A Recursion Removal Theorem

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

July 16, 1993

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. Recursion removal often forms an important step in the systematic development of an algorithm from a formal specification. We use semantic-preserving transformations to carry out such developments and the theorem proves the correctness of many different classes of recursion removal. This theorem includes as special cases the two techniques discussed by Knuth [13] and Bird [7]. We describe some applications of the theorem to cascade recursion, binary cascade recursion, Gray codes, 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 [20,26,26] 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 $\mathbf{S}_1 \leq \mathbf{S}_2$ if $\mathbf{S}_2$ is more defined and more deterministic than $\mathbf{S}_1$. If $\mathbf{S}_1 \leq \mathbf{S}_2$ and $\mathbf{S}_2 \leq \mathbf{S}_1$ 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. Thus a transformation is a special case of refinement. See [20] and [26] for a description of the semantics of WSL and the methods used for proving the correctness of transformations.

Many of the transformations of WSL programs are "refinements" in the wider sense of transforming an abstract specification or algorithm into a concrete implementation; in [22] we discuss ways of defining the relative "degree of abstractness" of semantically equivalent WSL programs.

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, with primitive statements to add and remove variables to/from the state space. Together with guards and assertions, this allows specifications expressed in first order logic to be 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, not only to the goal of refining specifications into algorithms by formal transformation (see [21,25,27]), but also working in the reverse direction: starting with an unstructured program we can transform it into a high-level specification [24].

2 The Wide Spectrum Language

Our kernel language has four primitive statements:

1. Assertion: $\{\mathbf{P}\}$
2. Guard: $[\mathbf{P}]$
3. Add variables (with arbitrary values): $\mathsf{add}(\mathbf{x})$
4. Remove variables: $\mathsf{remove}(\mathbf{x})$

where $\mathbf{x}$ is a sequence of variables and $\mathbf{P}$ a formula of first order logic.

An assertion is a partial **skip** statement, it aborts if the condition is false but does nothing if the condition is true. The **abort** statement $\{\mathbf{false}\}$ therefore always aborts. The guard statement $[\mathbf{P}]$ always terminates, it enforces $\mathbf{P}$ to be true at this point in the program.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", $[\mathbf{false}]$, is a "correct refinement" of *any* specification whatsoever. Clearly guard statements cannot be directly implemented but they are nonetheless a useful theoretical tool.

The $\mathsf{add}(\mathbf{x})$ statement is unrestricted in its nondeterminacy, by following it with a suitable guard we can restrict the nondeterminacy and achieve the effect of a general assignment. For example, Back's atomic description [4], written $\mathbf{x}/\mathbf{y}.\mathbf{Q}$, where $\mathbf{Q}$ is a formula of first order logic (with equality) and $\mathbf{x}$ and $\mathbf{y}$ are sets of variables, is equivalent to the sequence $\{\exists \mathbf{x}.\, \mathbf{Q}\}$; $\mathsf{add}(\mathbf{x})$; $[\mathbf{Q}]$; $\mathsf{remove}(\mathbf{y})$. 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.

Morgan and others [14,15,16,17] use a different specification statement, written $\mathbf{x}$: [Pre, 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 $\{\mathsf{Pre}\}$; $\mathsf{add}(\mathbf{x})$; [Post].

The kernel language is constructed from these four 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 four 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. It is the strongest program refined by both $\mathbf{S}_1$ and $\mathbf{S}_2$.

3. **Join:** $(\mathbf{S}_1 \sqcup \mathbf{S}_2)$
   The join of two programs is the weakest program which refines them both.

4. **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.

There is a rather pleasing duality between the assertion and the guard and also between the choice and join constructs. In fact, the set of programs forms a *lattice* [11] with [**false**] as the top element, {**false**} as the bottom element, $\sqcap$ as the lattice meet and $\sqcup$ as the lattice join operators.

The kernel language 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 transformation 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 four hundred transformations, accessible through a simple user interface [10].

2.1 Syntax of Expressions

Expressions include variable names, numbers, strings of the form "`text...`", the constants $\mathbb{N}$, $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, and the following operators and functions: (in the following e_1, e_2 etc. represent any valid expressions):

Numeric operators: $e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$, e_1/e_2, $e_1^{e_2}$, e_1 mod e_2, e_1 div e_2, frac(e_1), abs(e_1), sgn(e_1), max$(e_1, e_2, \ldots)$, min$(e_1, e_2, \ldots)$, with the usual meanings.

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 $s[i..\ell(s)]$. reverse$(s) = \langle a_n, a_{n-1}, \ldots, a_2, a_1 \rangle$, head(s) is the same as s[1] and tail(s) is s[2..].

Sequence concatenation: $s_1 +\!\!+ s_2 = \langle s_1[1], \ldots, s_1[\ell(s_1)], s_2[1], \ldots, s_2[\ell(s_2)] \rangle$. The append function, append$(s_1, s_2, \ldots, s_n)$, is the same as $s_1 +\!\!+ s_2 +\!\!+ \cdots +\!\!+ s_n$.

Subsequences: The assignment s[i..j] := t[k..l] where $j - i = l - k$ assigns s the value $\langle s[1], \ldots, s[i-1], t[k], \ldots, t[l], s[j+1], \ldots, s[\ell(s)] \rangle$.

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

Relations and Functions: A relation is a (finite or infinite) set of pairs, a subset of $A \times B$ where A is the domain and B the range. A relation f is a function iff $\forall x, y_1, y_2.\ \Big(\big((x, y_1) \in f \ \wedge\ (x, y_2) \in f\big) \Rightarrow y_1 = y_2\Big)$. In this case we write $f(x) = y$ when $(x, y) \in f$.

Substitution: The expression $e[e_2/e_1]$ where e, e1 and e2are expressions means the result of replacing all occurrences of e_1 in e by e_2. (This notation is also used for substitution in statements).

2.2 Syntax of Formulae

true and **false** are true and false conditions, **true** is defined as $\forall v.(v = v)$ and **false** as $\neg\forall v.(v = v)$. In the following $\mathbf{Q}$, $\mathbf{Q}_1$, $\mathbf{Q}_2$ etc. represent arbitrary formulae and e_1, e_2, etc. arbitrary expressions:

Relations: $e_1 = e_2$, $e_1 \neq e_2$, $e_1 < e_2$, $e_1 \leq e_2$, $e_1 > e_2$, $e_1 \geq e_2$, even?(e_1), odd?(e_1);

Logical operators: $\neg\mathbf{Q}$, $\mathbf{Q}_1 \vee \mathbf{Q}_2$, $\mathbf{Q}_1 \wedge \mathbf{Q}_2$;

Quantifiers: $\forall \mathbf{v}.\,\mathbf{Q}$ and $\exists \mathbf{v}.\,\mathbf{Q}$ are allowed in formulae.

2.3 Language Extensions

The first set of language extensions, which go to make up the "first level" language, are as follows. 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 [26] and [20] 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:

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

- Assignment: We can express a general assignment using add, remove, and guards:

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q} \quad =_{\mathrm{DF}} \quad \{\exists \mathbf{x}.\,\mathbf{Q}\};\ \mathsf{add}(\mathbf{x}');\ [\mathbf{Q}];\mathsf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'];\ \mathsf{remove}(\mathbf{x}')$$

  Here, $\mathbf{x}$ is a sequence of variables and $\mathbf{x}'$ is a sequence of new variables. The formula $\mathbf{Q}$ expresses the relation between the initial values of $\mathbf{x}$ and the final values. For example: $\langle x\rangle := \langle x'\rangle.(x' = x + 1)$ increments the value of the variable x. We will sometimes omit the sequence brackets around singleton sequences of variables and expressions where this causes no confusion.

- Simple Assignment: If $\mathbf{e}$ is a list of expressions and $\mathbf{x}$ a list of variables and $\mathbf{x}'$ a list of new variables, then:

$$\mathbf{x} := \mathbf{e} \quad =_{\mathrm{DF}} \quad \mathbf{x} := \mathbf{x}'.(\mathbf{x}' = \mathbf{e})$$

  With this notation, the statement to increment x can be written: $x := x + 1$ (omitting the sequence brackets as discussed above).

- Stack Operations:

$$\begin{aligned} x \leftarrow e \quad &=_{\mathrm{DF}} \quad x := \langle e\rangle \mathbin{+\!\!+} x \\ x \xleftarrow{\text{push}} e \quad &=_{\mathrm{DF}} \quad x := \langle e\rangle \mathbin{+\!\!+} x \\ x \xleftarrow{\text{pop}} e \quad &=_{\mathrm{DF}} \quad e := x[1];\ x := x[2\,..] \end{aligned}$$

- Nondeterministic Choice: The "guarded command" of Dijkstra [12]:

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

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

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

- Nondeterministic Iteration:

$$\begin{array}{l} \textbf{do}\ B_1 \rightarrow S_1 \\ \quad \Box\ B_2 \rightarrow S_2 \\ \quad \ldots \\ \quad \Box\ B_n \rightarrow S_n\ \textbf{od} \end{array} \quad =_{DF} \quad \begin{array}{l} \textbf{while}\ (B_1 \vee B_2 \vee \cdots \vee B_n)\ \textbf{do} \\ \quad \textbf{if}\ B_1 \rightarrow S_1 \\ \quad \Box\ B_2 \rightarrow S_2 \\ \quad \ldots \\ \quad \Box\ B_n \rightarrow S_n\ \textbf{fi}\ \textbf{od} \end{array}$$

- Initialised local Variables:

$$\textbf{var}\ x := t : S\ \textbf{end} \quad =_{DF} \quad \mathsf{add}(x);\ [x = t];\ S;\ \mathsf{remove}(x)$$

- Counted Iteration:

$$\begin{array}{l} \textbf{for}\ i := b\ \textbf{to}\ f\ \textbf{step}\ s\ \textbf{do}\ S\ \textbf{od} \end{array} \quad =_{DF} \quad \begin{array}{l} \textbf{var}\ i := b: \\ \quad \textbf{while}\ i \leq f\ \textbf{do} \\ \quad\quad S;\ i := i + s\ \textbf{od}\ \textbf{end} \end{array}$$

- Procedure call:

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

- Block with local procedure:

$$\textbf{begin}\ S_1\ \textbf{where}\ \textbf{proc}\ X \equiv S_2.\ \textbf{end} \quad =_{DF} \quad S_1[\textbf{proc}\ X \equiv S_2./X]$$

2.4 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 [13] and more recently in [19]. 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 n of the 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:

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

where n is an integer constant representing the depth of the loop (1 for an outermost loop, 2 for double nested loops etc.) and $\text{guard}_n(\mathbf{S})$ is the statement $\mathbf{S}$ 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 if depth $\neq n$ initially then $\text{guard}_n(\mathbf{S}) \approx \mathbf{skip}$. 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 fi od;
       if depth = 1 then count := number[i] fi od
```

2.5 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 $\mathbf{Z}$, 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** $x = 1 \rightarrow \mathbf{S}_1[x := 1;\ A/X_1][x := 2;\ A/X_2]$
$\quad \Box\ x = 2 \rightarrow \mathbf{S}_2[x := 1;\ A/X_1][x := 2;\ A/X_2]$ **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"; guard$_Z(\mathbf{S}_1)$[action := "A_i"; A/**call** A_i]
$\Box$ action = "A_2" $\rightarrow$ action := "O"; guard$_Z(\mathbf{S}_2)$[action := "A_i"; A/**call** A_i]
...
$\Box$ action = "A_n" $\rightarrow$ action := "O"; 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 guard$_Z(\mathbf{S})$ is defined in a similar way to guard$_n(\mathbf{S})$ so that:

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

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 [18]).

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.

2.6 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.

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 $\ldots$; $F(t, v)$; $\ldots$
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 $F \equiv$
    $a \overset{\text{push}}{\longleftarrow} d$;
    $S[x[1]/x][y[1]/y][a[1]/a]$
      $[x \overset{\text{push}}{\longleftarrow} t'$; $y \overset{\text{push}}{\longleftarrow} v'$; 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 S) and will not affect the stacks. The proof is by the theorems on invariant maintenance for recursive statements [20].

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:

$\lceil x := x + 1;\ x \rfloor$

$\lceil x := x + 1;\ x - 1 \rfloor$

We also have conditional expressions where **if** and **fi** are used as expression brackets, for example:

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:

$a := \lceil S_1;\ b := \textbf{if} \lceil S_2;\ Q \rfloor \textbf{then} \lceil S_3;\ t_1 \rfloor \textbf{else}\ t_2\ \textbf{fi};\ b * b \rfloor$

is represented as:

$S_1;\ S_2;\ \textbf{if}\ Q\ \textbf{then}\ S_3;\ b := t_1\ \textbf{else}\ b := t_2\ \textbf{fi};\ a := b * b$

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** B **then** t_1 **else** t_2 **fi. end**

is interpreted:

begin var r_1, r_2 :
  $F(x);\ r_1 := r;\ F(y);\ r_2 := r;$
  $a := r_1 + r_2$ **end**
where
  proc $F(x)\ \equiv$ **if** B **then** $r := t_1$ **else** $r := t_2$ **fi. end**

The statement:

begin
  $a := \lceil$ **while** $B(x)$ **do** $x := F(x)$ **od**; $x + c \rfloor$
where
  funct $B(x)\ \equiv \lceil S;\ x > y \rfloor$.
  funct $F(x)\ \equiv$ **if** B **then** t_1 **else** t_2 **fi.**

is interpreted:

begin
  do $B(x)$; **if** $r =$ "ff" **then exit fi**;
    $F(x);\ x := r$ **od**;
  $a := x + c$ **where**
  proc $B(x)\ \equiv$ S; **if** $x > y$ **then** $r :=$ "tt" **else** $r :=$ "ff" **fi**,
    proc $F(x)\ \equiv$ **if** B **then** $r := t_1$ **else** $r := t_2$ **fi. end**

See [20] 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:

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

can be expanded over the following statement to give:

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

3.2 Loop Inversion

If the statement S_1 contains no **exit**s which can cause termination of an enclosing loop (i.e. in the notation of [20] it is a *proper sequence*) then the loop:

$$\underline{\textbf{do}}\ \mathrm{S}_1;\ \mathrm{S}_2\ \underline{\textbf{od}}$$

can be inverted to:

$$\mathrm{S}_1;\ \underline{\textbf{do}}\ \mathrm{S}_2;\ \mathrm{S}_1\ \underline{\textbf{od}}$$

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 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:*

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

Lemma 3.2 *Selective unrolling of* **while** *loops:* For any condition $\mathbf{Q}$ we have:

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

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

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

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 $\mathbf{S}$; $\mathbf{S}'$ of two statements can be merged together by the *absorption* The statement $\mathbf{S}'$ following $\mathbf{S}$ is "absorbed" into it by replacing all of the terminal statements of $\mathbf{S}$ which would lead to $\mathbf{S}'$ by a copy of $\mathbf{S}'$ 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:
  $A_1 \equiv \mathbf{S}_1$.
  $\ldots A_i \equiv \mathbf{S}_i$.
  $\ldots B_j \equiv \mathbf{S}_{j0}$; $F(g_{j1}(x))$; $\mathbf{S}_{j1}$; $F(g_{j2}(x))$; $\ldots$; $F(g_{jn_j}(x))$; $\mathbf{S}_{jn_j}$.
  $\ldots$ **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 and where we have added a new action $\hat{F}$ to the action system:

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

Proof: See [23] for the proof.

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 [20] 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 [20].

Corollary 4.2 By unfolding some calls to $\hat{F}$ and pruning, we get the following, slightly more efficient, version:

proc $F'(x)$ $\equiv$
  var $L := \langle\rangle, m := 0$:
    actions A_1 :
    $A_1 \equiv S_1[\underline{\textbf{call}}\ \hat{F}/\underline{\textbf{call}}\ Z]$.
    $\ldots A_i \equiv S_i[\underline{\textbf{call}}\ \hat{F}/\underline{\textbf{call}}\ Z]$.
    $\ldots B_j \equiv S_{j0};\ L := \langle\langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle +\!\!+ L;$
      $x := g_{j1}(x);$ **call** A_1.
    $\ldots \hat{F} \equiv$ **if** $L = \langle\rangle$ **then call** Z
      else $\langle m, x\rangle \leftarrow L;$
        if $m = 0 \rightarrow$ **call** A_1
        $\square \ldots \square\ m = \langle j, k\rangle \rightarrow S_{jk}[\underline{\textbf{call}}\ \hat{F}/\underline{\textbf{call}}\ Z]$
        $\ldots$ **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 $S_1;\ F(g_1(x));\ M(x);\ F(g_2(x));\ S_2$ **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** $\mathbf{B}$ **then** $\mathbf{T}$
    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 :
    $A_1 \equiv$ **if** $\mathbf{B}$ **then** $\mathbf{T}$
      else call B_1 **fi**.
    $B_1 \equiv \mathbf{S}_1$; $L := \langle\langle 0, g_1(x)\rangle, \langle 1, x\rangle\rangle +\!\!+ L$; **call** $\hat{F}$.
    $B_2 \equiv M(x)$; $L := \langle\langle 0, g_2(x)\rangle, \langle 2, x\rangle\rangle +\!\!+ L$; **call** $\hat{F}$.
    $A_2 \equiv \mathbf{S}_2$; **call** $\hat{F}$.
    $\hat{F} \equiv$ **if** $L = \langle\rangle$ **then call** Z
      else $\langle m, x\rangle \leftarrow L$;
        if $m = 0 \rightarrow$ **call** A_1
        $\Box$ $m = 1 \rightarrow$ **call** B_1
        $\Box$ $m = 2 \rightarrow$ **call** B_2 **fi fi**. **endactions end**.

The action system is (naturally) regular, so we can apply the transformations in [20] to restructure the action system:

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

Applying the general recursion removal transformation we get:

proc $F(x)$ $\equiv$
  var $L := \langle\rangle, m := 0$:
    actions A :
    $A \equiv$ **if** $\mathbf{B}$ **then** $\mathbf{T}$

```
                else call B fi.
B  ≡ S1; L := ⟨⟨0, g1(x)⟩, ⟨1, x⟩, ⟨0, g2(x)⟩, ⟨2, x⟩⟩ ++ L; call F̂.
F̂  ≡ if L = ⟨⟩ then call Z
                else ⟨m, x⟩ ← L;
                     if m = 0 → call A
                     □ m = 1 → M(x); call F̂
                     □ m = 2 → S2; call F̂ 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)
                □ m = 2 → S2 fi od end.
```

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

```
proc F(x) ≡
    var L := ⟨⟩, m := 0:
        do while ¬B do S1; L := ⟨⟨1, x⟩, ⟨0, g2(x)⟩, ⟨2, x⟩⟩ ++ L; x := g1(x) od;
           T;
           do if L = ⟨⟩ then exit(2) fi;
               ⟨m, x⟩ ← L;
               if m = 0 → exit
               □ m = 1 → M(x)
               □ m = 2 → S2 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) ≡
    if n = 0 then T
             else S1; G(n − 1); M(n); G(n − 1); S2 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 ≡
    if n = 0 then T
             else S1; n := n − 1; G; M(n + 1); G; n := n + 1; S2 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 := ⟨⟩, d := 0 :
    do while n ≠ 0 do S1; n := n − 1; L := ⟨1⟩ ++ L od;
        T;
        do if L = ⟨⟩ then exit(2) fi;
            d ← L;
            if d = 1 → M(n + 1); L := ⟨2⟩ ++ L; exit
            □ d = 2 → S2; 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 $L := \langle 1\rangle +\!\!+ L$ becomes $c := 2.c + 1$, $L := \langle 2\rangle +\!\!+ L$ 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; $n := n - 1$; $L := \langle 1\rangle +\!\!+ L$ **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
```

Using the transformations in [20] we can transform this into the following:

```
var n0 := n:
    for i := n step −1 to 1 do S1[i/n] od;
    T;
    for c := 1 step 1 to 2^n0 − 1 do
        n := ntz(c);
        for i := 0 step 1 to n − 1 do S2[i/n] od;
        M(n + 1);
        for i := n step −1 to 1 do S1[i/n] od;
        T od;
    for i := 0 step 1 to n0 − 1 do S2[i/n] od end
```

where ntz(c) is the number of trailing zeros in the binary representation of c.

For the case where S_1 and 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 $g(n) \equiv$
  if $n = 0$ **then** $\langle\langle\rangle\rangle$
    else $(\langle 0\rangle +\!\!+) * g(n-1) +\!\!+ \mathsf{reverse}((\langle 0\rangle +\!\!+) * g(n-1))$ **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 $g'(n,s) \equiv$
  if $n = 0$ **then** $\langle\langle\rangle\rangle$
    else $(\langle s\rangle +\!\!+) * g'(n-1,0) +\!\!+ (\langle 1-s\rangle +\!\!+) * g'(n-1,1)$ **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 $g''(L,n,s) \equiv$
  if $n = 0$ **then** $\langle L\rangle$
    else $g''(\langle s\rangle +\!\!+ L, n-1, 0) +\!\!+ g''(\langle 1-s\rangle +\!\!+ L, n-1, 1)$ **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 $G(c', n) \equiv$
    if $n = 0$ **then** $r := r +\!\!+ \langle \mathsf{bits}(\lfloor c'/2 \rfloor) \rangle$
      else $G(2.c', n-1)$; $G(2.(c' \oplus 1) \oplus 1, n-1)$ **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 $G(c, n) \equiv$
    if $n = 0$ **then** $r := r +\!\!+ \langle \mathsf{bits}(\lfloor c/2 \rfloor) \rangle$
      else $G(c, n-1)$; $G((c \oplus 2^n) \oplus 2^{n-1}, n-1)$ **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 $G(c, n) \equiv$
      if $n = 0$ **then** $r := r +\!\!+ \langle \mathsf{bits}(\lfloor c/2 \rfloor) \rangle$; $c' := c' \oplus 1$
        else $G(c, n-1)$; $c' := c' \oplus 2^n$; $G((c \oplus 2^n) \oplus 2^{n-1}, n-1)$ **fi**. **end end**

By induction on n we prove: $\{c' = c\}$; $G(c, n-1) \leq \{c' = c\}$; $G(c, n-1)$; $\{c' = c \oplus 2^n\}$. 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 $G(n) \equiv$
      if $n = 0$ **then** $r := r +\!\!+ \langle \mathsf{bits}(\lfloor c'/2 \rfloor) \rangle$; $c' := c' \oplus 1$
        else $G(n-1)$; $c' := c' \oplus 2^n$; $G(n-1)$ **fi**. **end end**

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

begin var $c' := 2^{N+1}$:
    $r := r +\!\!+ \langle \mathsf{bits}(\lfloor c'/2 \rfloor) \rangle$;
    for $i := 1$ **step** 1 **to** $2^N - 1$ **do**
      $c' := c' \oplus 2^{\mathsf{ntz}(i)+1}$;
      $r := r +\!\!+ \langle \mathsf{bits}(\lfloor c'/2 \rfloor) \rangle$ **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 $c := 0$:
    $r := r +\!\!+ \langle \mathsf{Nbits}(c) \rangle$;
    for $i := 1$ **step** 1 **to** $2^N - 1$ **do**
      $c := c \oplus 2^{\mathsf{ntz}(i)}$;
      $r := r +\!\!+ \langle \mathsf{Nbits}(c) \rangle$ **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:

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

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:

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

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 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 analysed by 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 uncover the specification this short program. With the aid of our theorem the analysis breaks down into three steps:

1. Restructure into the right form for application of the theorem (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 ≡ 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 (4.2) 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.**

This procedure can be written in a functional form:

begin
  $r := F(n, m)$
where
  funct $F(m, n) \equiv$
      if $m = 0$ **then** $n + 1$
      elsif $n = 0$ **then** $F(m - 1, 1)$
          else $F(m - 1, F(m, n - 1))$ **fi.**
end

This is the famous Ackermann function [1].

9 Conclusion

In our work on the derivation of algorithms from specifications by formal refinement we find that the problem can often be broken down into the following stages:

1. Nonexecutable specification
2. Recursively defined specification
3. Recursive procedure
4. Iterative algorithm

In [26] we prove some important transformations which enable the transition from (2) to (3) to be carried out easily. In this paper we provide a general-purpose recursion removal transformation which can achieve the transition from (3) to (4). There is often more than one way to apply the theorem, with each method generating a different iterative algorithm. The aim here is not simply to improve efficiency but to discover new algorithms and prove properties of existing algorithms. An added benefit of the theorem, which illustrates its wide applicability, is that it can be applied to a given iterative algorithm which uses a stack or array in a particular way. This produces a recursive procedure which is often much easier to analyse and understand. This aspect of the work is being investigated in the "Maintainer's Assistant" project [10,28] 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 project uses program transformations as a means of code analysis as well as program development.

10 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] B. A. Davey & H. A. Priestley, "Partition-induced natural dualities for varieties of pseudocomplemented distributive lattices ," *Discrete Math.* to appear (1992).

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

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

[14] C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

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

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

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

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

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

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

[21] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990.

[22] M. Ward, "A Definition of Abstraction," University of Durham Technical Report, 1990.

[23] M. Ward, "A Recursion Removal Theorem - Proof and Applications," Durham University, Technical Report, 1991.

[24] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (1993), 101–122.

[25] M. Ward, "The Largest True Square Problem—An Exercise in the Derivation of an Algorithm," Durham University, Technical Report, Apr., 1990.

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

[27] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," Submitted to IEEE Trans. Software Eng., May, 1992.

[28] M. Ward, F. W. Calliss & M. Munro, "The Maintainer's Assistant," *Conference on Software Maintenance 16th–19th October 1989, Miami Florida* (Oct., 1989).