CHAPTER FIVE

Transformation of Recursive Systems

"The transformation from recursion to iteration is one of the most fundamental concepts of computer science." Knuth 1974

Introduction

In this chapter we build on the results of the previous chapter to derive further transformations of recursive actions. These are used to extend our programming language to include procedures with parameters and local variables. We initially concentrate on transformations from recursive to iterative forms: this is not simply because iterative algorithms are more efficient but because it gives insights into deriving new iterative (and recursive) strategies. As Arsac puts it [Arsac 82]: "The true problem is not efficiency: the recursive procedures may be so efficient that the effort to obtain iterative ones is not worthwhile. It is rather a problem of knowledge: we are interested in discovering new iterative strategies." He goes on to make the bold claim: "Transforming recursion into iteration is, in my opinion, the most powerful way to develop iterative procedures and discover new strategies". The transformations from recursion to iteration have proved fundamental to our research on the derivation of algorithms from specifications. In the course of deriving an algorithm from its specification, the application of our "theorem on the recursive implementation of specifications" naturally gives rise to recursive functions or procedures, which may well be good candidates for transformation to an iterative algorithm.

This Chapter deals with the two main techniques for recursion removal, adding a protocol stack and adding a stack to record "postponed" obligations to compute statements or recursive calls. The first section deals with the case of a regular action system (a regular system corresponds to a program written using labels and **goto** statements together with a halt statement).

Solution of a Regular Action Systems:

Suppose we have an action $\mathbf{X}_1 \equiv \mathbf{S}_1$ in a regular action system where $\mathbf{S}_1$ contains calls to several other actions, say $\mathbf{X}_2,\ldots,\mathbf{X}_n$, as well as $\mathbf{X}_1$. We can replace all these by calls to a new action **Y** by adding a new variable **call** and defining:

$\mathbf{Y} \equiv$ **if** $\mathbf{call}=\text{`}\mathbf{X}_2\text{'} \rightarrow \mathbf{X}_2 \;\square\; \ldots \;\square\; \mathbf{call}=\text{`}\mathbf{X}_n\text{'} \rightarrow \mathbf{X}_n \;\square\; \mathbf{call}=\text{`}\mathbf{Z}\text{'} \rightarrow \mathbf{Z}$ **fi**

$\mathbf{X}_1 \equiv \mathbf{S}_1[\mathbf{call}{:=}\text{`}\mathbf{X}_i\text{'};\mathbf{Y}/\mathbf{X}_i|\mathbf{i}=2,\ldots,\mathbf{n}][\mathbf{call}{:=}\text{`}\mathbf{Z}\text{'};\mathbf{Y}/\mathbf{Z}]$

Then by applying the theorem of the last chapter we get $\mathbf{X}_1 \approx$

$\mathbf{X_1} \equiv$ **do do** S_1[**exit**/X_1][**call:=‘X_i’;exit(2)/X_i|i=2,...,n**][**call:=‘Z’;exit(2)/Z**] **od od; Y.**
We would like to apply absorption to this to get something like:
$\mathbf{X_1} \equiv$ **do do** S_1[**exit**/X_1][**call:=‘X_i’;Y+2/X_i|i=2,...,n**][**call:=‘Z’;Y+2/Z**] **od od.** (*)

However, the way we defined **S+k** for primitive statements **S** was **S;exit(k)** which in this case would give **Y;exit(2)** within the loop which violates our rule about not having action calls from within **do...od** loops. This is because of the problem of keeping track of different incarnations of a recursive action being at different depths, which would make things more complicated than they need be. The action call is not "really" in the loop however since as soon as it is completed the loop is terminated. What we need is some way of expressing the fact that the action call also causes termination of the loop. We do this by extending the definitional transformation to transform $\mathbf{X_i}$**+k** to **action:=‘X_i’;X+k** and adding to the definition of $\mathbf{guard}_n$ so that $\mathbf{guard}_n$**(X+k)= if depth=n then depth:=depth−k; X fi**. We deal with **Z+k** within $\mathbf{guard}_Z$ by defining $\mathbf{guard}_Z$**(Z+k)= action:=‘Z’;exit(k)**. With these definitions we can allow action calls of the form **X+k** provided they occur within **k** nested loops. We will then still have **depth=0** for every procedure call.

To sum up:
Defn: If **AS=** $\langle \mathbf{X_1} \equiv \mathbf{S_1},\dots,\mathbf{X_m} \equiv \mathbf{S_m}\rangle$ is a (not necessarily regular) action system which includes action calls of the form $\mathbf{X_i}$**+k** then it is interpreted by the definitional transformation as:
action:=‘A_1’;A where
proc A $\equiv$
if action=‘A_1’ → action:=‘O’; guard$_Z$(S_1[action:=‘A_i’;A+k/A_i+k])
□ . . .
□ **action=‘A_m’ → action:=‘O’; guard$_Z$(S_m[action:=‘A_i’;A+k/A_i+k]) fi.**

where $\mathbf{guard}_Z$ is defined as in the previous Chapter with: $\mathbf{guard}_Z$**(Z+k)= action:=‘Z’; exit(k)** and $\mathbf{guard}_n$**(X+k)= if depth=n then depth:=depth−k; X fi**.

Note that if we unfold an action call $\mathbf{X_i}$**+k** where $\mathbf{X_i} \equiv \mathbf{S_i}$ then we replace it by $\mathbf{S_i}$**+k**.
Hence if we unfold the calls to **Y** in (*) we get:
$\mathbf{X_1} \equiv$ **do do** S_1[**exit**/X_1]
[**call:=‘X_i’; if call=‘X_2’ → X_2 □ ... □ call=‘X_n’ → X_n □ call=‘Z’ → Z fi+2/X_i|i=2,...,n**]
[**call:=‘Z’; if call=‘X_2’ → X_2 □ ... □ call=‘X_n’ → X_n □ call=‘Z’ → Z fi+2/Z**] **od od.**

Pruning the **if**s gives:
$\mathbf{X_1} \equiv$ **do do**S_1[**exit**/X_1][**call:=‘X_i’; X_i+2/X_i|i=2,...,n**][**call:=‘Z’; Z+2/Z**] **od od.**

Now the variable **call** is dead so it can be removed:
$\mathbf{X_1} \;\equiv\; \underline{\mathbf{do}}\ \underline{\mathbf{do}}\mathbf{S_1[\underline{exit}/X_1][X_i+2/X_i|i=2,...,n][Z+2/Z]}\ \underline{\mathbf{od}}\ \underline{\mathbf{od}}.$

With this substitution $\mathbf{X_1}$ is no longer recursive so it can be eliminated from the system (provided it is not the first in the system) by unfolding all calls to it and removing it. Also **Y** can be removed since there are no calls to it. Repeated use of this rule on each recursive action in turn, together with unfolding of each non-recursive action will eliminate every action in the system (except the first and **Z**). We get:

$\mathbf{AS} \approx$ **action:='$\mathbf{A_1}$'; A** $\underline{\textbf{where}}$
$\underline{\textbf{proc}}$ **A** $\equiv$
$\underline{\textbf{if}}$ **action='$\mathbf{A_1}$'** $\rightarrow$ **action:='O'; guard$_Z$($\mathbf{S_1}$[action:='$\mathbf{A_i}$';A+k/$\mathbf{A_i}$+k])** $\underline{\textbf{fi}}$.

where $\mathbf{S_1}$ contains no action calls except those of the form **Z+k**. Hence:

$\mathbf{AS} \approx$ **action:='$\mathbf{A_1}$';** $\underline{\textbf{proc}}$ **A** $\equiv$ $\underline{\textbf{if}}$ **action='$\mathbf{A_1}$'** $\rightarrow$ **action:='O'; guard$_Z$($\mathbf{S_1}$)** $\underline{\textbf{fi}}$.
$\approx$ **action:='$\mathbf{A_1}$';**
$\underline{\textbf{if}}$ **action='$\mathbf{A_1}$'** $\rightarrow$ **action:='O'; guard$_Z$($\mathbf{S_1}$)** $\underline{\textbf{fi}}$
[A/$\underline{\textbf{proc}}$ **A** $\equiv$ $\underline{\textbf{if}}$ **action='$\mathbf{A_1}$'** $\rightarrow$ **action:='O'; guard$_Z$($\mathbf{S_1}$)** $\underline{\textbf{fi}}$**.]**
by recursion unfolding.
$\approx$ **action:='$\mathbf{A_1}$';** $\underline{\textbf{if}}$ **action='$\mathbf{A_1}$'** $\rightarrow$ **action:='O'; guard$_Z$($\mathbf{S_1}$)** $\underline{\textbf{fi}}$

since $\mathbf{S_1}$ contains no calls to actions other than **Z+k**.
$\approx$ **action:='$\mathbf{A_1}$'; action:='O'; guard$_Z$($\mathbf{S_1}$)** by pruning the $\underline{\textbf{if}}$.
$\approx$ **action:='O'; guard$_Z$($\mathbf{S_1}$)**

We claim that this is equivalent to $\underline{\textbf{do}}$ **$\mathbf{S_1}$[$\underline{\textbf{exit}}$(k+1)/Z+k]** $\underline{\textbf{od}}$.

Note that if all occurrences of **Z** are in terminal positions then the body of the loop is reducible and therefore the loop is equivalent to **$\mathbf{S_1}$[$\underline{\textbf{exit}}$(k)/Z+k]** by false iteration since every execution of $\mathbf{S_1}$ leads to an action call which must be a **Z+k** at depth **k** (note that this property is preserved by the recursion removal transformation) hence every terminal statement of **$\mathbf{S_1}$[$\underline{\textbf{exit}}$(k+1)/Z+k]** is an **$\underline{\textbf{exit}}$(k+1)** substituted for a **Z+k**. **Proof of Claim: action:='O'; guard$_Z$($\mathbf{S_1}$)** $\approx$ $\underline{\textbf{do}}$ **(action:='O'; guard$_Z$($\mathbf{S_1}$))+1** $\underline{\textbf{od}}$ by false iteration.
$\approx$ $\underline{\textbf{do}}$ **action:='O'; guard$_Z$($\mathbf{S_1}$+1)** $\underline{\textbf{od}}$

Now **guard$_0$(**$\underline{\textbf{do}}$ **action:='O'; guard$_Z$($\mathbf{S_1}$+1)** $\underline{\textbf{od}}$**)**
$\approx$ **depth:=1;** $\underline{\textbf{while}}$ **depth=1** $\underline{\textbf{do}}$ **guard$_1$(action:='O'; guard$_Z$($\mathbf{S_1}$+1))** $\underline{\textbf{od}}$

From the way the recursion removal was carried out we see that each occurrence of **Z+k** must be at depth **k**. (Prove that this property is preserved by the recursion removal transformation).
guard$_Z$(Z+k)= action:='Z'; **$\underline{\textbf{exit}}$(k)** which therefore sets the depth to zero.

Hence **depth=0** iff **action='Z'** and **depth>0** iff **action='O'**. So we can replace **Z+k** by **<u>exit</u>(k)** and remove the $\mathbf{guard}_Z$.

<u>Removing</u> <u>Non</u>-<u>terminal</u> <u>Recursion</u>

Suppose our set of actions has some non-terminal calls such as: **F;Y**. If every execution of **F** eventually leads to a call of the terminating action (this is always the case for a <u>regular</u> action system) then the statement **Y** in the action cannot be reached and so may be eliminated.

To <u>regularise</u> a set of actions we need to add action calls to those terminal statements which are not action calls and which can be reached without encountering an action.

If **F** has such a terminal statement then we add a new action called **/F** and add a call to **/F** after each such terminal statement. So far this is simply a void action (ie the body of **/F** is simply **skip**) and is therefore itself non-regular. In order to "regularise" **/F** we need to copy into it the sequence of statements which are executed on termination of a call to **/F** via a call to **F**. If **F** is the only non-regular action then the result of this copying will make **/F** regular and hence the whole system will be regular. If there are several non-regular actions then they will have to be treated in turn.

<u>Theorem:</u> Suppose we have the following program which calls a recursive procedure:
$\mathbf{P} \equiv \mathbf{S_1;\ F_0;\ S_2.}$ **<u>where</u>**
<u>proc</u> $\mathbf{F_0} \equiv$ **<u>if</u> B <u>then</u>** $\mathbf{a;\ F_0;\ b;\ F_0;\ c}$
<u>else</u> d <u>fi</u>.

The problem with transforming this to an iterative form is that when an activation of $\mathbf{F_0}$ terminates we cannot tell which of the three calls gave rise to that particular activation. So we cannot tell whether $\mathbf{S_2}$ or **b** or **c** will be executed after $\mathbf{F_0}$ terminates. This becomes clearer if we write the program in this form (which can easily be seen to be equivalent to the first form by copying in **/F**, $\mathbf{X_0}$, $\mathbf{X_1}$ and $\mathbf{X_2}$):
$\mathbf{P} \equiv \mathbf{S_1;\ X_0.}$
$\mathbf{X_0} \equiv \mathbf{F_0;\ S_2;\ Z.}$
$\mathbf{F_0} \equiv$ **<u>if</u> B <u>then</u>** $\mathbf{a;\ X_1}$
<u>else</u> d; /F <u>fi</u>.
$\mathbf{X_1} \equiv \mathbf{F_0;\ b;\ X_2.}$
$\mathbf{X_2} \equiv \mathbf{F_0;\ c;\ /F_0.}$
$\mathbf{/F_0} \equiv$ **skip.**

Now a call to any action other than $\mathbf{/F_0}$ leads to an action call. The problem is that when $\mathbf{/F_0}$ terminates the next statement to be executed will either be $\mathbf{S_2}$ or **b** or **c** so we cannot jump directly to it. Suppose there is some condition of the state which can be tested to determine which statement to execute–ie suppose we can insert assertions thus:

$\mathbf{P} \equiv \mathbf{S}_1;\ \mathbf{X}_0.$
$\mathbf{X}_0 \equiv \mathbf{F};\ \{\mathbf{B}_0\};\ \mathbf{S}_2;\ \mathbf{Z}.$
$\mathbf{F}_0 \equiv \underline{\mathbf{if}}\ \mathbf{B}\ \underline{\mathbf{then}}\ \mathbf{a};\ \mathbf{X}_1$
$\quad \underline{\mathbf{else}}\ \mathbf{d};\ /\mathbf{F}\ \underline{\mathbf{fi}}.$
$\mathbf{X}_1 \equiv \mathbf{F}_0;\ \{\mathbf{B}_1\};\ \mathbf{b};\ \mathbf{X}_2.$
$\mathbf{X}_2 \equiv \mathbf{F};\ \{\mathbf{B}_2\};\ \mathbf{c};\ /\mathbf{F}_0.$
$/\mathbf{F}_0 \equiv \mathbf{skip}.$

where $\mathbf{B}_0$, $\mathbf{B}_1$ and $\mathbf{B}_2$ are disjoint, ie $(\mathbf{B}_0 \wedge \mathbf{B}_1) \iff (\mathbf{B}_0 \wedge \mathbf{B}_2) \iff (\mathbf{B}_1 \wedge \mathbf{B}_2) \iff \mathbf{false}.$

Then we claim that the program is equivalent to the following:

$\mathbf{P}_2 \equiv \mathbf{S}_1;\ \mathbf{X}_0$
$\mathbf{X}_0 \equiv \mathbf{F}.$
$\mathbf{F} \equiv \underline{\mathbf{if}}\ \mathbf{B}\ \underline{\mathbf{then}}\ \mathbf{a};\ \mathbf{X}_1$
$\quad \underline{\mathbf{else}}\ \mathbf{d};\ /\mathbf{F}\ \underline{\mathbf{fi}}.$
$\mathbf{X}_1 \equiv \mathbf{F}.$
$\mathbf{X}_2 \equiv \mathbf{F}.$
$/\mathbf{F} \equiv \underline{\mathbf{if}}\ \mathbf{B}_0 \rightarrow \mathbf{S}_2;\ \mathbf{Z}$
$\quad \Box\ \mathbf{B}_1 \rightarrow \mathbf{b};\ \mathbf{X}_2$
$\quad \Box\ \mathbf{B}_2 \rightarrow \mathbf{c};\ /\mathbf{F}\ \underline{\mathbf{fi}}.$

Unfolding the calls to $\mathbf{F}$ in $\mathbf{X}_0$, $\mathbf{X}_1$ and$\mathbf{X}_2$ will make the bodies of these four actions identical. They can
all be replaced by $\mathbf{F}$ to get:

$\mathbf{P}_2 \equiv \mathbf{S}_1;\ \mathbf{F}.$
$\mathbf{F} \equiv \underline{\mathbf{if}}\ \mathbf{B}\ \underline{\mathbf{then}}\ \mathbf{a};\ \mathbf{F}$
$\quad \underline{\mathbf{else}}\ \mathbf{d};\ /\mathbf{F}\ \underline{\mathbf{fi}}.$
$/\mathbf{F} \equiv \underline{\mathbf{if}}\ \mathbf{B}_0 \rightarrow \mathbf{S}_2;\ \mathbf{Z}$
$\quad \Box\ \mathbf{B}_1 \rightarrow \mathbf{b};\ \mathbf{F}$
$\quad \Box\ \mathbf{B}_2 \rightarrow \mathbf{c};\ /\mathbf{F}\ \underline{\mathbf{fi}}.$

To prove that the two programs are equivalent we will prove that:

$$\mathbf{F}_0;\ /\mathbf{F} \approx \mathbf{F}$$

From this we have:

$\mathbf{P}_2 \approx \mathbf{S}_1; \mathbf{F}_0; /\mathbf{F}$
$\approx \mathbf{S}_1; \mathbf{F}_0; \{\mathbf{B}_0\}; /\mathbf{F}$ from the premises.
$\approx \mathbf{S}_1; \mathbf{F}_0; \{\mathbf{B}_0\}; \mathbf{S}_2; \mathbf{Z}$ since the $\mathbf{B}_i$ are disjoint conditions.
$\approx \mathbf{S}_1; \mathbf{F}_0; \mathbf{S}_2; \mathbf{Z}$
$\approx \mathbf{P}$
as required.

To prove $\mathbf{F}_0; /\mathbf{F} \approx \mathbf{F}$ we will prove the following by induction on $\mathbf{n}$:
(1) $\mathbf{F}^n \leqslant \mathbf{F}_0; (/\mathbf{F})^n$.
(2) $(\mathbf{F}_0)^n; /\mathbf{F} \approx \mathbf{F}$.

Proof of (1):
$\mathbf{F}^{n+1} \approx$ **<u>if</u> B <u>then</u> a;** $\mathbf{F}^n$ by unfolding.
<u>else</u> d; $(/\mathbf{F})^n$ **<u>fi</u>**
$\leqslant$ **<u>if</u> B <u>then</u> a;** $\mathbf{F}_0; (/\mathbf{F})^n$ by induction hypothesis.
<u>else</u> d; $(/\mathbf{F})^n$ **<u>fi</u>**
$\leqslant$ **<u>if</u> B <u>then</u> a;** $\mathbf{F}_0; (/\mathbf{F})^{n+1}$
<u>else</u> d; $(/\mathbf{F})^n$ **<u>fi</u>**
$\approx$ **<u>if</u> B <u>then</u> a;** $\mathbf{F}_0$; **b;** $\mathbf{F}^n$ by unfolding $(/\mathbf{F})^{n+1}$ and inserting
<u>else</u> d; $(/\mathbf{F})^n$ **<u>fi</u>** the assertion $\{\mathbf{B}_1\}$.
$\leqslant$ **<u>if</u> B <u>then</u> a;** $\mathbf{F}_0$; **b;** $\mathbf{F}_0; (/\mathbf{F})^n$ by induction hypothesis.
<u>else</u> d; $(/\mathbf{F})^n$ **<u>fi</u>**
$\approx$ **<u>if</u> B <u>then</u> a;** $\mathbf{F}_0$; **b;** $\mathbf{F}_0$
<u>else</u> d <u>fi</u>; $(/\mathbf{F})^n$ by taking out the $(/\mathbf{F})^n$.
$\approx \mathbf{F}_0; (/\mathbf{F})^n$ by folding $\mathbf{F}_0$.
$\leqslant \mathbf{F}_0; (/\mathbf{F})^{n+1}$ as required.

Proof of (2):
$(\mathbf{F}_0)^{n+1}; /\mathbf{F} \approx$ **<u>if</u> B <u>then</u> a;** $(\mathbf{F}_0)^n$; **b;** $(\mathbf{F}_0)^n$; **c**
<u>else</u> d <u>fi</u>; /F.
$\approx$ **<u>if</u> B <u>then</u> a;** $(\mathbf{F}_0)^n; \{\mathbf{B}_1\}$; **b;** $(\mathbf{F}_0)^n; \{\mathbf{B}_2\}$; **c**
<u>else</u> d <u>fi</u>; /F. by inserting assertions.
$\approx$ **<u>if</u> B <u>then</u> a;** $(\mathbf{F}_0)^n; \{\mathbf{B}_1\}$; **b;** $(\mathbf{F}_0)^n; \{\mathbf{B}_2\}$; **c; /F**
<u>else</u> d; /F <u>fi</u>.

$\approx$ **<u>if</u> B <u>then</u> a; $(\mathbf{F}_0)^n$; $\{\mathbf{B}_1\}$; b; $(\mathbf{F}_0)^n$; /F** by folding **/F**.
<u>else</u> d; /F <u>fi</u>.
$\leqslant$ **<u>if</u> B <u>then</u> a; $(\mathbf{F}_0)^n$; $\{\mathbf{B}_1\}$; b; F** by induction hypothesis.
<u>else</u> d; /F <u>fi</u>.

$\approx$ **<u>if</u> B <u>then</u> a; $(\mathbf{F}_0)^n$; /F** by definition of **/F**.
<u>else</u> d; /F <u>fi</u>.
$\leqslant$ **<u>if</u> B <u>then</u> a; F** by induction hypothesis.
<u>else</u> d; /F <u>fi</u>.
$\approx$ **F**. by folding.
This proves the Theorem.
For the general case we can treat each non-regular action in turn by using this technique.

Finding such $\mathbf{B}_i$ to insert may be quite a problem since there may be no way of distinguishing the next action to call from the state alone. A way round this is to introduce new "ghost variables" which "enrich" the state so that it becomes possible to distinguish the actions. For example, by adding a single variable **next** and changing $\mathbf{X}_i$ to $\mathbf{X}_i \equiv$ **next:='$\mathbf{Y}_i$';F;$\mathbf{Y}_i$** will work (with $\mathbf{B}_i$ as **next='$\mathbf{Y}_i$'**) <u>provided</u> **F** does not change the value of **next**. In general **F** will call other actions $\mathbf{X}_j$, which will re-assign **next**, so this may not work. A completely general solution is to use a stack onto which information designating the successor to **F** is pushed immediately prior to calling **F**. For example:

$\mathbf{X}_0 \equiv$ **stack:=$\langle\rangle$; F; {stack= $\langle\rangle$}; $\mathbf{Y}_0$**
$\mathbf{X}_i \equiv$ **stack:=stack&$\langle$'$\mathbf{Y}_i$'$\rangle$; {hd(stack)='$\mathbf{Y}_i$'}; F;**
{hd(stack)='$\mathbf{Y}_i$'}; stack:=tl(stack); $\mathbf{Y}_i$
/F $\equiv$ **skip**

Given that each inner action call preserves **stack** it is easy to prove that each action body preserves **stack** and hence, by induction, each action preserves **stack**.

We could push the integer **i** onto the stack instead of **'$\mathbf{Y}_i$'**, then the stack will contain integers in the range **0..n−1**. Such a stack can be represented by a single integer, with the digits of this integer in base **n** giving the elements of our stack. Note that we represent the empty stack by **1** rather than **0** so that we can distinguish an empty stack from a stack containing only zeros.

When there is only one $\mathbf{X}_i$, ie when **n=1**, the stack contains a sequence of **1**'s so we only need to record the <u>length</u> of the stack: we represent a sequence of **1**'s by an integer set to zero in $\mathbf{X}_0$, incremented when **1**'s are added and decremented when **1**'s are removed. In fact this integer records the current depth of recursion nesting: while it is non-zero we know that we are within a recursive procedure call, once it reaches zero we know that the outermost procedure call has just terminated.

<u>FORMAL</u> <u>PARAMETERS</u> <u>AND</u> <u>LOCAL</u> <u>VARIABLES</u>

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 "Lvalues" 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 [Ghezzi & Jazayeri 82]. It is generally better to specify that a compiler <u>rejects</u> 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 as follows:

F(t,v) <u>**where**</u>
<u>**proc**</u> **F(x,** <u>**var**</u> **y)** $\equiv$ Here **x** is a value parameter and **y** is a value-result parameter.
  <u>**var**</u> **a:=d, b;** Defines two local variables, the first is assigned an initial value.
  S. The body of the procedure. May contain recursive calls.

Transforms to:
x:=⟨⟩; y:=⟨⟩; a:=⟨⟩; b:=⟨⟩;
x←t; y←v; F; v:=hd(y); x:=tl(x); y:=tl(y) <u>**where**</u>

proc **F** $\equiv$
  a←d; beg b′: b←b′ end;
  S[hd(x),hd(y),hd(a),hd(b) /x,y,a,b]
  [x←t′; y←v′; F; v′:=hd(y); x:=tl(x); y:=tl(y) / F(t′, v′)];
  a:=tl(a); b:=tl(b).
where **x←v** pushes the value of **v** onto the stack stored in **x** and **v←x** pops the top value off the stack in **x** and stores it in **v**.

Here the substitution of **hd(x)** for **x** etc. ensures that the body of the procedure only accesses and updates the top of the stacks which have replaced the parameters and local variables. The statement **beg b′: b←b′ end** has the effect of pushing an arbitrary value onto the top of stack **b**: our formal system ensures that once the procedure has been proved correct we know it will give the right answer whatever value is pushed onto **b**. This means that the program will give the correct result no matter what values are assigned to uninitialised local variables.

With such a procedure it is easy to prove that any call of **F** will only affect the values at the top of the stacks **x**, **y**, **a**, and **b** so an inner recursive call of **F**, which is of the form: **x←t′; y←v′; F; v:=hd(y); x:=tl(x); y:=tl(y)**, will only affect the value of **v** and not affect the stack. The proof is by the theorems on invariant maintenance for recursive statements. All the theorems we derived for recursive procedures acting on global variables have analogous forms for procedures with parameters and local variables. For example:

$$\Delta \vdash \mathbf{S}\ \underline{\mathbf{where}}\ \underline{\mathbf{proc}}\ \mathbf{F(x)} \equiv \mathbf{S'}. \ \approx\ \mathbf{S[S'[y/x]\ /F(y)]}\ \underline{\mathbf{where}}\ \underline{\mathbf{proc}}\ \mathbf{F(x)} \equiv \mathbf{S'}.$$

This is a form of unfolding for procedures with parameters (we are ignoring problems caused by name clashes which can be removed by systematic re-naming of variables).

Defn: We say that a parameter **x** in a procedure **F** is fixed if the call **F(..,x,..)** leads to no other calls of the procedure except those of the form **F(...,x,...)** (with **x** in the same place).
Thus all parameters of non-recursive procedures are fixed.

A fundamental result about recursive procedures is that fixed parameters may be directly replaced by global variables. This follows from the definitional transformation: prove that all the values on the stack representing **x** are the same (we call such a stack a constant stack) whence we can replace the stack by a simple variable, replacing **push** and **pop** operations by simple assignments. We do not need to record the length of the stack since it is never needed (technically we represent a constant stack by two variables, a value and a length, and then note that the length variable is never accessed and hence is dead and can be removed).

Regularisation only applies to parameterless procedures acting on global variables. To remove the recursion when parameters and local variables are present we convert them to global stacks, by using the definitional transformation, and then can apply regularisation. In some cases the initial value of a parameter or local variable (ie the value pushed onto the stack) can be computed from the final value (ie the value popped off the stack) so that we can dispense with the stack.

Transforming Formal Parameters into Global Variables

In this section we discuss two special cases where formal parameters may be replaced by global variables.
Suppose **B** is a formula,
x is a list of formal parameters called by value.
$\mathbf{S}_1$, $\mathbf{S}_2$ are statements which do not call **F** and do not modify **x**.
$\mathbf{S}'$ is a statement which calls **F** but does not modify **x**.

(A): Suppose we have:
proc F(x) $\equiv$ **if B then** $\mathbf{S}_1$
else $\mathbf{S}'$; $\mathbf{S}_2$ **fi.**

To make **x** global we require that every call of **F** in $\mathbf{S}'$, such as **F(g(x))** is replaced by **x:=g(x); F**. This has the effect of destroying the previous value of **x** which must be restored after the call to ensure that **x** is globally invariant over **F**. (this is because $\mathbf{S}_1$, $\mathbf{S}_2$, and $\mathbf{S}'$ do not affect **x**). This can be achieved by using a stack for **x** (as in the definitional transformation): **push(x); x:=g(x); F; pop(x)**. If an inverse function $\mathbf{g^{-1}(x)}$ exists then the stack can be avoided by recomputing the original value of **x**.

To prove that **F** globally preserves **x** we take **T** to be any statement which preserves **x** (ie $\{x=x_0\};T \approx \{x=x_0\};T;\{x=x_0\}$ where $x_0 \notin \mathbf{vars}(T)$) and prove:

$\{x=x_0\}$; **if B then** S_1 **else** S'; S_2 **fi**$[x:=g(x);\ T;\ x:=g^{-1}(x)\ /\ F(g(x))]$
$\approx$ $\{x=x_0\}$; **if B then** S_1 **else** S'; S_2 **fi**$[x:=g(x);\ T;\ x:=g^{-1}(x)\ /\ F(g(x))]$; $\{x=x_0\}$.

Then by the theorem on invariant maintenance:

$\approx$ $\{x=x_0\}$; **proc F** $\equiv$ **if B then** S_1 **else** S'; S_2 **fi**$[x:=g(x);\ F;\ x:=g^{-1}(x)\ /\ F(g(x))]$. ; $\{x=x_0\}$.

Since only $\mathbf{S}'$ contains calls to **F** and nothing else affects **x** we need only prove:

$\{x=x_0\}$; $S'[x:=g(x);\ T;\ x:=g^{-1}(x)\ /\ F(x)]$
$\approx$ $\{x=x_0\}$; $S'[x:=g(x);\ T;\ x:=g^{-1}(x)\ /\ F(g(x))]$; $\{x=x_0\}$.

If the above can be proved then an equivalent recursive action is:

$\mathbf{F} \equiv$ **if** **B** **then** $\mathbf{S_1}$
else $\mathbf{S'[x{:=}g(x);\ F;\ x{:=}g^{-1}(x)\ /\ F(g(x))];\ S_2}$ **fi**.

(B): Suppose we have:
proc $\mathbf{F(x)} \equiv$ **if** **B** **then** $\mathbf{S_1}$
else $\mathbf{S';\ F(h(x))}$ **fi**.

In this case if a stack is used this becomes:
$\mathbf{F} \equiv$ **if** **B** **then** $\mathbf{S_1}$
else $\mathbf{S'[push(x);\ x{:=}g(x);\ F;\ pop(x)\ /\ F(g(x))];\ x{:=}h(x);\ F}$ **fi**.

The final call of **F** is terminal so we need only distinguish the outermost call and the first inner one. This leads to a simpler iterative procedure but still requires a stack. If then functions $\mathbf{g^{-1}}$ and $\mathbf{h^{-1}}$ exist (as above) then we can avoid the stack by recomputing **x**:
$\mathbf{F} \equiv$ **if** **B** **then** $\mathbf{S_1}$
else $\mathbf{S'[x{:=}g(x);\ F;\ x{:=}g^{-1}(x)\ /\ F(g(x))];\ x{:=}h(x);\ F;\ x{:=}h^{-1}(x)}$ **fi**.

The final call of **F** is now no longer regular and so we have at least two non-regular calls of **F** which means that if we wish to remove the recursion we would need some way of distinguishing them (the general solution would be to use a protocol stack).

Alternatively we could keep the final call terminal. The final value $\delta(\mathbf{x_0})$ of **x** when exiting **F** has to be connected in some way with the initial value $\mathbf{x_0}$, ie: $\mathbf{\{x{=}x_0\};\ F;\ \{x{=}\ \delta(x_0)\}}$. When **B** holds **x** is not modified so we have: $\mathbf{B} \Rightarrow \mathbf{x_0} = \delta(\mathbf{x_0})$. Otherwise $\mathbf{F(x)}$ ends with $\mathbf{F(h(x))}$ so $\mathbf{x_0}$ and $\mathbf{h(x_0)}$ must give the same final value of **x**. We have the recursive definition for $\delta(\mathbf{x})$:
$\delta(\mathbf{x}) =$ **if** **B** **then** **x** **else** $\delta(\mathbf{h(x)})$ **fi**.

This is tail-recursive and can be transformed into the form of an iterative statement:
$\mathbf{\{x{=}x_0\}}$; **while** $\neg$**B** **do** $\mathbf{x{:=}h(x)}$ **od**; $\mathbf{\{x{=}\ \delta(x_0)\}}$.

If δ and **g** both have computable inverses then we can transform **F** to:
$\mathbf{F} \equiv$ **if** **B** **then** $\mathbf{S_1}$
else $\mathbf{S'[x{:=}g(x);\ F;\ x{:=}\delta^{-1}(x);\ x{:=}g^{-1}(x)\ /\ F(g(x))];\ x{:=}h(x);\ F}$ **fi**

This has the final recursive call of **F** in a terminal position. Recursion removal can then be applied to get:

F $\equiv$ **while B do**
$\quad$ **S′[x:=g(x); F; x:=δ^{-1}(x); x:=g^{-1}(x) / F(g(x))]; x:=h(x) od;**
S$_1$.

Note that if not all of **x** can be recomputed then part of it could be put on a stack.

Summery of Technique

This technique of recursion removal involves the following stages:
(1) Starting with a recursive procedure, transform it to a system of actions on global variables by removing the parameters and local variable. Note that different ways of doing this often give different action systems.
(2) Regularise the system of actions. (generally need to use a stack of some kind to record the action to be executed on completion of the current one. Sometimes the stack can be avoided by making use of assertions).
(3) Carefully examine for possible simplifications. For example: try to replace simple loops by direct computations and look for other representations of the stack which will allow further simplifications. See the proof of the theorem "Arithmetising the Flow of Control" for an example of the simplifications which become possible using this technique.

THE POSTPONED OBLIGATIONS TECHNIQUE

Introduction

The previous section discussed a frequently-used method of recursion removal which, in the general case, involves a "protocol stack": a stack of marks which record which occurrence of the call statement created each invocation of the recursion. This chapter discusses a different technique for implementing recursive procedures and functions as iterative programs, called the "postponed obligations" technique. With this technique, rather than recording our "current position" on a stack when we encounter a recursive call, we record the list of operations (statements and recursive calls) which have to be done once the current operation has finished. To see how this works, imagine an executive working through a pile of paperwork. Some of the tasks can be carried out immediately, others will generate subtasks which can be listed onto more pieces of paper and placed on top of the pile in the order in which they need to be carried out. Some of these subtasks will create further sub-sub-tasks and so on. Once the pile is empty, the executive knows that he has finished.

Sometimes the resulting program can be simplified if the order of execution of certain statements can be re-arranged: this leads to the concept of "statement order independence":

Lemma: Statement Order Independence:

If $\mathbf{S}_1$ and $\mathbf{S}_2$ are statements such that $\mathbf{var}(\mathbf{S}_1) \subseteq \mathbf{V} \cup \mathbf{W}_1$ and $\mathbf{var}(\mathbf{S}_2) \subseteq \mathbf{V} \cup \mathbf{W}_2$, where $\mathbf{V}$, $\mathbf{W}_1$ and $\mathbf{W}_2$ are disjoint sets of variables, and $\mathbf{S}_1$ and $\mathbf{S}_2$ preserve the values of all variables in $\mathbf{V}$ then:

$$\Delta \vdash \mathbf{S}_1;\mathbf{S}_2 \ \approx\ \mathbf{S}_2;\mathbf{S}_1$$

Proof: Let $\mathbf{v}$ be a list of all the variables in $\mathbf{V}$. If $\mathbf{v}'$ are new variables then

$$\mathbf{S}_1 \ \approx\ \underline{\mathbf{beg}}\ \mathbf{v}':=\mathbf{v};\ \mathbf{S}_1;\ \{\mathbf{v}=\mathbf{v}'\}\ \underline{\mathbf{end}}$$

since $\{\mathbf{v}'=\mathbf{v}\};\ \mathbf{S}_1 \ \approx\ \{\mathbf{v}'=\mathbf{v}\};\ \mathbf{S}_1;\ \{\mathbf{v}'=\mathbf{v}\}$.

We now represent the variables $\mathbf{v}$ in $\mathbf{S}_1$ by $\mathbf{v}'$ to form $\mathbf{S}'_1$. We get:

$\mathbf{S}_1;\mathbf{S}_2 \ \approx\ \underline{\mathbf{beg}}\ \mathbf{v}':=\mathbf{v};\ \mathbf{S}'_1;\ \mathbf{v}:=\mathbf{v}'\ \underline{\mathbf{end}};\ \mathbf{S}_2$

$\approx\ \underline{\mathbf{beg}}\ \mathbf{v}':=\mathbf{v};\ \mathbf{S}'_1\ \underline{\mathbf{end}};\ \mathbf{S}_2$ (from above)

$\approx\ \underline{\mathbf{beg}}\ \mathbf{v}':=\mathbf{v};\ \mathbf{S}'_1;\mathbf{S}_2\ \underline{\mathbf{end}}$ since $\mathbf{v}'$ do not occur in $\mathbf{S}_2$

Now $\mathbf{S}'_1$ and $\mathbf{S}_2$ have <u>no</u> variables in common:

<u>Claim</u>: If $\mathbf{S}'_1$ and $\mathbf{S}_2$ have no variables in common then:

$$\Delta \vdash \mathbf{S}'_1;\mathbf{S}_2 \ \approx\ \mathbf{S}_2;\mathbf{S}'_1$$

Then $\mathbf{S}_1;\mathbf{S}_2 \ \approx\ \underline{\mathbf{beg}}\ \mathbf{v}':=\mathbf{v};\ \mathbf{S}_2;\mathbf{S}'_1\ \underline{\mathbf{end}}$

$\approx\ \mathbf{S}_2;\ \underline{\mathbf{beg}}\ \mathbf{v}':=\mathbf{v};\ \mathbf{S}'_1\ \underline{\mathbf{end}}$ since $\mathbf{S}_2$ preserves the values of variable in $\mathbf{v}$.

$\approx\ \mathbf{S}_2;\ \underline{\mathbf{beg}}\ \mathbf{v}':=\mathbf{v};\ \mathbf{S}'_1\ \underline{\mathbf{end}}$

$\approx\ \mathbf{S}_2;\mathbf{S}_1$ as above.

Proof of claim: By induction on the structure of $\mathbf{S}'_1$.

<u>Case</u> (i): $\mathbf{S}'_1 = \mathbf{x}/\mathbf{y}.\mathbf{Q}$

We prove $\mathbf{x}/\mathbf{y}.\mathbf{Q};\mathbf{S}_2 \ \approx\ \mathbf{S}_2;\mathbf{x}/\mathbf{y}.\mathbf{Q}$ by induction on the structure of $\mathbf{S}_2$:

Base step: $\mathbf{x}/\mathbf{y}.\mathbf{Q};\ \mathbf{x}_1/\mathbf{y}_1.\mathbf{Q}_1$

The premise implies $\mathbf{var}(\mathbf{Q}) \cap \mathbf{var}(\mathbf{Q}_1) = \varnothing$ and $\tilde{x} \cap \mathbf{var}(\mathbf{Q}_1) = \tilde{x}_1 \cap \mathbf{var}(\mathbf{Q}) = \varnothing$

$\mathbf{WP}(\mathbf{x}/\mathbf{y}.\mathbf{Q};\ \mathbf{x}_1/\mathbf{y}_1.\mathbf{Q}_1,\ \mathbf{G}(\mathbf{w}))$

$\iff\ \mathbf{WP}(\mathbf{x}/\mathbf{y}.\mathbf{Q},\ \mathbf{WP}(\mathbf{x}_1/\mathbf{y}_1.\mathbf{Q}_1,\ \mathbf{G}(\mathbf{w})))$

$\iff\ \exists \mathbf{x}.\mathbf{Q}\ \wedge\ \forall \mathbf{x}.(\mathbf{Q} \Rightarrow (\exists \mathbf{x}_1.\mathbf{Q}_1\ \wedge\ \forall \mathbf{x}_1(\mathbf{Q}_1 \Rightarrow \mathbf{G}(\mathbf{w}))))$

$\iff\ \exists \mathbf{x}_1.\mathbf{Q}_1\ \wedge\ \forall \mathbf{x}_1(\mathbf{Q}_1 \Rightarrow (\exists \mathbf{x}.\mathbf{Q}\ \wedge\ \forall \mathbf{x}.(\mathbf{Q} \Rightarrow \mathbf{G}(\mathbf{w}))))$

$\iff\ \mathbf{WP}(\mathbf{x}_1/\mathbf{y}_1.\mathbf{Q}_1,\ \mathbf{WP}(\mathbf{x}/\mathbf{y}.\mathbf{Q},\ \mathbf{G}(\mathbf{w})))$

$\iff\ \mathbf{WP}(\mathbf{x}_1/\mathbf{y}_1.\mathbf{Q}_1;\ \mathbf{x}/\mathbf{y}.\mathbf{Q},\ \mathbf{G}(\mathbf{w}))$ as required.

The various induction steps are simple applications of the induction hypothesis. For example, suppose $\mathbf{S}'_1 = \mathbf{S}';\mathbf{S}''$. The premise implies $\mathbf{S}'$ and $\mathbf{S}_2$ have no variables in common and $\mathbf{S}''$ and $\mathbf{S}_2$ have no variables in common. Hence by the induction hypothesis:

$\mathbf{S}';\mathbf{S}'';\mathbf{S}_2 \ \approx\ \mathbf{S}';\mathbf{S}_2;\mathbf{S}'' \ \approx\ \mathbf{S}_2;\mathbf{S}';\mathbf{S}'' \ \approx\ \mathbf{S}_2;\mathbf{S}'_1$.

This step shows why induction could not be used on the problem as originally stated since if $\mathbf{S}_1$ preserves the values of variables in $\mathbf{v}$ and $\mathbf{S}_1 = \mathbf{S}';\mathbf{S}''$ then we cannot assume $\mathbf{S}'$ and $\mathbf{S}''$ also preserve the values of variable in $\mathbf{v}$. (For instance if variable $\mathbf{x}$ is in $\mathbf{v}$ then **x:=x+1; x:=x−1** preserves $\mathbf{x}$ but **x:=x+1** by itself does not).

The Method of "Postponed Obligations"

In this chapter we will use the following notation for operations on stacks:
If $\mathbf{A}$ is a stack, $\mathbf{e}$ an expression and $\mathbf{x}$ a variable we write:
$\mathbf{A} \leftarrow \mathbf{e}$ for $\mathbf{A}{:=}\mathbf{e}{::}\mathbf{A}$ ie $\mathbf{A}{:=}\langle \mathbf{e} \rangle \& \mathbf{A}$ (pushing the value of $\mathbf{e}$ onto stack $\mathbf{A}$)
$\mathbf{x} \leftarrow \mathbf{A}$ for $\mathbf{x}{:=}\mathbf{hd}(\mathbf{A});\ \mathbf{A}{:=}\mathbf{tl}(\mathbf{A})$ ie $\mathbf{x}{:=}\mathbf{A}[1];\ \mathbf{A}{:=}\mathbf{A}[2..]$ (popping a value off $\mathbf{A}$ into variable $\mathbf{x}$)

The next theorem introduces the postponed obligations technique discussed above. This method can give a more efficient iterative form than that described in the previous chapter (which from now on will be described as the "direct method", following the terminology of [Bird 77]) but it does impose more conditions on the recursive procedure. Some special cases of the method of postponed obligations are discussed in [Manna & Waldinger 78].

We will be concentrating on procedures of the following general form: (which is called "cascade recursion"):

proc $\mathbf{F}(\mathbf{x}) \equiv$ **if** $\mathbf{B}$ **then** $\mathbf{S}_1$; $\mathbf{F}(\mathbf{g}_1(\mathbf{x}))$; $\mathbf{S}_2$; $\mathbf{F}(\mathbf{g}_2(\mathbf{x}))$; $\mathbf{S}_3$
    else $\mathbf{S}_4$ **fi**.

where $\mathbf{S}_1$, $\mathbf{S}_2$, $\mathbf{S}_3$, and $\mathbf{S}_4$ contain no calls to $\mathbf{F}$.

Theorem: If $\mathbf{x}$ is invariant over $\mathbf{S}_2$ then $\mathbf{F}(\mathbf{x})$ is equivalent to:

proc $\mathbf{F}(\mathbf{x}) \equiv$ **var** $\mathbf{A}{:=}\langle \mathbf{0},\mathbf{x} \rangle$;
  while $\mathbf{A} \neq \langle\rangle$ **do**
    $\langle \mathbf{m},\mathbf{x} \rangle \leftarrow \mathbf{A}$;
    if $\mathbf{m}=\mathbf{0} \ \rightarrow$ **if** $\mathbf{B}$ **then** $\mathbf{S}_1$; $\mathbf{A} \leftarrow \langle \mathbf{3},\mathbf{x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0},\mathbf{g}_2(\mathbf{x}) \rangle$;
        $\mathbf{A} \leftarrow \langle \mathbf{2},\mathbf{x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0},\mathbf{g}_1(\mathbf{x}) \rangle$
       else $\mathbf{S}_4$ **fi**
     $\square$ $\mathbf{m}=\mathbf{2} \ \rightarrow \ \mathbf{S}_2$
     $\square$ $\mathbf{m}=\mathbf{3} \ \rightarrow \ \mathbf{S}_3$ **fi** **od**.

where $\mathbf{A}$ and $\mathbf{m}$ are new variables which only exist within the procedure.

Here, the stack $\mathbf{A}$ records the list of postponed obligations which are gradually "worked through" or "discharged" by the body of the loop. Discharging some obligations may result on other obligations being pushed onto the stack. For example, in this case if $\mathbf{B}$ is true then discharging $\langle \mathbf{0},\mathbf{x} \rangle$ causes $\mathbf{S}_1$ to be executed and four other items to be put on the stack. The stack contains pairs of numbers, the first is a "mark" which records what statement has been postponed ($\mathbf{m{=}0}$ for a recursive call, $\mathbf{m{=}2}$ for $\mathbf{S}_2$ and $\mathbf{m{=}3}$ for $\mathbf{S}_3$) and the second is the value of $\mathbf{x}$ for which the statement is to be executed. Note that the statements and the condition $\mathbf{B}$ may contain other variables as well as $\mathbf{x}$ and $\mathbf{x}$ may be a list of variables. Discharging an obligation to execute $\mathbf{F}$ will result in further obligations being added to the list if $\mathbf{B}$ holds.

Proof: The proof uses the induction rules for recursion and iteration:

Let $\mathbf{DO}{=}\ \underline{\textbf{while}}\ \mathrm{A}\neq\langle\rangle\ \underline{\textbf{do}}$
$\quad\langle\mathbf{m,x}\rangle\leftarrow\mathbf{A};$
$\quad\underline{\textbf{if}}\ \mathbf{m{=}0}\ \rightarrow\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathrm{S}_1;\ \mathbf{A}\leftarrow\langle\mathbf{3,x}\rangle;\ \mathbf{A}\leftarrow\langle\mathbf{0,g_2(x)}\rangle;$
$\qquad\qquad\mathbf{A}\leftarrow\langle\mathbf{2,x}\rangle;\ \mathbf{A}\leftarrow\langle\mathbf{0,g_1(x)}\rangle$
$\qquad\quad\underline{\textbf{else}}\ \mathrm{S}_4\ \underline{\textbf{fi}}$
$\quad\ \Box\ \mathbf{m{=}2}\ \rightarrow\ \mathrm{S}_2$
$\quad\ \Box\ \mathbf{m{=}3}\ \rightarrow\ \mathrm{S}_3\ \underline{\textbf{fi}}\ \underline{\textbf{od}}.$

Let $\mathbf{DO}' = \underline{\textbf{while}}\ \mathrm{A}\neq\langle\rangle\ \underline{\textbf{do}}$
$\quad\langle\mathbf{m,x}\rangle\leftarrow\mathbf{A};$
$\quad\underline{\textbf{if}}\ \mathbf{m{=}0}\ \rightarrow\ \mathbf{F(x)}$
$\quad\ \Box\ \mathbf{m{=}2}\ \rightarrow\ \mathrm{S}_2$
$\quad\ \Box\ \mathbf{m{=}3}\ \rightarrow\ \mathrm{S}_3\ \underline{\textbf{fi}}\ \underline{\textbf{od}}.$

<u>**Claim:**</u> $\mathbf{DO}\approx\mathbf{DO}'$. Proof is by the induction rules.
(i) Prove $\mathbf{DO}^n\leqslant\mathbf{DO}'\forall\mathbf{n}<\omega$. The induction step is:
$\mathbf{DO}^{n+1}\leqslant\underline{\textbf{if}}\ \mathrm{A}\neq\langle\rangle\ \underline{\textbf{then}}\ \langle\mathbf{m,x}\rangle\leftarrow\mathbf{A};$
$\qquad\underline{\textbf{if}}\ \mathbf{m{=}0}\ \rightarrow\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathrm{S}_1;\ \mathbf{A}\leftarrow\langle\mathbf{3,x}\rangle;\ \mathbf{A}\leftarrow\langle\mathbf{0,g_2(x)}\rangle;$
$\qquad\qquad\quad\mathbf{A}\leftarrow\langle\mathbf{2,x}\rangle;\ \mathbf{A}\leftarrow\langle\mathbf{0,g_1(x)}\rangle$
$\qquad\qquad\underline{\textbf{else}}\ \mathrm{S}_4\ \underline{\textbf{fi}}$
$\qquad\ \Box\ \mathbf{m{=}2}\ \rightarrow\ \mathrm{S}_2$
$\qquad\ \Box\ \mathbf{m{=}3}\ \rightarrow\ \mathrm{S}_3\ \underline{\textbf{fi}};\ \mathbf{DO}'\ \underline{\textbf{fi}}$ by induction hypothesis.

Push the $\mathbf{DO}'$ inside both $\underline{\textbf{if}}$s and push the statement $\langle\mathbf{m,x}\rangle\leftarrow\mathbf{A}$ inside the outer $\underline{\textbf{if}}$, changing the tests to $\mathrm{A}[\mathbf{1}][\mathbf{1}]{=}\mathbf{0}$ etc.:
$\ \approx\ \underline{\textbf{if}}\ \mathrm{A}\neq\langle\rangle$
$\quad\underline{\textbf{then}}\ \underline{\textbf{if}}\ \mathrm{A}[\mathbf{1}][\mathbf{1}]{=}\mathbf{0}\ \rightarrow\ \langle\mathbf{m,x}\rangle\leftarrow\mathbf{A};\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathrm{S}_1;\ \mathbf{A}\leftarrow\langle\mathbf{3,x}\rangle;\ \mathbf{A}\leftarrow\langle\mathbf{0,g_2(x)}\rangle;$
$\qquad\qquad\qquad\mathbf{A}\leftarrow\langle\mathbf{2,x}\rangle;\ \mathbf{A}\leftarrow\langle\mathbf{0,g_1(x)}\rangle;\ \mathbf{DO}'$
$\qquad\qquad\quad\underline{\textbf{else}}\ \mathrm{S}_4;\ \mathbf{DO}'\ \underline{\textbf{fi}}$
$\qquad\Box\ \mathrm{A}[\mathbf{1}][\mathbf{1}]{=}\mathbf{2}\ \rightarrow\ \langle\mathbf{m,x}\rangle\leftarrow\mathbf{A};\ \mathrm{S}_2;\ \mathbf{DO}'$
$\qquad\Box\ \mathrm{A}[\mathbf{1}][\mathbf{1}]{=}\mathbf{3}\ \rightarrow\ \langle\mathbf{m,x}\rangle\leftarrow\mathbf{A};\ \mathrm{S}_3;\ \mathbf{DO}'\ \underline{\textbf{fi}}\ \underline{\textbf{fi}}$

Now push the statement $\langle \mathbf{m,x} \rangle \leftarrow \mathbf{A}$ inside the inner <u>**if**</u>, replacing **B** by $\mathbf{B[A[1][2]/x]}$:

$\approx$ <u>**if**</u> $\mathbf{A} \neq \langle\rangle$
<u>**then**</u> <u>**if**</u> $\mathbf{A[1][1]=0} \rightarrow$ <u>**if**</u> $\mathbf{B[A[1][2]/x]}$
<u>**then**</u> $\langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle;$
$\mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{0,g_1(x)} \rangle;\ \mathbf{DO}'$
<u>**else**</u> $\langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_4;\ \mathbf{DO}'$ <u>**fi**</u>
$\square\ \mathbf{A[1][1]=2} \rightarrow \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_2;\ \mathbf{DO}'$
$\square\ \mathbf{A[1][1]=3} \rightarrow \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_3;\ \mathbf{DO}'$ <u>**fi**</u> <u>**fi**</u>

To prove this consider the cases:
<u>Case</u> (i): $\mathbf{A[1][1]=0} \wedge \neg\mathbf{B[A[1][2]/x]}$.
<u>Case</u> (ii): $\mathbf{A[1][1]=2}$.
<u>Case</u> (iii): $\mathbf{A[1][1]=3}$. -these are trivial.
<u>Case</u> (iv): $\mathbf{A[1][1]=0} \wedge \mathbf{B[A[1][2]/x]}$:
Then $\langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{0,g_1(x)} \rangle;\ \mathbf{DO}'$
$\approx \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle;$
$\mathbf{A} \leftarrow \langle \mathbf{0,g_1(x)} \rangle;\ \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \{\mathbf{m=0} \wedge \mathbf{B}\};$ <u>**if**</u> $\mathbf{m=0} \rightarrow \mathbf{F(x)}\ \square \ldots$ <u>**fi**</u>; $\mathbf{DO}'$
by unrolling first step of loop. **A** is a new variable so does not occur in **B** so assignments to **A** do not affect **B**.
$\approx \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle;\ \mathbf{F(g_1(x))};\ \mathbf{DO}'$
$\approx \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{F(g_1(x))};\ \mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle;\ \mathbf{DO}'$
since **F** preserves **x**, **m**, and **A**.
$\approx \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{F(g_1(x))};\ \mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle;\ \mathbf{S}_2;\ \mathbf{DO}'$
by unrolling first step of $\mathbf{DO}'$ and simplifying.
$\approx \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{F(g_1(x))};\ \mathbf{S}_2;\ \mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle;\ \mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle;\ \mathbf{DO}'$
since $\mathbf{S}_2$ preserves **x**, **m**, and **A**.
$\approx \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{F(g_1(x))};\ \mathbf{S}_2;\ \mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle;\ \mathbf{F(g_2(x))};\ \mathbf{DO}'$ as above.
$\approx \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{F(g_1(x))};\ \mathbf{S}_2;\ \mathbf{F(g_2(x))};\ \mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle;\ \mathbf{DO}'$
$\approx \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \mathbf{S}_1;\ \mathbf{F(g_1(x))};\ \mathbf{S}_2;\ \mathbf{F(g_2(x))};\ \mathbf{S}_3;\ \mathbf{DO}'$
$\approx \langle \mathbf{m,x} \rangle \leftarrow \mathbf{A};\ \{\mathbf{m=0} \wedge \mathbf{B}\};\ \mathbf{F(x)};\ \mathbf{DO}'$ by folding $\mathbf{F(x)}$.
$\approx \mathbf{DO}'$ by rolling the first step of the loop.

Putting these together gives:

$\mathbf{DO}^{n+1} \leqslant \underline{\mathbf{if}}$ $\mathbf{A} \neq \langle\rangle$ $\underline{\mathbf{then}}$ $\underline{\mathbf{if}}$ $\mathbf{A[1][1]=0} \ \rightarrow\ \underline{\mathbf{if}}$ $\mathbf{B[A[1][2]/x]}$ $\underline{\mathbf{then}}$ $\mathbf{DO'}$

$\underline{\mathbf{else}}$ $\mathbf{DO'}$ $\underline{\mathbf{fi}}$

$\Box\ \mathbf{A[1][1]=2} \ \rightarrow\ \mathbf{DO'}$

$\Box\ \mathbf{A[1][1]=3} \ \rightarrow\ \mathbf{DO'}$ $\underline{\mathbf{fi}}$ $\underline{\mathbf{fi}}$

Hence $\mathbf{DO}^{n+1} \leqslant \underline{\mathbf{if}}$ $\mathbf{A} \neq \langle\rangle$ $\underline{\mathbf{then}}$ $\mathbf{DO'}$ $\underline{\mathbf{fi}}$

But $\mathbf{\{A=\langle\rangle\}} \ \approx\ \mathbf{\{A=\langle\rangle\};\ DO'}$ so

$\mathbf{DO}^{n+1} \leqslant \mathbf{DO'}$.

Hence by induction $\mathbf{DO} \leqslant \mathbf{DO'}$.

For the converse we first prove $\mathbf{F(x)}^n\mathbf{;DO} \leqslant \mathbf{A} \leftarrow \langle\mathbf{0,x}\rangle\mathbf{;DO}$ by induction on $\mathbf{n}$:

The induction step is:

$\mathbf{F(x)}^{n+1}\mathbf{;DO} \ \approx\ \underline{\mathbf{if}}$ $\mathbf{B}$ $\underline{\mathbf{then}}$ $\mathbf{S}_1\mathbf{;\ F(g_1(x))}^n\mathbf{;\ S}_2\mathbf{;\ F(g_2(x))}^n\mathbf{;\ S}_3$

$\underline{\mathbf{else}}$ $\mathbf{S}_4$ $\underline{\mathbf{fi}}$$\mathbf{;\ DO}$

By case analysis on $\mathbf{B}$ we prove:

$\mathbf{F(x)}^n \leqslant \mathbf{A} \leftarrow \langle\mathbf{0,x}\rangle\mathbf{;\ DO}$ for all $\mathbf{n} < \omega$.

Hence by induction rule for recursion:

$\mathbf{F(x)} \leqslant \mathbf{A} \leftarrow \langle\mathbf{0,x}\rangle\mathbf{;\ DO}$

Now we claim $\mathbf{DO'}^n \leqslant \mathbf{DO}$ by induction on $\mathbf{n}$:

$\mathbf{DO'}^{n+1} \leqslant \underline{\mathbf{if}}$ $\mathbf{A} \neq \langle\rangle$

$\underline{\mathbf{then}}$ $\langle\mathbf{m,x}\rangle \leftarrow \mathbf{A;}$

$\underline{\mathbf{if}}$ $\mathbf{m=0} \ \rightarrow\ \mathbf{F(x);\ DO}$

$\Box\ \mathbf{m=2} \ \rightarrow\ \mathbf{S}_2\mathbf{;\ DO}$

$\Box\ \mathbf{m=3} \ \rightarrow\ \mathbf{S}_3\mathbf{;\ DO}$ $\underline{\mathbf{fi}}$ $\underline{\mathbf{fi}}$.

(by induction hypothesis and forward expansion).

$\leqslant \underline{\mathbf{if}}$ $\mathbf{A} \neq \langle\rangle$

$\underline{\mathbf{then}}$ $\langle\mathbf{m,x}\rangle \leftarrow \mathbf{A;}$

$\underline{\mathbf{if}}$ $\mathbf{m=0} \ \rightarrow\ \mathbf{A} \leftarrow \langle\mathbf{0,x}\rangle\mathbf{;\ DO}$

$\Box\ \mathbf{m=2} \ \rightarrow\ \mathbf{A} \leftarrow \langle\mathbf{2,x}\rangle\mathbf{;\ DO}$

$\Box\ \mathbf{m=3} \ \rightarrow\ \mathbf{A} \leftarrow \langle\mathbf{3,x}\rangle\mathbf{;\ DO}$ $\underline{\mathbf{fi}}$ $\underline{\mathbf{fi}}$. by above.

$\leqslant \underline{\mathbf{if}}$ $\mathbf{A} \neq \langle\rangle$ $\underline{\mathbf{then}}$ $\mathbf{DO}$ $\underline{\mathbf{fi}}$

$\approx\ \mathbf{DO}$ since $\mathbf{\{A=\langle\rangle\};\ DO} \ \approx\ \mathbf{\{A=\langle\rangle\}}$.

Hence $\mathbf{DO}' \leqslant \mathbf{DO}$ by induction rule for iteration.
Hence $\mathbf{DO}' \approx \mathbf{DO}$ as required.

We can simplify the resulting iterative procedure if we note that $\langle \mathbf{0,g_1(x)} \rangle$ is pushed onto the stack only to be popped off again on the next iteration of the **while** loop. We will avoid this by showing that the **while** loop is equivalent to the following nested loop:

$\underline{\textbf{while}}$ A$\neq \langle\rangle$ $\underline{\textbf{do}}$
 $\langle \mathbf{m,x} \rangle \leftarrow \mathbf{A}$;
 $\underline{\textbf{if}}$ $\mathbf{m=0} \ \rightarrow \ \underline{\textbf{while}}$ **B** $\underline{\textbf{do}}$ S_1; $\mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle$; $\mathbf{x:=g_1(x)}$ $\underline{\textbf{od}}$; S_4
  $\Box$ $\mathbf{m=2} \ \rightarrow \ S_2$
  $\Box$ $\mathbf{m=3} \ \rightarrow \ S_3$ $\underline{\textbf{fi}}$ $\underline{\textbf{od}}$.

First apply entire loop unfolding after the statement $\mathbf{A} \leftarrow \langle \mathbf{0,g_1(x)} \rangle$, the inner **if** statement becomes:

$\underline{\textbf{if}}$ **B** $\underline{\textbf{then}}$ S_1; $\mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_1(x)} \rangle$;
  $\underline{\textbf{while}}$ $A \neq \langle\rangle \ \wedge \ \mathbf{A[1][1]=0} \ \wedge \ \mathbf{B[A[1][2]/x]}$ $\underline{\textbf{do}}$
   $\langle \mathbf{m,x} \rangle \leftarrow \mathbf{A}$;
   $\underline{\textbf{if}}$ $\mathbf{m=0} \ \rightarrow \ \underline{\textbf{if}}$ **B** $\underline{\textbf{then}}$ S_1; $\mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle$;
    $\mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_1(x)} \rangle$
      $\underline{\textbf{else}}$ S_4 $\underline{\textbf{fi}}$
   $\Box$ $\mathbf{m=2} \ \rightarrow \ S_2$
   $\Box$ $\mathbf{m=3} \ \rightarrow \ S_3$ $\underline{\textbf{fi}}$ $\underline{\textbf{od}}$
 $\underline{\textbf{else}}$ S_4 $\underline{\textbf{fi}}$

$\approx$ $\underline{\textbf{if}}$ **B** $\underline{\textbf{then}}$ S_1; $\mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_1(x)} \rangle$;
  $\underline{\textbf{while}}$ $A \neq \langle\rangle \ \wedge \ \mathbf{A[1][1]=0} \ \wedge \ \mathbf{B[A[1][2]/x]}$ $\underline{\textbf{do}}$
   $\langle \mathbf{m,x} \rangle \leftarrow \mathbf{A}$; $\{\mathbf{m=0} \ \wedge \ \mathbf{B}\}$;
   S_1; $\mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle$;
   $\mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_1(x)} \rangle$ $\underline{\textbf{od}}$
  $\underline{\textbf{else}}$ S_4 $\underline{\textbf{fi}}$

Convert the **while** loop to a **do...od** loop, apply proper inversion and simplify to get:

$\approx$ $\underline{\textbf{do}}$ S_1; $\mathbf{A} \leftarrow \langle \mathbf{3,x} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{0,g_2(x)} \rangle$; $\mathbf{A} \leftarrow \langle \mathbf{2,x} \rangle$; $\mathbf{x:=g_1(x)}$;
  $\underline{\textbf{if}}$ $\neg\mathbf{B}$ $\underline{\textbf{then}}$ $\underline{\textbf{exit}}$ $\underline{\textbf{fi}}$ $\underline{\textbf{od}}$;
 $\mathbf{A} \leftarrow \langle \mathbf{m,x} \rangle$

The result is therefore:

proc $\mathbf{F(x)} \equiv$
  var $\mathbf{A:=\langle 0,x\rangle;}$
  while $\mathbf{A \neq \langle\rangle}$ **do**
    $\mathbf{\langle m,x\rangle \leftarrow A;}$
    if $\mathbf{m=0} \ \rightarrow$ **if** $\mathbf{B}$ **then** **do** $\mathbf{S_1;\ A\leftarrow \langle 3,x\rangle;\ A\leftarrow \langle 0,g_2(x)\rangle;\ A\leftarrow \langle 2,x\rangle;\ x:=g_1(x);}$
        if $\mathbf{\neg B}$ **then** **exit** **fi** **od**;
          $\mathbf{A\leftarrow \langle m,x\rangle}$
      else $\mathbf{S_4}$ **fi**
    $\Box$ $\mathbf{m=2} \ \rightarrow \ \mathbf{S_2}$
    $\Box$ $\mathbf{m=3} \ \rightarrow \ \mathbf{S_3}$ **fi** **od**.

After the loop we have $\mathbf{\neg B} \ \wedge \ \mathbf{m=0}$ so by loop unrolling we can replace $\mathbf{A\leftarrow \langle m,x\rangle}$ by:
$\mathbf{A\leftarrow \langle m,x\rangle;\ \langle m,x\rangle \leftarrow A;\ S_4} \ \approx \ \mathbf{S_4}$. Then take the statement $\mathbf{S_4}$ outside the **if**, apply proper inversion to the inner loop and convert it to a **while** loop to get the final form:

proc $\mathbf{F(x)} \equiv$
  var $\mathbf{A:=\langle 0,x\rangle;}$
  while $\mathbf{A \neq \langle\rangle}$ **do**
    $\mathbf{\langle m,x\rangle \leftarrow A;}$
    if $\mathbf{m=0} \ \rightarrow$ **while** $\mathbf{B}$ **do** $\mathbf{S_1;\ A\leftarrow \langle 3,x\rangle;\ A\leftarrow \langle 0,g_2(x)\rangle;\ A\leftarrow \langle 2,x\rangle;\ x:=g_1(x)}$ **od**; $\mathbf{S_4}$
    $\Box$ $\mathbf{m=2} \ \rightarrow \ \mathbf{S_2}$
    $\Box$ $\mathbf{m=3} \ \rightarrow \ \mathbf{S_3}$ **fi** **od**.

The method can be applied to more complicated procedures, (with more than two inner calls for example) provided: (a) the values of the parameters are invariant over any statements between inner calls, (b) the guards on any conditional statements <u>within</u> the procedure are invariant over the body and (c) no inner call is within a loop or nested recursion.

The condition (b) means that backward expansion can be applied to any conditional statements within the body which can then be collected into a single conditional at the beginning so that the procedure can be written in the form:

proc $\mathbf{F(x)} \equiv$ **if** $\mathbf{B_1} \ \rightarrow \ \mathbf{S_{1,1};\ F(g_{1,1}(x));\ S_{1,2};\ F(g_{1,2}(x));\ \ldots\ ;\ S_{1,n(1)}}$
  $\Box$ $\mathbf{B_2} \ \rightarrow \ \mathbf{S_{2,1};\ F(g_{2,1}(x));\ S_{2,2};\ F(g_{2,2}(x));\ \ldots\ ;\ S_{2,n(2)}}$
  $\Box$ $\ldots$
  $\Box$ $\mathbf{B_m} \ \rightarrow \ \mathbf{S_{m,1};\ F(g_{m,1}(x));\ S_{m,2};\ F(g_{m,2}(x));\ \ldots\ ;\ S_{m,n(m)}}$ **fi**.

where the $\mathbf{S}_{i,j}$ contain no calls to $\mathbf{F}$.

The $\mathbf{i}$th guarded statement has $\mathbf{n(i)}$ statements separating $\mathbf{n(i)-1}$ inner calls.
Condition (a) implies that $\mathbf{x}$ is invariant over $\mathbf{S}_{1,2},\ldots,\mathbf{S}_{1,n(1)-1},\mathbf{S}_{2,2},\ldots,\mathbf{S}_{m,n(m)-1}$.

Note that procedures with **var** parameters will not work in this scheme in general: though if each $\mathbf{g}_{ij}$ is the identity function for the **var** parameters then the **var** parameters can be replaced by global variables and the above scheme can be applied to the new procedure. We can apply the scheme to parameterless procedures, in this case the stack will contain integers which record the next statement to be executed and the result is very similar to the result of the direct method of recursion removal.

General Tree-Traversal Algorithm

In this section we will consider the special case when $\mathbf{S}_2$ =**skip** and $\mathbf{S}_4$ =**skip**, ie transformations of the following procedure:

$$\underline{\textbf{proc}}\ \mathbf{F(x)} \equiv\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathbf{S}_1;\ \mathbf{F(g_1(x))};\ \mathbf{F(g_2(x))};\ \mathbf{S}_3\ \underline{\textbf{fi}}.$$

This general scheme is used in several sorting and tree-traversal algorithms.

An example of this kind of procedure is:

$$\underline{\textbf{proc}}\ \mathbf{P(x,y)} \equiv\ \underline{\textbf{if}}\ \mathbf{x \neq y}\ \underline{\textbf{then}}\ \mathbf{z:=m(x,y);\ T(x,z);\ T(z+1,y);\ M(x,y)}\ \underline{\textbf{fi}}.$$

Here if we further specify $\mathbf{m(x,y)}$=
$\mathbf{(x+y)/2}\supseteq$ and specify $\mathbf{M(x,y)}$ to be a procedure which merges
the two sorted arrays $\mathbf{k[x..z]}$ and $\mathbf{k[z+1..y]}$ where $\mathbf{z}$=
$\mathbf{(x+y)/2}\supseteq$ then we get a simple version of
Mergesort. **P** first sorts the left half of $\mathbf{k[x..y]}$, then the right half, then merges the halves. Another interpretation gives Quicksort: $\mathbf{M(x,y)}$ is **skip** and $\mathbf{m(x,y)}$ is a function which produces some value $\mathbf{z}$ such that $\mathbf{x \leqslant z < y}$. **m** also re-arranges array **k** such that no value in $\mathbf{k[x..z]}$ exceeds a value in $\mathbf{k[z+1..y]}$. To sort **k** we perform $\mathbf{m(x,y)}$ and then sort the left and right halves of **k**.
There are two obvious special cases of this procedure:

Case (1): $\mathbf{S}_3$ =**skip**. (as for the Quicksort example) ie:

$$\underline{\textbf{proc}}\ \mathbf{F(x)} \equiv\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathbf{S}_1;\ \mathbf{F(g_1(x))};\ \mathbf{F(g_2(x))}\ \underline{\textbf{fi}}.$$

Replace the parameter by a global variable:

$$\underline{\textbf{proc}}\ \mathbf{F(x)} \equiv\ \underline{\textbf{var}}\ \mathbf{A:=\langle\rangle;\ F;\ Z}\ \underline{\textbf{where}}$$
$$\mathbf{F} \equiv\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathbf{S}_1;\ \mathbf{A \leftarrow x;\ x:=g_1(x);\ F;\ x \leftarrow A;\ A \leftarrow x;\ x:=g_2(x);\ F;\ x \leftarrow A}\ \underline{\textbf{fi}}.$$

Note that **F** does not have to preserve the value of **x** so we can simplify
$\mathbf{A \leftarrow x;\ x:=g_2(x);\ F;\ x \leftarrow A}$ to $\mathbf{x:=g_2(x);\ F}$ to give:

$$\mathbf{F} \equiv\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathbf{S}_1;\ \mathbf{A \leftarrow x;\ x:=g_1(x);\ F;\ x \leftarrow A;\ x:=g_2(x);\ F}\ \underline{\textbf{fi}}.$$

Regularise **F** and add **G**:

$\mathbf{F} \equiv$ **if** **B** **then** $\mathbf{S_1}$; **A**$\leftarrow$**x**; **x**:=$\mathbf{g_1}$(**x**); **F**; **G**
else **/F** **fi**.
$\mathbf{G} \equiv$ **x**$\leftarrow$**A**; **x**:=$\mathbf{g_2}$(**x**); **F**.
/F $\equiv$ **skip**.

There are only two non-terminal calls to **F** (in **F(x)** and in **F**) and the one in **F(x)** can be characterised by **A**= $\langle\rangle$ (since **F** preserves **A**) so we can regularise **/F** to give:

proc **F(x)** $\equiv$ **var** **A**:=$\langle\rangle$; **F** **where**
$\mathbf{F} \equiv$ **if** **B** **then** $\mathbf{S_1}$; **A**$\leftarrow$**x**; **x**:=$\mathbf{g_1}$(**x**); **F**
else **/F** **fi**.
$\mathbf{G} \equiv$ **x**$\leftarrow$**A**; **x**:=$\mathbf{g_2}$(**x**); **F**.
/F $\equiv$ **if** **A**= $\langle\rangle$ **then** **Z** **else** **G** **fi**.

Remove the recursion in the usual and simplify to get :

proc **F(x)** $\equiv$ **var** **A**:=$\langle\rangle$;
do **while** **B** **do** $\mathbf{S_1}$; **A**$\leftarrow$**x**; **x**:=$\mathbf{g_1}$(**x**) **od**;
if **A**= $\langle\rangle$ **then** **exit** **fi**;
x$\leftarrow$**A**; **x**:=$\mathbf{g_2}$(**x**) **od**.

Convert the inner **while** loop to a **do...od** loop, apply absorption to the rest of the body of the outer loop (which makes the body reducible) and remove the double loop (by false iteration):

proc **F(x)** $\equiv$ **var** **A**:=$\langle\rangle$;
do **if** **B** **then** $\mathbf{S_1}$; **A**$\leftarrow$**x**; **x**:=$\mathbf{g_1}$(**x**)
elsf **A**$\neq$ $\langle\rangle$ **then** **x**$\leftarrow$**A**; **x**:=$\mathbf{g_2}$(**x**)
else **exit** **fi** **od**.

Case (2): $\mathbf{S_1}$ =**skip**. ie:

proc **F(x)** $\equiv$ **if** **B** **then** **F**($\mathbf{g_1}$(**x**)); **F**($\mathbf{g_2}$(**x**)); $\mathbf{S_3}$ **fi**.

Remove the parameter:

proc **F(x)** $\equiv$ **var** **A**:=$\langle\rangle$; **F**; **Z** **where**
$\mathbf{F} \equiv$ **if** **B** **then** **A**$\leftarrow$**x**; **x**:=$\mathbf{g_1}$(**x**); **F**; **x**$\leftarrow$**A**; **A**$\leftarrow$**x**; **x**:=$\mathbf{g_2}$(**x**); **F**; **x**$\leftarrow$**A**; $\mathbf{S_3}$ **fi**.

In this case the last call of **P** is not in a terminal position so the first method above will not work. We need to stack the argument for both calls, since it is needed after the second call, since $\mathbf{S_3}$ may well use the value of **x**:

proc **F(x)** $\equiv$ **var** **A**:=$\langle\rangle$; **F**; **Z** **where**
$\mathbf{F} \equiv$ **if** **B** **then** **A**$\leftarrow$**x**; **x**:=$\mathbf{g_1}$(**x**); **F**; **G**
else **/F** **fi**.
$\mathbf{G} \equiv$ **x**$\leftarrow$**A**; **A**$\leftarrow$**x**; **x**:=$\mathbf{g_2}$(**x**); **F**; **H**.
$\mathbf{H} \equiv$ **x**$\leftarrow$**A**; $\mathbf{S_3}$.

$/F \equiv$ **skip.**

Here we have two inner calls of **F** so we need to add another stack to record which is the current return point. In fact we will merge the stacks into one since we **pop** from them and **push** to them in the same places:

```
F ≡ if B then A← ⟨0,x⟩; x:=g₁(x); F; {A[1][1]=0}; G
      else /F fi.
G ≡ x←A; A← ⟨1,x⟩; x:=g₂(x); F; {A[1][1]=1}; H.
H ≡ x←A; S₃.
```

This leads to the procedure:

```
proc F(x) ≡
  var A:=⟨⟩;
  do if B then A← ⟨0,x⟩; x:=g₁(x)
         else do if A= ⟨⟩ then exit(2) fi;
                 ⟨m,x⟩ ←A;
                 if m=0 then A← ⟨1,x⟩; x:=g₂(x); exit
                      else S₃ fi od fi od.
```

Applying the method of postponed obligations gives a different form of the solution:

```
proc F(x) ≡
  var A:=⟨⟨0,x⟩⟩;
  while A≠ ⟨⟩ do
    ⟨m,x⟩ ←A;
    if m=0 then if B then A← ⟨1,x⟩; A← ⟨0,g₁(x)⟩; A← ⟨0,g₂(x)⟩ fi
      else S₃ fi od.
```

With the simplification given there we get the more efficient form:

```
proc F(x) ≡
  var A:=⟨⟨0,x⟩⟩;
  while A≠ ⟨⟩ do
    ⟨m,x⟩ ←A;
    if m=0 then while B do A← ⟨1,x⟩; A← ⟨0,g₁(x)⟩; x:=g₂(x) od
      else S₃ fi od.
```

Another solution is to use the stack to record postponed obligations to execute $\mathbf{S}_3$. We postpone all executions of $\mathbf{S}_3$ until the last moment: this only works when **B** contains no variables (other than **x**) which are modified by $\mathbf{S}_3$. We get:

$\underline{\textbf{proc}}$ **F(x)** $\equiv$
$\underline{\textbf{var}}$ **A:=**$\langle\rangle$**;**
G(x); $\underline{\textbf{while}}$ A$\neq\langle\rangle$ $\underline{\textbf{do}}$ **x**$\leftarrow$**A;** S_3 $\underline{\textbf{od}}$. $\underline{\textbf{where}}$
$\underline{\textbf{proc}}$ **G(x)** $\equiv$
$\underline{\textbf{if}}$ **B** $\underline{\textbf{then}}$ **A**$\leftarrow$**x; G(**g_2**(x)); G(**g_1**(x))** $\underline{\textbf{fi}}$.

If the condition on **B** is relaxed then the transformation may not work, for example:

$\underline{\textbf{proc}}$ **F(x)** $\equiv$ $\underline{\textbf{if}}$ $\neg$(**z=1** $\vee$ **x=2** $\vee$ **x=** $-$**1**) $\underline{\textbf{then}}$ **F(x+1); F(x**$-$**2); z:=1** $\underline{\textbf{fi}}$.

If **z=0** initially then:

F(0) $\approx$ **F(1); F(**$-$**2); z:=1** $\approx$ **F(2); F(**$-$**1); z:=1; F(**$-$**2); z:=1**
$\approx$ **F(**$-$**1); z:=1; F(**$-$**2); z:=1**
$\approx$ **z:=1; F(**$-$**2); z:=1**
$\approx$ **z:=1.**

However if we try to apply the transformation we get:

$\underline{\textbf{proc}}$ **F(x)** $\equiv$
$\underline{\textbf{var}}$ **A:=**$\langle\rangle$**;**
G(x); $\underline{\textbf{while}}$ A$\neq\langle\rangle$ $\underline{\textbf{do}}$ **x**$\leftarrow$**A; z:=1** $\underline{\textbf{od}}$. $\underline{\textbf{where}}$
$\underline{\textbf{proc}}$ **G(x)** $\equiv$
$\underline{\textbf{if}}$ $\neg$(**z=1** $\vee$ **x=2** $\vee$ **x=** $-$**1**) $\underline{\textbf{then}}$ **A**$\leftarrow$**x; G(x+1); G(x**$-$**2)** $\underline{\textbf{fi}}$.

So if **z=0** initially then:

F(0) $\approx$ $\underline{\textbf{var}}$ **A:=**$\langle\rangle$**; G(0);** $\underline{\textbf{while}}$ A$\neq\langle\rangle$ $\underline{\textbf{do}}$ **x**$\leftarrow$**A; z:=1** $\underline{\textbf{od}}$
$\approx$ **...A**$\leftarrow$**x; G(1); G(**$-$**2);...** note that **z** is invariant over **G** so:
$\approx$ **...G(1); {z=0}; G(**$-$**2);...**
$\approx$ **...G(1); {z=0}; A**$\leftarrow$**x; G(**$-$**1); G(**$-$**4);...**
$\approx$ **...G(**$-$**1); {z=0}; G(**$-$**4);...**
$\approx$ **...G(**$-$**3); {z=0}; G(**$-$**6);...**

etc. So the call to **G(0)** does not terminate. This is why we needed the extra condition that **B** must not reference any variable (other than **x**) modified in S_3.

For the proofs of these transformations we will use the induction rule for recursion:
Let **DO=** $\underline{\textbf{while}}$ A$\neq\langle\rangle$ $\underline{\textbf{do}}$ **x**$\leftarrow$**A;** S_3 $\underline{\textbf{od}}$.
Claim: $\mathbf{F(x)}^k \approx$ **A:=**$\langle\rangle$**;** $\mathbf{G(x)}^k$**; DO**. The induction step is:

$\mathbf{F(x)}^{k+1} \approx$ $\underline{\textbf{if}}$ **B** $\underline{\textbf{then}}$ $\mathbf{F(g_1(x))}^k$**;** $\mathbf{F(g_2(x))}^k$**;** S_3 $\underline{\textbf{fi}}$
$\approx$ $\underline{\textbf{if}}$ **B** $\underline{\textbf{then}}$ **A**$''$**:=**$\langle$**x**$\rangle$**;** $\mathbf{F(g_1(x))}^k$**; A:=**$\langle\rangle$**;** $\mathbf{G(g_2(x))}^k$**; DO; x**$\leftarrow$**A**$''$**;** S_3 $\underline{\textbf{fi}}$

by the induction hypothesis since **F** preserves **x** but **DO** may not do so.