.7) above, all we need to do is to move the stack assignments past the assignments to n and m:

```
var d := 0, stack := ⟨⟩:
  actions A1 :
  A1  ≡ if m = 0 then n := n + 1; call /A
        elsif n = 0 then call B1
                    else call B2 fi.
  B1  ≡ m := m − 1; n := 1; stack := ⟨1⟩ ++ stack; call A1.
  B2  ≡ n := n − 1; stack := ⟨0⟩ ++ stack; call A1.
  /A  ≡ if stack = ⟨⟩ then call Z
                  else d ← stack;
                       if d = 0 then call B3
                                else m := m + 1; call /A fi fi.
  B3  ≡ m := m − 1; stack := ⟨1⟩ ++ stack; call A1.
  endactions end
```

Apply the transformation in Corollary (4.7) to get the recursive version:

```
proc F ≡
  actions A1 :
  A1  ≡ if m = 0 then n := n + 1; call Z
        elsif n = 0 then call B1
                    else call B2 fi.
  B1  ≡ m := m − 1; n := 1; F; m := m + 1; call Z.
  B2  ≡ n := n − 1; F; call B3.
  B3  ≡ 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:

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

Substituting this into the original function we 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.

This is the famous Ackermann function [1].

10 Conclusion

The program transformation theory used in this paper forms the foundation of the "Maintainer's Assistant" project [10,22] at Durham University and the Centre for Software Maintenance Ltd. which aims to produce a prototype tool to assist a maintenance programmer to understand and modify an initially unfamiliar program, given only the source code. The tool consists of a structure editor, a library of proven transformations and a knowledge-based system which analyses the programs and specifications under consideration and uses heuristic knowledge to determine which transformations will achieve a given end (for example, deriving the specification of a section of code, finding the most suitable technique for recursion removal, optimising for efficiency etc.) Part of the project involves building front ends for IBM Assembler and **Z** specifications with the aim of analysing Assembler programs and turning them into equivalent high-level language programs and **Z** specifications.

11 References

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

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

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

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

[5] 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).

[6] F. L. Bauer & H. Wossner, *Algorithmic Language and Program Development*, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

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

[8] R. Bird, "Lectures on Constructive Functional Programming," Oxford University, Technical Monograph PRG-69, Sept., 1988.

[9] P. T. Breuer, K. Lano & J. Bowen, "Understanding Programs through Formal Methods," Oxford University, Programming Research Group, 9 Apr., 1991.

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

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

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

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

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

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

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

[17] E. L. Post, "Formal Reduction of the General Combinatorial Decision Problem," *Amer. J. Math.* (1943).

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

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

[20] M. Ward, "Specifications and Programs in a Wide Spectrum Language," Submitted to J. Assoc. Comput. Mach., 1991.

[21] M. Ward, "A Model for Partial Programs," Submitted to J. Assoc. Comput. Mach., Nov., 1989.

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