CHAPTER EIGHT

Non-Linear Recursion and the Derivation of Algorithms

Introduction

We have shown that with a linear recursive function or procedure the transformation to an iterative form does not require a protocol stack but may require a parameter stack. In this chapter we examine some cases of non-linear recursion (ie where the body of the procedure contains more than one recursive call). Note that for non-linear functions both protocol and parameter stacks may be required; for example Strong [Strong 71] showed that the scheme:

```
funct F(x) ≡
 if B(x) then φ(F(M(x)),F(N(x)))
      else H(x) fi.
```

cannot (without further restrictions) be transformed to iterative form. Both the protocol stack and a parameter stack are needed in general.
The function:

```
funct fusc(n) ≡
 if n=1  →  1
 □ n>1 ∧ even(n)  →  fusc(n/2)
 □ n>1 ∧ odd(n)  →  fusc((n−1)/2)+fusc(n+1)/2) fi.
```

is defined in terms of various linear combinations of **fusc(i)** and **fusc(i−1)**. We have dealt with similar functions by defining a more general function which returns these two values, say:

funct Gfusc(n) ≡ ⟨fusc(n),fusc(n−1)⟩.

This was used to derive a linear form of the function for Fibonacci series. Here we use a different method: define a function which computes a given linear combination of **fusc(i)** and **fusc(i−1)**:

funct F(m,a,b) ≡ a.fusc(m)+b.fusc(m−1).

Thus **fusc(m)=F(m,1,0)** (for **m**$\geqslant$**2**). We want to transform **F** so that it is defined in terms of **F** rather than **fusc**. Take out the case **m=2** and split the other case:

```
funct F(m,a,b) ≡
 if m=2
  then a.fusc(2/2)+b.fusc(1)
  else if even(m) then a.fusc(m)+b.fusc(m−1)
       else a.fusc(m)+b.fusc(m−1) fi fi.
```

Unfolding the calls of **fusc** (using the fact that **even(m)⇒odd(m+1)** etc.):

```
funct F(m,a,b) ≡
  if m=1
   then a+b
   else if even(m)
      then a.fusc(m/2)+b.(fusc(m/2)+fusc(m/2−1))
      else a.(fusc((m−1)/2)+fusc((m+1)/2))+b.fusc((m−1)/2) fi fi.
```

These are linear combinations of **fusc(i)** and **fusc(i+1)** with **i<m** so by the theorem on recursive implementation of specifications **F** is equivalent to:

```
funct F(m,a,b) ≡
  if m=2
   then a+b
   else if even(m) then F(m/2, a+b, b)
          else F((m+1)/2, a, a+b) fi fi.
```

This is tail-recursive so can be transformed to the iterative form:

```
funct F(m,a,b) ≡
 ⌈ while m≠2 do
    if even(m) then m:=m/2; a:=a+b
         else m:=(m+1)/2; b:=a+b fi od;
  a+b⌋.
```

Using this to calculate the values of **fusc** for **m**$\geqslant$**2** (since **fusc(m)** = **F(m,1,0)**) gives the following log time version of **fusc**:

```
funct fusc(m) ≡
  if m=1 then 1
     else ⌈ a:=1; b:=0;
        while m≠2 do
          if even(m) then m:=m/2; a:=a+b
               else m:=(m+1)/2; b:=a+b fi od;
        a+b⌋ fi.
```

Theorem: A generalisation of **fib** is:

```
funct H(m,r,s) ≡
  if B(m)
   then if B(δ(m)) then φ(H(δ(m),r,s), H(δ^2(m),r,s))
          else r fi
   else s fi.
```

If we set $\delta(\mathbf{m})=\mathbf{m}-\mathbf{1}$, $\mathbf{B(m)} \Longleftrightarrow \mathbf{m>1}$, $\phi(\mathbf{x,y})=\mathbf{x+y}$
then $\mathbf{H(m,0,1)} = \mathbf{fib(m)}$.

We will prove that this is equivalent to:

```
funct G(m,r,s) ≡
  if B(m) then ⌈ m:=δ(m);
              while B(m) do
                 m:=δ(m); ⟨r,s⟩:=⟨φ(r,s),r⟩ od;
              r⌋
     else s fi.
```

Provided δ, ϕ and **B** are determinate and have no side effects and ϕ is independent of **H** and δ.
(ϕ and **B** need not be defined everywhere though).
If any of δ, ϕ or **B** are not determinate then **G** will be a refinement of **H**.

The condition ensures that **H** is determinate and has no side effects, in turn this ensures that sequences of calls to **H** (which assign the result to variables which are not global in δ, ϕ and **B**) may be carried out in any order, and that two consecutive calls may be replaced by a single call and an assignment. Hence for example:

$\mathbf{a{:=}H(\delta(m),r,s);\ b{:=}H(m,r,s);\ c{:=}H(\delta(m),r,s)}$
$\approx\ \mathbf{a{:=}H(\delta(m),r,s);\ c{:=}H(\delta(m),r,s);\ b{:=}H(m,r,s)}$
$\approx\ \mathbf{a{:=}H(\delta(m),r,s);\ c{:=}a;\ b{:=}H(m,r,s)}$
$\approx\ \mathbf{a{:=}H(\delta(m),r,s);\ b{:=}H(m,r,s);\ c{:=}a}$

Proof: Note that $\mathbf{H(m,r,s)}$ is defined (in the general case) in terms of $\mathbf{H(\delta(m),r,s)}$ and $\mathbf{H(\delta^2(m),r,s)}$ (the parameters **r** and **s** are never changed so may just as well be global variables. They are made use of in **G**). Our linear form of the similar function **fib** suggests defining an auxiliary function which returns both $\mathbf{H(m,r,s)}$ and $\mathbf{H(\delta(m),r,s)}$. What we actually do is define a procedure which sets the variables $\langle\mathbf{r,s}\rangle$ to $\langle\mathbf{H(m,r,s),\ H(\delta(m),r,s)}\rangle$ except when $\neg\mathbf{B(m)}$ holds when we only set **r** (since in this case $\mathbf{H(\delta(m),r,s)}$ may not even be defined). We define:

```
proc F(m) ≡
  if B(m)
   then if B(δ(m)) then ⟨r,s⟩:=⟨H(m,r,s), H(δ(m),r,s)⟩ fi
   else r:=s fi
```

Then $\mathbf{H(m,r,s)}\ \approx\ \underline{\mathbf{funct}}\ \mathbf{H_1(m,r,s)} \equiv \lceil\mathbf{F(m);\ r}\rfloor$.

<u>Claim:</u> $\mathbf{F(m)}\ \approx\ \mathbf{F_1(m)}$ where:

funct $\mathbf{F_1(m)} \equiv$
 if $\mathbf{B(m)}$ **then** **var** **i:=0;**
  while $\mathbf{B(\delta(m))}$ **do**
   $\mathbf{m{:=}\delta(m);\ i{:=}i+1}$ **od;**
  for **j:=1** **to** **i** **step** **1** **do**
   $\mathbf{\langle r,s\rangle{:=}\langle \phi(r,s),r\rangle}$ **od**
 else **r:=s** **fi.**
$=$ **if** $\mathbf{B(m)}$ **then** **var** **i:=0; DO; FOR**
 else **r:=s** **fi.**

Proof of Claim:
We use the induction rule for recursion to prove $\mathbf{F}^n\mathbf{(m)} \leqslant \mathbf{F_1(m)}\ \forall \mathbf{n} < \omega$.
This uses the fact that:
$\langle \mathbf{r,s}\rangle{:=}\langle \mathbf{H}^{n+1}\mathbf{(m,r,s)},\ \mathbf{H}^{n+1}(\delta\mathbf{(m),r,s)}\rangle$
 $\approx\ \langle \mathbf{r,s}\rangle{:=}\langle \phi(\mathbf{H}^n(\delta\mathbf{(m),r,s)},\ \mathbf{H}^n(\delta^2\mathbf{(m),r,s)}),\ \mathbf{H}^{n+1}(\delta\mathbf{(m),r,s)}\rangle$
 $\leqslant \kappa\kappa \langle \mathbf{r,s}\rangle{:=}\langle \phi(\mathbf{H}^{n+1}(\delta\mathbf{(m),r,s)},\ \mathbf{H}^n(\delta^2\mathbf{(m),r,s)}),\ \mathbf{H}^{n+1}(\delta\mathbf{(m),r,s)}\rangle$
 $\leqslant \kappa\kappa \langle \mathbf{r,s}\rangle{:=}\langle \mathbf{H}^{n+1}(\delta\mathbf{(m),r,s)},\ \mathbf{H}^n(\delta^2\mathbf{(m),r,s)}\rangle;\ \langle \mathbf{r,s}\rangle{:=}\langle \phi\mathbf{(r,s),\ r}\rangle$
where we have replaced two calls of $\mathbf{H}^{n+1}(\delta\mathbf{(m),r,s)}$ by a single call.
Now apply the induction hypothesis to get:
$\mathbf{F}^{n+1}\mathbf{(m)} \leqslant$ **if** $\mathbf{B(m)}$
 then **if** $\mathbf{B(\delta(m))}$
  then $\mathbf{m{:=}\delta(m);}$
   if $\mathbf{B(\delta(m))}$
   then $\mathbf{m{:=}\delta(m);}$
    var **i:=0; DO; FOR;**
    $\mathbf{\langle r,s\rangle{:=}\langle \phi(r,s),r\rangle}$ **fi**
   else $\mathbf{\langle r,s\rangle{:=}\langle \phi(r,s),r\rangle}$ **fi**
  else $\mathbf{\langle r,s\rangle{:=}\langle \phi(r,s),r\rangle}$ **fi**
 else **r:=s** **fi**
 $\approx$ **if** $\mathbf{B(m)}$
 then **var** **i:=0; DO; FOR**
 else **r:=s** **fi**
by loop rolling (twice).
 $\approx\ \mathbf{F_1(m)}$.
Conversely:

if $\mathbf{B(m)} \leqslant$ **if** $\mathbf{B(m)}$
then $\mathbf{i{:=}0;\ DO}^n\mathbf{;\ FOR}$ **then** **if** $\mathbf{B(\delta(m))}$
else r:=s **fi** **then** $\mathbf{m{:=}\delta(m);}$
$\langle \mathbf{r,s} \rangle \mathbf{:=} \langle \mathbf{H}^n\mathbf{(m,r,s)},\ \mathbf{H}^n\mathbf{(\delta(m),r,s)} \rangle;$
$\langle \mathbf{r,s} \rangle \mathbf{:=} \langle \phi\mathbf{(r,s),r} \rangle$ **fi**
else r:=s **fi**

by induction hypothesis.

$\leqslant$ **if** $\mathbf{B(m)}$
then **if** $\mathbf{B(\delta(m))}$
then $\mathbf{m{:=}\delta(m);}$
$\langle \mathbf{r,s} \rangle \mathbf{:=} \langle \phi(\mathbf{H}^n\mathbf{(m,r,s)}, \mathbf{H}^n\mathbf{(\delta(m),r,s)}),\ \mathbf{H}^{n+1}\mathbf{(\delta(m),r,s)}) \rangle$ **fi**
else r:=s **fi**

where we have refined a single call $\mathbf{H}^n\mathbf{(\delta(m),r,s)}$ to two calls $\mathbf{H}^n\mathbf{(\delta(m),r,s)}$ and $\mathbf{H}^{n+1}\mathbf{(\delta(m),r,s)}$.

$\approx\ \mathbf{F}_1^{n+1}\mathbf{(m)}$ by folding.

which proves the claim.

Now since ϕ is independent of $\mathbf{B}$ and δ we may apply loop overlapping to $\mathbf{F}_1$ to get:

$\mathbf{F_1(m)}\ \approx\ \mathbf{G'(m)}$ where

proc $\mathbf{G'(m)} \equiv$
if $\mathbf{B(m)}$ **then** **while** $\mathbf{B(\delta(m))}$ **do**
$\mathbf{m{:=}\delta(m);}\ \langle \mathbf{r,s} \rangle \mathbf{:=} \langle \phi\mathbf{(r,s),r} \rangle$ **od**
else r:=s **fi**.

which may be written (using proper inversion):

proc $\mathbf{G'(m)} \equiv$
if $\mathbf{B(m)}$ **then** $\mathbf{m{:=}\delta(m);}$
while $\mathbf{B(m)}$ **do**
$\langle \mathbf{r,s} \rangle \mathbf{:=} \langle \phi\mathbf{(r,s),r} \rangle\mathbf{;\ m{:=}\delta(m)}$ **od**
else r:=s **fi**.

Hence $\mathbf{H(m,r,s)}\ \approx$

funct $\mathbf{H_2(m,r,s)} \equiv$
$\lceil \mathbf{G'(m);\ r} \rfloor$.

which leads to our function $\mathbf{G}$ above.

(The subsidiary function $\mathbf{G}^*$ implements the **while** loop in the procedure $\mathbf{G'}$).

If we add the extra condition that ϕ and δ are defined everywhere then we can add extra statements to $\mathbf{G'}$ to give:

```
proc G′(m) ≡
  if B(m) then m:=δ(m);
               while B(m) do
                 ⟨r,s⟩:=⟨φ(r,s),r⟩; m:=δ(m) od;
               ⟨r,s⟩:=⟨φ(r,s),r⟩;
               r:=s
           else r:=s fi.
```
(since we don't care what the final value of s is).

```
≈ if B(m) then m:=δ(m); ⟨r,s⟩:=⟨φ(r,s),r⟩;
        while B(m) do
          ⟨r,s⟩:=⟨φ(r,s),r⟩; m:=δ(m) od fi; r:=s.
```
since ϕ is independent of **B** and δ.

```
≈ while B(m) do ⟨r,s⟩:=⟨φ(r,s),r⟩; m:=δ(m) od; r:=s.
```
by loop rolling.

Thus we get the more compact (though less efficient) version of **G**:

```
funct G₁(m,r,s) ≈
  if B(m) then G₁(δ(m),φ(r,s),r)
    else s fi.
```

Arithmetization of the Flow of Control

In the case of a "cascade recursion" for which all the parameters can be replaced by global variables, but the transformation to iterative form requires a protocol stack (which will be a stack of marks to decide the next action) we can replace the stack of marks by an integer (as described in Chapter Eight). The stack operations become integer operations which can frequently be simplified. Bauer & Wossner [Bauer & Wossner 80] describe one such transformation under the name "Arithmetization of the Flow of Control". For example, consider the function:

```
funct F(n,x) ≡
  if n>0 then F(n−1, φ(n, F(n−1,x)))
    else x fi.
```
This can be transformed to:

```
funct F(n,x) ≡
  ⌈ for c:=2^n−1 step −1 to 1 do
     x:=φ(ntz(c)+1,x) od;
   x⌋, where
   funct ntz(c) ≡ "the number of trailing zeros in the binary
          representation of c".
```

A similar example is described by Partsch & Pepper [Partsch & Pepper 76].

Proof: A procedural version of **F** is:

```
proc F(n,x) ≡
  if n>0 then F(n−1,x); x:=φ(n,r); F(n−1,x)
     else r:=x fi.
```

The parameter **x** is now fixed so we can use function inversion on **n** to remove both parameters:

```
proc F(n,x) ≡  F; Z. where
  F ≡  if n>0 then n:=n−1; F; x:=φ(n+1,r); F; n:=n+1
        else r:=x fi.
```

We can now remove the variable **r** from **F** by replacing it by **x** so that **F** now returns its result in **x** which is then assigned to **r** in **F(n,x)**:

```
proc F(n,x) ≡  F; r:=x; Z. where
  F ≡  if n>0 then n:=n−1; F; x:=φ(n+1,x); F; n:=n+1 fi.
```

The theorem on recursion removal gives the iterative version:

```
proc F(n,x) ≡
  Stack:=⟨⟩;
  while n>0 do n:=n−1; Stack←1 od;
     while Stack≠ ⟨⟩ do
   d←Stack;
   if d=1 then x:=φ(n+1,x); Stack←0;
        while n>0 do n:=n−1; Stack←1 od
      else n:=n+1 fi od; r:=x.
```

If we represent the stack by an integer **c** so that the digits in the binary representation of **c** are the elements of **Stack** then:

Stack:=⟨⟩ becomes **c:=1**
Stack= ⟨⟩ becomes **c=1**
Stack←d becomes **c:=2.c+d**
d←Stack becomes **⟨c,d⟩:=⟨c÷2⟩**

The statement **while n>0 do n:=n−1; Stack←1 od** becomes $\mathbf{c{:=}c.2^n+2^n-1;\ n{:=}0}$
ie $\mathbf{c{:=}(c+1).2^n-1;\ n{:=}0}$

We get:
```
proc F(n,x) ≡
    c:=2^(n+1)−1; n:=0;
    while c≠1 do
      ⟨c,d⟩:=⟨c÷2⟩;
      if d=1 then x:=φ(n+1,x); c:=2.c;
           c:=(c+1).2^n−1; n:=0
         else n:=n+1 fi od; r:=x.
```
Apply entire loop unroll to the **else** part:
```
  proc F(n,x) ≡
    c:=2^(n+1)−1; n:=0;
    while c≠1 do
      ⟨c,d⟩:=⟨c÷2⟩;
      if d=1 then x:=φ(n+1,x); c:=2.c;
           c:=(c+1).2^n−1; n:=0
         else n:=n+1;
                while c≠1 ∧ even(c) do
                  c:=c/2; n:=n+1 od fi od; r:=x.
```
We now backward expand the **if**, changing the test from **d=1** to **odd(c)**. The statement **⟨c,d⟩:=⟨c÷2⟩** can now be replaced by **c:=c/2** when **c** is even and **c:=(c−1)/2** when **c** is odd. The variable **d** disappears.
```
  proc F(n,x) ≡
    c:=2^(n+1)−1; n:=0;
    while c≠1 do
      if odd(c) then c:=(c−1)/2; x:=φ(n+1,x); c:=2.c;
                 c:=(c+1).2^n−1; n:=0
          else c:=c/2; n:=n+1;
            while c≠1 ∧ even(c) do
              c:=c/2; n:=n+1 od fi od; r:=x.
```
When **c** is even the statements **c:=c/2; n:=n+1; while c≠1 ∧ even(c) do c:=c/2; n:=n+1 od** can be replaced by $\mathbf{c{:=}c/2^{ntz(c)};\ n{:=}n+ntz(c)}$.
Also the assignments $\mathbf{c{:=}(c-1)/2;\ c{:=}2.c;\ c{:=}(c+1).2^n-1}$ may be merged to: $\mathbf{c{:=}c.2^n-1}$. We get:

```
proc F(n,x) ≡
  c:=2^(n+1)−1; n:=0;
  while c≠1 do
    if odd(c) then c:=c.2^n−1; x:=φ(n+1,x); n:=0
       else c:=c/2^ntz(c); n:=n+ntz(c) fi od; r:=x.
```

Now we apply loop unrolling to the first arm of the inner **if**, using the fact that if **c** is odd and greater than **1** then $\mathbf{c.2}^{n}\mathbf{-1}$ is even. This means that **c** will be odd after every loop and as **c** is odd initially we can prune the **if** in the loop to give:

```
proc F(n,x) ≡
  c:=2^(n+1)−1; n:=0;
  while c≠1 do
    c:=c.2^n−1; x:=φ(n+1,x); n:=0
    c:=c/2^ntz(c); n:=n+ntz(c) od; r:=x.
```

Convert the **while** loop to a **do...od** loop and simplify the body:

```
proc F(n,x) ≡
  c:=2^(n+1)−1; n:=0;
  do if c=1 then exit fi;
    c:=c.2^n−1; x:=φ(n+1,x);
    n:=ntz(c); c:=c/2^n od; r:=x.
```

Since $\mathbf{2}^{n+1}\mathbf{-1}$ is odd we can replace **n:=0** by **n:=ntz(c); c:=c/**$\mathbf{2}^{n}$ and then apply proper inversion to the loop:

```
proc F(n,x) ≡
  c:=2^(n+1)−1;
  do n:=ntz(c); c:=c/2^n;
    if c=1 then exit fi;
    c:=c.2^n−1; x:=φ(n+1,x) od; r:=x.
```

If **n0** is the initial value of **n** then we see that $\mathbf{c} \geqslant \mathbf{2}^{n0} \wedge \left(\mathbf{c}=\mathbf{2}^{n0} \Rightarrow \mathbf{c}/\mathbf{2}^{ntz(c)}=\mathbf{1}\right)$ is invariant over the loop. So we can move the test to the beginning of the loop (since we are not interested in the final values of **n** and **c**):

```
proc F(n,x) ≡
  c:=2^(n+1)−1; n0:=n;
  do if c=2^n0 then exit fi;
    n:=ntz(c); c:=c/2^n;
    c:=c.2^n−1; x:=φ(n+1,x) od; r:=x.
```

Now we can merge the assignments **c:=c/2^n; c:=c.2^n−1** to **c:=c−1** and replace **c** by **c−2^{n0}** since for **c>2^{n0}**, **ntz(c)=ntz(c−2^{n0})**: (and **c⩾2^{n0}** is invariant over the loop so **c>2^{n0}** is true within the loop.

```
proc F(n,x) ≡
  c:=2^(n+1)−1; n0:=n; c:=c−2^n0;
  do if c=0 then exit fi;
    n:=ntz(c);
    c:=c−1; x:=φ(n+1,x) od; r:=x.
```

Finally we replace the **do...od** loop by a **while** and do some simplification to get:

```
proc F(n,x) ≡
  c:=2^n−1;
  while if c≠0 do
    x:=φ(ntz(c)+1,x) c:=c−1; od; r:=x.
```

The loop may be replaced by a **for** loop thus:

```
proc F(n,x) ≡
  for c:=2^n−1 to 1 step −1 do
    x:=φ(ntz(c)+1,x) od;
  r:=x.
```

With this version of the program we can see precisely what the sequence of values of the first argument of ϕ is. The sequence of values which was originally defined by the flow of control, is now defined by an arithmetic function, whence the term “Arithmetisation of the flow of control”. This illustrates Arsac's comment about the value of transforming recursion to iteration in giving insights and suggesting new strategies.

Note that **ntz(2^n−c)=ntz(c)** for **1⩽c<2^n** so we could replace **c** by **2^n−c** which runs from **1** to **2^n**. Hence our decreasing **for** loop could be replaced by an increasing one:

```
proc F(n,x) ≡
  for c:=1 to 2^n−1 step 1 do
    x:=φ(ntz(c)+1,x) od;
  r:=x.
```

Range-of-Values Tabulation

Suppose we have a recursive function **h(x)** and a well-order on the type of **x**. If the “argument on termination” $\mathbf{x}_0$ is known in advance (or can be calculated easily from **x**) then we can build up a table of the values of **h** from $\mathbf{x}_0$ to **x** and in doing so we can replace the inner recursive calls of **h** by references to this table. Note that this will only work if the parameters for the inner calls are always smaller than the current parameter. An added advantage is that the table can be made a global variable (if we start it from the smallest possible parameter) which will be preserved between calls to the function. Then repeated function calls with the same (or smaller) arguments can be read straight off the table without requiring further computation. This type of function is often called a "memorised" function [Abelson & Sussman 84] since previously calculated values of the function are "remembered" in the table. For this to work our function must be determinate, have no side effects, and depend only on its argument (ie not on any global variable). In other words, it must be a "pure" function.

A procedure for tabulating the function **h** from **h(a)** to **h(b)** in table **T[a..b]** is:

```
proc tab_h(a,b) ≡
  for i:=a to b step 1 do
    T[i]:=h(i) od.
```

A function which returns the list of values of **h** from **h(a)** to **h(b)** is:

```
funct tab_h(b) ≡
  if b=a then ⟨h(a)⟩
     else h(b)::tab_h(b−1) fi.
```

where $\langle \mathbf{x} \rangle$ is the sequence with one element **x** and **x::list** is **list** with **x** added to the front ie $\langle \mathbf{x} \rangle$**&list**.

Applying function inversion to this gives:

```
funct tab_h(b) ≡
  t(a,⟨h(a)⟩). where
  funct t(y,z) ≡
    if y≠b then t(y+1, h(y+1)::z)
       else z fi.
```

which can also be written in the form:

```
funct tab_h(b) ≡
  t(a,⟨⟩). where
  funct t(y,z) ≡
    if y≠b+1 then t(y+1, h(y)::z)
         else z fi.
```

Here we have the invariant $\forall \mathbf{i}.(\mathbf{a} \leqslant \mathbf{i} < \mathbf{y} \Rightarrow \mathbf{z}[\mathbf{i}-\mathbf{a}+1]=\mathbf{h}(\mathbf{i}))$ within the inner function so if we unfold **h** within the inner function we can replace the recursive calls of **h** by references to **z**. Then **t** becomes independent of **h**. In procedural form this is:

```
proc tab_h(b) ≡
  for i:=a to b step 1 do
    {∀i.(a⩽j<i ⇒ T[j]=h(j))};
    T[i]:=h(i) od.
```

We unfold the call **h(i)** using the assertion. Suppose **h** is:

```
funct h(x) ≡ E_h.
```

We get:

```
proc tab_h(b) ≡
  for i:=a to b step 1 do
    T[i]:=E_h[T[e]/h(e)][i/x] od.
```

where **T[e]** replaces **h(e)** for every expression **e**.
Now we can replace **h(i)** by:

```
funct h(x) ≡
  ⌈tab_h(x); T[x]⌋. where
  proc tab_h(b) ≡
    for i:=a to b step 1 do
      T[i]:=E_h[T[e]/h(e)][i/x] od.
```

We can make this more efficient by adding another global variable **l** which records how far **T** has been filled in. At the start of the program we initialise **l** to **a−1** which sets up the invariant $\forall \mathbf{i}.(\mathbf{a} \leqslant \mathbf{i} \leqslant \mathbf{l} \Rightarrow \mathbf{T}[\mathbf{i}]=\mathbf{h}(\mathbf{i}))$. We can then replace **h** by:

```
funct h(x) ≡
 ⌈ if l<x
   then for i:=l+1 to x step 1 do
          T(i):=E_h[T[e]/h(e)] od;
        l:=x;
   T[x]⌋.
```

Example:

$\mathbf{q}_n$ denotes the number of ways a set of **n** elements may be partitioned, where $\mathbf{q}_0 = \mathbf{1}$,
$\mathbf{q}_{n+1} = \sum_{0 \leqslant s \leqslant n} (\mathbf{n}\ \mathbf{s}).\mathbf{q}_{n-s}$

where $(\mathbf{n}\ \mathbf{s}) = \dfrac{\mathbf{n!}}{\mathbf{s!.(n-s)!}}$
is the number of ways **s** elements may be chosen from a set of **n** different elements.

This is because with **n+1** elements we may choose **1** element to form the first partition in **(n+1 1)** ways and form the other partitions from the remaining **n** elements in $\mathbf{q}_n$ ways, choose **2** elements in **(n+1 2)** ways and form the other partitions in $\mathbf{q}_{n-1}$ ways, . . .

In general for $\mathbf{0 \leqslant s \leqslant n}$ we form the first partition from **s+1** elements in **(n+1 s+1)** ways and form the other partitions in $\mathbf{q}_{n+1-s-1} = \mathbf{q}_{n-s}$ ways.

A simple recursive function to evaluate this is:

```
funct part(n) ≡
  if n=0 then 1
    else ⌈ var sum:=0;
      for s:=0 to n step 1 do
       sum:=sum+comb(n,s).part(n−s) od;
      sum ⌋ fi. where
  funct comb(n,s) ≡
    fact(n)/(fact(s).fact(n−s)),
  funct fact(n) ≡
    if n=0 then 1
      else n.fact(n−1) fi.
```

However this is grossly inefficient; a call **part(5)** will lead to **31** recursive calls, **15** of which will call **comb**. Thus **fact** will be called **45** times. In general **part(n)** results in $\mathbf{2}^n\mathbf{-1}$ recursive calls and calls **fact** $\mathbf{3.2}^{n-1}$ times.

Note that for **n>0 part(n)** is defined in terms of all of **part(0)** to **part(n−1)** which suggests that it is a good candidate for range-of-values tabulation. If we initialise the program with **l:=1; T[1]:=1** then the transformation gives:

```
funct part(n) ≡
⌈ if l<n
   then for i:=l+1 to n step 1 do
      sum:=0;
      for s:=0 to i−1 step 1 do
        sum:=sum+comb(i−1,s).T[i−1−s] od;
      T[i]:=sum od fi;
  T[n] ⌋. where
  funct comb(n,s) ≡
    fact(n)/(fact(s).fact(n−s)),
```

```
funct fact(n) ≡
  if n=0 then 1
     else n.fact(n−1) fi.
```

This calls **comb n.(n+1)/2** times (for a call of **part(n)** when **l=1**) and therefore calls **fact 3.n.(n+1)/2** times. We can make this more efficient by noting that: **comb(n,s+1) = comb(n,s).(n−s)/(s+1)** for **0⩽s⩽n** so we can add a variable **c** and maintain the invariant **c=comb(i−1,s)** within the inner loop. This means that we can dispense with all calls to **comb** and **fact**. We get:

```
funct part(n) ≡
⌈ if l<n
  then for i:=l+1 to n step 1 do
     sum:=0; c:=1;
     for s:=0 to i−1 step 1 do
       sum:=sum+c.T[i−1−s]; c:=c.(i−1−s)/(s+1) od;
     T[i]:=sum od fi;
 T[n] ⌋.
```

Representing **s** by **t** where **t=i−1−s** gives the slightly improved version:

```
funct part(n) ≡
⌈ if l<n
  then for i:=l+1 to n step 1 do
      sum:=0; c:=1;
      for t:=i−1 to 0 step −1 do
        sum:=sum+c.T[t]; c:=c.t/(i−t) od;
      T[i]:=sum od fi;
 T[n] ⌋.
```

Disentanglement of the Control

Linear recursive routines do not require a protocol stack, more general recursive routines may require a protocol stack and a parameter stack. However, in some cases we can use function inversion to remove some or all of the parameters without needing the parameter stack. To see which parameters do not need stacks we add assignments to local variables until no parameter positions contain expressions. For example Ackermans function:

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

becomes:

```
funct Ack(m,n) ≡
  if m=0 then n+1
  elsf n=0 then ⌈var m1,n1; m1:=m−1; n1:=1; Ack(m1,n1)⌋
     else ⌈var m1,n1; n1:=n−1; n1:=Ack(m,n1); m1:=m−1; Ack(m1,n1)⌋ fi.
```

The result is said to be disentangled if none of the parameters (or auxiliary variables) is used both before and after the same recursive call. This means that their values need not be preserved by the function (or procedure) and so they can be replaced by global variables. **Ack(m,n)** is not disentangled since the parameter **m** is used after an inner call. However the function:

```
funct F(x) ≡
 if B(x) then G(x)
   else F(F(H(x))) fi
```

can be written in the disentangled form:

```
funct F(x) ≡
 if B(x) then G(x)
   else ⌈var x1,z1; x1:=H(x); z1:=F(x1); F(z1)⌋ fi.
```

We can use function inversion to disentangle a routine: if we deliver one of the parameters as an extra result of the function then we can apply the inverse of the function to this result to recover the original parameter. For **Ack(m,n)** we define a function **Ack*(m,n)** which returns the pair of integers $\langle$**m,Ack(m,n)**$\rangle$.

```
funct Ack(m,n) ≡
  Ack*(m,n)[2]. where
  funct Ack*(m,n) ≡
    if m=0 then ⟨m,n+1⟩
    elsf n=0 then ⌈var m1,n1; m1:=m−1; n1:=1; Ack*(m1,n1)⌋
       else ⌈var m1,m2,n1;
          n1:=n−1; ⟨m2,n1⟩:=Ack*(m,n1); m1:=m−1; Ack*(m1,n1)⌋ fi.
```

Initially the extra result has no use, but now we have $\mathbf{m}_2 = \mathbf{m}$ after the first inner call so we can replace the reference to **m** by $\mathbf{m}_2$ to give:

```
funct Ack(m,n) ≡
  Ack*(m,n)[2]. where
  funct Ack*(m,n) ≡
    if m=0 then ⟨m,n+1⟩
    elsf n=0 then ⌈var m₁,n₁; m₁:=m−1; n₁:=1; Ack*(m₁,n₁)⌋
             else ⌈var m₁,m₂,n₁;
        n₁:=n−1; ⟨m₂,n₁⟩:=Ack*(m,n₁); m₁:=m₂−1; Ack*(m₁,n₁)⌋ fi.
```

which is disentangled. Replacing the parameters of **Ack*** by global variables gives:

```
funct Ack(m,n) ≡
  Ack**()[2]. where
  funct A**() ≡
    if m=0 then ⟨m,n+1⟩
    elsf n=0 then ⌈m:=m−1; n:=1; A**()⌋
       else ⌈n:=n−1; ⟨m,n⟩:=A**; m:=m−1; A**()⌋ fi.
```

Finally we can replace **A**** by a procedure which returns the results in **m** and **n**:

```
funct Ack(m,n) ≡
 ⌈A**; n⌋. where
  proc A**  ≡
    if m=0 then n:=n+1
    elsf n=0 then m:=m−1; n:=1; A**
             else n:=n−1; A**; m:=m−1; A** fi.
```

Cascade Recursion

Consider the cascade recursion (where $\mathbf{k}_1$ and $\mathbf{k}_2$ are different):

```
funct F(x) ≡
  if B(x) then φ(F(k₁(x)), F(k₂(x)), E(x))
    else H(x) fi.
```

The detailed form is:

```
funct F(x) ≡
  if B(x) then ⌈var x1,x2,z1,z2;
        x1:=k1(x); z1:=F(x1); x2:=k2(x); z2:=F(x2);
        φ(z1, z2, E(x))⌋
     else H(x) fi.
```

x and $\mathbf{z}_1$ violate the condition. As above we return $\mathbf{z}_1$ as an additional result:

```
funct F(x) ≡
  F*(x)[2]. where
  funct F*(x) ≡
    if B(x) then ⌈var x1,x2,y1,y2,z1,z2;
          x1:=k1(x); ⟨y1,z1⟩:=F*(x1);
          x2:=k2(x); ⟨y2,z2⟩:=F*(x2);
          φ(z1, z2, E(x))⌋
       else ⟨x,H(x)⟩ fi.
```

Using the inverses $\mathbf{k}_1'$ and $\mathbf{k}_2'$ we replace **x** by $\mathbf{k}_1'(\mathbf{y}_1)$ or $\mathbf{k}_2'(\mathbf{y}_2)$ to disentangle **x**:

```
funct F(x) ≡
  F*(x)[2]. where
  funct F*(x) ≡
    if B(x) then ⌈var x1,x2,y1,y2,z1,z2;
          x1:=k1(x); ⟨y1,z1⟩:=F*(x1);
          x2:=k2(k'1(y1)); ⟨y2,z2⟩:=F*(x2);
          ⟨x, φ(z1, z2, E(k'2(y2))) ⟩⌋
       else ⟨x,H(x)⟩ fi.
```

$\mathbf{z}_1$ is still entangled and we cannot in general use function inversion to restore it so it needs a stack but **x** and the other local variables do not. Using a stack for $\mathbf{z}_1$ and replacing **x** and the local variables by global variables and writing $\mathbf{F}^*$ as a procedure gives:

```
funct F(x) ≡
 ⌈Stack:=⟨⟩; F**; x⌋. where
  F**  ≡  if B(x) then x:=k1(x); F**; x:=k2(k'1(x)); Stack←z; F**; z1 ←Stack; x:=k'2(x);
         z:=φ(z1, z, E(x)); ⌋
       else z:=H(x) fi.
```

where **x** is used for **x**, $\mathbf{y}_1$ and $\mathbf{y}_2$ and **z** is used for $\mathbf{z}_1$ and $\mathbf{z}_2$.

If $\mathbf{k}_1$ or $\mathbf{k}_2$ do not have inverses then $\mathbf{x}$ will need a stack as well: all the intermediate results can be put on the same stack if needed.

Nested Recursion:

Consider:

$\underline{\textbf{funct}}\ \mathbf{G(x)} \equiv$
$\quad \underline{\textbf{if}}\ \mathbf{B(x)}\ \underline{\textbf{then}}\ \phi(\ \mathbf{G}(\Psi(\ \mathbf{G}(\mathbf{k}_1(\mathbf{x})),\ \mathbf{k}_2(\mathbf{x})\ )),\ \mathbf{E(x)})$
$\qquad \underline{\textbf{else}}\ \mathbf{H(x)}\ \underline{\textbf{fi}}.$

In detailed form:

$\underline{\textbf{funct}}\ \mathbf{G(x)} \equiv$
$\quad \underline{\textbf{if}}\ \mathbf{B(x)}\ \underline{\textbf{then}} \ulcorner \underline{\textbf{var}}\ \mathbf{x}_1,\mathbf{x}_2,\mathbf{z}_1,\mathbf{z}_2;$
$\qquad\quad \mathbf{x}_1 := \mathbf{k}_1(\mathbf{x});\ \mathbf{z}_1 := \mathbf{G}(\mathbf{x}_1);$
$\qquad\quad \mathbf{x}_2 := \Psi(\mathbf{z}_1, \mathbf{k}_2(\mathbf{x}));\ \mathbf{z}_2 := \mathbf{G}(\mathbf{x}_2);$
$\qquad\quad \phi(\mathbf{z}_2, \mathbf{E(x)}) \lrcorner$
$\qquad \underline{\textbf{else}}\ \mathbf{H(x)}\ \underline{\textbf{fi}}.$

$\mathbf{z}_1$ and $\mathbf{z}_2$ are already disentangled, but in general we will not be able to reconstruct $\mathbf{x}$ by function inversion so it required a stack:

$\underline{\textbf{funct}}\ \mathbf{G(x)} \equiv$
$\quad \ulcorner\ \mathbf{Stack} := \langle\rangle;\ \mathbf{G};\ \mathbf{z} \lrcorner.\ \underline{\textbf{where}}$
$\quad \mathbf{G} \equiv\ \underline{\textbf{if}}\ \mathbf{B(x)}\ \underline{\textbf{then}}\ \mathbf{Stack} \leftarrow \mathbf{x};\ \mathbf{x} := \mathbf{k}_1(\mathbf{x});\ \mathbf{G};\ \mathbf{x} \leftarrow \mathbf{Stack};$
$\qquad\quad \mathbf{Stack} \leftarrow \mathbf{x};\ \mathbf{x} := \Psi(\mathbf{z}, \mathbf{k}_2(\mathbf{x}));\ \mathbf{G};\ \mathbf{x} \leftarrow \mathbf{Stack};\ \mathbf{z} := \phi(\mathbf{z}, \mathbf{E(x)}) \lrcorner$
$\qquad \underline{\textbf{else}}\ \mathbf{z} := \mathbf{H(x)}\ \underline{\textbf{fi}}.$

Reshaping the type of control flow

Sometimes reshaping the control flow allows us to disentangle more efficiently. For example if ρ is an associative operation, suppose we have:

$\underline{\textbf{funct}}\ \mathbf{F(x)} \equiv$
$\quad \underline{\textbf{if}}\ \mathbf{B(x)}\ \underline{\textbf{then}}\ \mathbf{F}(\mathbf{k}_1(\mathbf{x}))\ \rho\mathbf{F}(\mathbf{k}_2(\mathbf{x}))\ \rho\mathbf{E(x)}$
$\qquad\quad \underline{\textbf{else}}\ \mathbf{H(x)}\ \underline{\textbf{fi}}.$

To use functional embedding we need a close relation between $\mathbf{k}_1$ and $\mathbf{k}_2$ (eg $\mathbf{k}_1(\mathbf{x})=\mathbf{k}_2(\mathbf{k}_1(\mathbf{x}))$), to use range-of-values tabulation $\mathbf{k}_1$ and $\mathbf{k}_2$ also have to satisfy strict conditions. For many cases these conditions fail and we have to use a stack.

Since ρ is associative we can use re-bracketing on $\mathbf{F}$. Let $\mathbf{F}_1$ be identical to $\mathbf{F}$, then:

$\underline{\textbf{funct}}\ \mathbf{F(x)} \equiv$
$\quad \underline{\textbf{if}}\ \mathbf{B(x)}\ \underline{\textbf{then}}\ \mathbf{F(k_1(x))}\ \rho\mathbf{F_1(k_2(x))}\ \rho\mathbf{E(x)}$
$\qquad \underline{\textbf{else}}\ \mathbf{H(x)}\ \underline{\textbf{fi}}\textbf{.}$

and re-bracketing gives (where $\mathbf{e}$ is an identity element for ρ):

$\underline{\textbf{funct}}\ \mathbf{F(x)} \equiv$
$\quad \mathbf{G(x,e)}.\ \underline{\textbf{where}}$
$\quad \underline{\textbf{funct}}\ \mathbf{G(x,z)} \equiv$
$\qquad \underline{\textbf{if}}\ \mathbf{B(x)}\ \underline{\textbf{then}}\ \mathbf{G(k_1(x),\ (F_1(k_2(x))}\ \rho\mathbf{E(x))}\ \rho\mathbf{z)}$
$\qquad\quad \underline{\textbf{else}}\ \mathbf{H(x)}\ \rho\mathbf{z}\ \underline{\textbf{fi}}\textbf{.}$

The inner call of $\mathbf{F}$ can be replaced by $\mathbf{G(x,e)}$ since $\mathbf{F}_1$ is still equivalent to $\mathbf{F}$:

$\underline{\textbf{funct}}\ \mathbf{F(x)} \equiv$
$\quad \mathbf{G(x,e)}.\ \underline{\textbf{where}}$
$\quad \underline{\textbf{funct}}\ \mathbf{G(x,z)} \equiv$
$\qquad \underline{\textbf{if}}\ \mathbf{B(x)}\ \underline{\textbf{then}}\ \mathbf{G(k_1(x),\ G(k_2(x),\ e)}\ \rho\mathbf{E(x)}\ \rho\mathbf{z))}$
$\qquad\quad \underline{\textbf{else}}\ \mathbf{H(x)}\ \rho\mathbf{z}\ \underline{\textbf{fi}}\textbf{.}$

Now since ρ is associative $\mathbf{G(x,y)}\ \rho\mathbf{z} = \mathbf{G(x,\ y}\ \rho\mathbf{z)}$ (the proof is by the induction rule for recursion) so we can replace $\mathbf{G(k_1(x),\ G(k_2(x),\ e)}\ \rho\mathbf{E(x)}\ \rho\mathbf{z))}$ by $\mathbf{G(k_1(x),\ G(k_2(x),\ E(x))}\ \rho\mathbf{z))}$ to get the form:

$\underline{\textbf{funct}}\ \mathbf{F(x)} \equiv$
$\quad \mathbf{G(x,e)}.\ \underline{\textbf{where}}$
$\quad \underline{\textbf{funct}}\ \mathbf{G(x,z)} \equiv$
$\qquad \underline{\textbf{if}}\ \mathbf{B(x)}\ \underline{\textbf{then}}\ \mathbf{G(k_1(x),\ G(k_2(x),\ E(x))}\ \rho\mathbf{z))}$
$\qquad\quad \underline{\textbf{else}}\ \mathbf{H(x)}\ \rho\mathbf{z}\ \underline{\textbf{fi}}\textbf{.}$

which in detailed form is:

```
funct F(x) ≡
  G(x,e). where
  funct G(x,z) ≡
   if B(x) then ⌈var x1,x2,z1,z2,r1;
                 x1:=k2(x); z1:=E(x) ρz;
        r1:=G(x1,z1);
        x2:=k1(x); z2:=r1;
        G(x2,z2)⌋
       else H(x) ρz fi.
```

Now only **x** violates the condition and, as we have assumed nothing about $\mathbf{k}_1$ and $\mathbf{k}_2$ we must use a stack for **x**:

```
funct F(x) ≡
 ⌈Stack:=⟨⟩; z:=e; G; z⌋. where
  G ≡ if B(x)
    then Stack←x; x:=k2(x); z:=E(x) ρz; G;
      x←Stack; x:=k1(x); G
    else z:=H(x) ρz fi.
```

If we had an inverse for $\mathbf{k}_2$ we could use function inversion to remove this stack.

Recurrence Relations

The **r**-term recurrence:

```
funct F(m) ≡
 if m=m0  →  H0(m)
  □ m=m1  →  H1(m)
  □ ...
  □ m=m_{r-2}  →  H_{r-2}(m)
  □ m⩾m_{r-1}  →  φ(F(pred(m)),F(pred²(m)),...,F(pred^{r-1}(m)),m) fi.
```

(where $\mathbf{m}_i = \mathbf{succ}^i(\mathbf{m}_0)$ and **pred(succ(m))=m**), can be treated in a similar way to our derivation of a linear function for computing **fib**, (which is in fact a **3**-term recurrence).

As with our first treatment of **fib** we proceed by introducing the function:

```
funct G(m) ≡
 ⟨F(pred^{r-2}(m)),F(pred^{r-3}(m)),...,F(pred(m)),F(m)⟩.
```

which returns an $(\mathbf{r-1})$-tuple of results. Unfold $\mathbf{F(m)}$ and then transform $\mathbf{G(m)}$ so that it is defined in terms of $\mathbf{G(m-1)}$ (taking out the cases $\mathbf{m=m_0,\ldots,m=m}_{r-2}$). The result can be transformed to give the following linear iterative form for $\mathbf{F}$:

$$
\begin{array}{l}
\underline{\textbf{funct}}\ \mathbf{F(m)} \equiv \\
\quad \underline{\textbf{if}}\ \mathbf{m=m}_0 \ \rightarrow\ \mathbf{H}_0\mathbf{(m)} \\
\quad \square\ \mathbf{m=m}_1 \ \rightarrow\ \mathbf{H}_1\mathbf{(m)} \\
\quad \square\ \ldots \\
\quad \square\ \mathbf{m=m}_{r-2} \ \rightarrow\ \mathbf{H}_{r-2}\mathbf{(m)} \\
\quad \square\ \mathbf{m \geqslant m}_{r-1} \ \rightarrow\ \lceil \langle \mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_{r-1}\rangle := \langle \mathbf{H}_0(\mathbf{m}_0),\mathbf{H}_1(\mathbf{m}_1),\ldots,\mathbf{H}_{r-2}(\mathbf{m}_{r-2})\rangle; \\
\qquad \mathbf{y:=m}_{r-1}; \\
\qquad \underline{\textbf{while}}\ \mathbf{y \neq succ(m)}\ \underline{\textbf{do}} \\
\qquad \langle \mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_{r-1}\rangle := \langle \mathbf{v}_2,\mathbf{v}_3\ldots,\mathbf{v}_{r-1},\phi(\mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_{r-1},\mathbf{y})\rangle; \\
\qquad \mathbf{y:=succ(y)}\ \underline{\textbf{od}}; \\
\qquad \mathbf{v}_{r-1} \rfloor\ \underline{\textbf{fi}}.
\end{array}
$$

This can also be thought of as a form of range-of-values tabulation in which the table is not preserved between calls: so, as only the top $\mathbf{r-1}$ elements of the table are needed to compute the next element, we only need to save $\mathbf{r-1}$ elements in the table. These elements are stored in the variables $\mathbf{v}_1, \ldots, \mathbf{v}_{r-1}$.

The Derivation of Algorithms

In this section and the next chapter we give examples of the transformation of a specification into a complete, efficient algorithm which implements the specification, using the results developed so far. We will also discuss the application of these results to software maintenance, particularly in discovering the specification of a section of code. We will describe our current research at Durham into the analysis of IBM Assembler programs and the derivation of **Z** specifications from the code.

A Traditional Example: Set $\mathbf{x}$ to $\mathbf{n!}$ ($\mathbf{n}$ factorial).

This is a simple example of transforming a specification to a recursive procedure, making use of our theorem on recursive implementation of specifications. We will carry out the transformation in some detail so as to illustrate the stages in the process. These general stages apply to a great many algorithm developments and many of the detailed steps can be automated.

<u>Defn:</u> **n!=1** if **n=0**

n!=n.(n−1)! otherwise.

Our specification is simply: **S ≡ x:=n!**

First we split this on the cases occurring in the definition to separate the base case and general case:

≈ **<u>if</u> n=0 <u>then</u> x:=n!** by splitting a tautology.
<u>else</u> x:=n! <u>fi</u>

Next we apply the definition to the two cases:

≈ **<u>if</u> n=0 <u>then</u> x:=1**
<u>else</u> x:=(n−1)! ; x:=x.n <u>fi</u>

Massage the general case to get it in terms of the specification (here by splitting the assignments):

≈ **<u>if</u> n=0 <u>then</u> x:=1**
<u>else</u> n:=n−1; x:=n! ; n:=n+1; x:=x.n <u>fi</u>

≈ **<u>if</u> n=0 <u>then</u> x:=1**
<u>else</u> n:=n−1; S; n:=n+1; x:=x.n <u>fi</u>

Now the term **n** is reduced before the inner occurrence of **S** so we can apply our theorem on the recursive implementation of a specification to give the recursive procedure:

S ≈ **F <u>where</u>**
<u>proc</u> F ≡ **<u>if</u> n=0 <u>then</u> x:=1**
<u>else</u> n:=n−1; F; n:=n+1; x:=x.n <u>fi</u>.

Next we apply our general theorem on recursion removal. To distinguish the outer and inner calls of **F** we use an auxiliary variable $\mathbf{n_0}$ which stores the initial value of **n**. Then $\mathbf{n=n_0}$ only for the outermost call of **F**. The theorem gives:

S ≈
P ≡ $\mathbf{n_0}$**:=n; F.**
F ≡ **<u>if</u> n=0 <u>then</u> x:=1; /F**
<u>else</u> n:=n−1; F <u>fi</u>.
G ≡ **n:=n+1; x:=x.n; /F**
/F ≡ **<u>if</u> n=0 <u>then</u> Z**
<u>else</u> G <u>fi</u>.

Remove the tail recursion in **F**:

F ≡ **<u>while</u> n≠0 <u>do</u> n:=n−1 <u>od</u>; x:=1; /F.**

For integer **n⩾0** initially this is equivalent to:

F ≡ **n:=0; x:=1; /F.**

Copy **G** into **/F** and remove the recursion:

$\mathbf{/F} \equiv$ __while__ $\mathbf{n \neq n_0}$ __do__ $\mathbf{n{:=}n{+}1;\ x{:=}x.n}$ __od__; **Z.**

Copy all into **P** and get:

$\mathbf{P} \equiv$ $\mathbf{n{:=}n_0;\ n{:=}0;\ x{:=}1;}$
__while__ $\mathbf{n \neq n_0}$ __do__ $\mathbf{n{:=}n{+}1;\ x{:=}x.n}$ __od__.

which for integer **n** can be replaced by a __for__ loop: (replacing **n** by **i** and $\mathbf{n_0}$ by **n**):

$\mathbf{P} \equiv$ **x:=1;**
__for__ **i:=0** __to__ $\mathbf{n-1}$ __step__ **1** __do__
x:=x.(i+1) __od__

Which is equivalent to: (representing **i+1** by **i**):

$\mathbf{P} \equiv$ **x:=1;**
__for__ **i:=1** __to__ **n** __step__ **1** __do__
x:=x.i __od__

A typical iterative factorial calculation.

__Largest__ __true__ __Square__

Given a two dimensional boolean array **b[1..m, 1..n]**, where $\mathbf{m,n \geqslant 0}$ find a largest **true** square, ie the largest $\mathbf{size \geqslant 0}$ such that:

$\exists \mathbf{m_1, n_1}.\ \mathbf{0 < m_1 \wedge m_1 + size - 1 \leqslant m \wedge 0 < n_1 \wedge n_1 + size - 1 \leqslant n}$
$\wedge\ \forall \mathbf{i,j}.\ (\mathbf{m_1 \leqslant i < m_1 + size \wedge n_1 \leqslant j < n_1 + size}) \Rightarrow \mathbf{b[i,j] = true}$.

Thus if **m=0** or **n=0** or **b** is all **false** then **size** will be **0**, if **n>0** and **m>0** and **b** contains a single **true** element then **size** will be **1** etc.

Testing that a given square is **true** involves testing up to $\mathbf{size}^2$ elements of the boolean array, so a simple search for a **true** square of size **size>0** will involve up to $\mathbf{(m-size).(n-size).\ size^2}$ tests. If **m=n** and the whole array is **true** then a simple search will require:

$$\sum_{1 \leqslant size \leqslant n} \mathbf{(n-size).(n-size).\ size^2}$$

tests, which is of order $\mathbf{n^4}$.

To find a more efficient method we make use of the technique of “finite differencing” as follows:
(1) Find some properties of the data structure which, if known, would make the calculation of the required property fairly simple.
(2) Add other properties (if necessary) so that the properties (1) and (2) for a data structure can be easily determined given their values for all smaller data structures.

(3) Find a way of scanning through the data so that the number of sets of values of the properties is as small as possible. (It may help (1) and (2) if we try various scanning methods first and then work out the required properties for each method).

Suppose we are scanning through the array from left to right and top to bottom so that the current position is $\langle \mathbf{i,j} \rangle$ and the area already scanned is

$$\{\langle \mathbf{i_1,j_1} \rangle | \mathbf{1 \leqslant j} \wedge \mathbf{1 \leqslant i} \wedge (\mathbf{j_1 < j} \vee (\mathbf{j_1 = j} \wedge \mathbf{i_1 \leqslant i}))\}$$

Suppose know the position and size of largest true square within this area but know nothing about any true squares which extend beyond this area. We will suppose **LTS(i,j)** returns the size and bottom right-hand corner of the largest true square in the area up to and including $\langle \mathbf{i,j} \rangle$. This scanning method leads to the following program:
(where **P** $\iff$ $\langle$**size,row,col**$\rangle$ =**LTS(i,j)**):

```
i:=1; j:=0; size:=0; row:=0; col:=0; {P};
while i⩽m do
  while j<n do
    {P};
    j:=j+1;
    ⟨size,row,col⟩:=LTS(i,j) od;
  {P};
  i:=i+1; j:=0;
  ⟨size,row,col⟩:=LTS(i,j) od;
{i=m+1 ∧ j=n ∧ P}.
```

Incrementing **i** and setting **j** to zero does not add any squares to the scanned area so the occurrence of **LTS** outside the inner loop is redundant. We have:

```
i:=1; j:=0; size:=0; row:=0; col:=0; {P};
while i⩽m do
  while j<n do
    j:=j+1;
    ⟨size,row,col⟩:=LTS(i,j) od;
  i:=i+1; j:=0 od;
{i=m+1 ∧ j=n ∧ P}.
```

If we add the square $\langle \mathbf{i,j+1} \rangle$ to the area scanned then we need to change **LTS** only if the added square is the bottom right-hand corner of a true square larger than **LTS** (in fact it must be exactly one unit larger than **LTS**).

To test for this we need to test if all the elements of the square [**i−size..i, j−size−1..j+1**] are true (where **size** is the size of **LTS**) -provided these squares are all in the array. This will be so only when $\langle \mathbf{i,j+1} \rangle$ forms the bottom right-hand corner of a true rectangle at least **size+1** high and **size** wide and if $\langle \mathbf{i,j+1} \rangle$ is at the bottom of a column of true elements which is at least **size+1** high.

This last point leads to our considering the size of the "true column" above each element in the bottom row of the scanned area. This quantity should be easy to maintain as the area is increased: either we add a new true element to the column or we start a new column of height zero. So suppose we have a function **col(p,j)** which returns the size of the true column above $\langle \mathbf{p,j} \rangle$. We can use this to implement the function **LTS** under the condition **P**:

```
i:=1; j:=0; size:=0; row:=0; col:=0; {P};
while i≤m do
  while j<n do
    j:=j+1;
    p:=i;
    while p≤i−size ∧ p≥1 ∧ col(p,j)>size do
      p:=p+1 od;
    if p=i−size+1 then ⟨size,row,col⟩:=⟨size+1,i,j⟩ fi od;
  i:=i+1; j:=0 od;
{i=m+1 ∧ j=n ∧ P}.
```

In the case of an $\mathbf{n} \times \mathbf{n}$ array which is all true this will require order $\mathbf{n}^3$ iterations. Note that the columns to the left of the current position may be tested several times over. Trying to find further properties of the scanned area which would eliminate this leads to two suggestions:

(1) Maintain the width of the tallest true rectangle of width **size** which has $\langle \mathbf{i,j} \rangle$ at the bottom right-hand corner.

(2) Maintain the height of the widest true rectangle of height **size** which has $\langle \mathbf{i,j} \rangle$ as the bottom right-hand corner.

Either of these will allow us to replace the innermost loop in the last version by a simple test and thus lead to a more efficient program provided the property can be maintained efficiently.

Case (A): For the tallest **size**−wide rectangle: taking a column from the left and adding a column to the right leads to the following cases (where **H** is the current height, **left** is the height of the leftmost column and **right** is the height of the rightmost column):

(1) If **left**>**H** then adding a column **right**≥**H** cannot change **H** (since all the columns are still of height≥**H** and there is still some column in the rectangle with height **H**).

(2) If **left**>**H** then adding a column **right**<**H** means that the new **H** is **right**.

(3) If **left=H** then adding **right>H** could increase **H** iff no other column has height **H** (perhaps we could maintain the number of columns in the rectangle with height **H**?).
(4) If **left=H** then adding **right=H** doesn't change **H**.
(5) If **left=H** then adding **right<H** means that the new **H** is **right**.

The problem here is that each time **H** is increased (in case (3) we have to do a scan to calculate the new **H** (the worst case is still of order $\mathbf{n}^3$).

Case (B): For the widest **size**-high rectangle we have only two cases (again **right** is the height of the column added to the right of the current rectangle and **W** is the current width):
(1) If **right<size** then there cannot be a **size**-high rectangle which includes this column so the new **W** is **0**.
(2) If **right⩾size** then we can add the column to the rectangle which will still be **size**-high so the new **W** is **W+1**.

If we take **TSW(size,i,j)** to be the height of the tallest **size**-wide rectangle with **⟨i,j⟩** as the bottom right-hand corner and **WSH(size,i,j)** as the widest **size**-high rectangle then we have the two versions:

```
size:=0; row:=0; col:=0; i:=1; j:=0; H:=0; {H=TSW(i,j) ∧ P};
while i⩽m do
  while j<n do
    {H=TSW(i,j) ∧ P};
    j:=j+1;
    if H⩾size+1 ∧ col(i,j)⩾size+1
     then size:=size+1; row:=i; col:=j; H:=TSW(i,j)
     else H:=TSW(i,j) fi od;
  i:=i+1; j:=0 od;
{i=m+1 ∧ j=n ∧ P}.
```

and:

```
size:=0; row:=0; col:=0; i:=1; j:=0; W:=0; {W=WSH(i,j) ∧ P};
while i⩽m do
  while j<n do
    {W=WSH(i,j) ∧ P};
    j:=j+1;
    if W⩾size ∧ col(i,j)⩾size+1
     then size:=size+1; row:=i; col:=j; W:=WSH(i,j)
     else W:=WSH(i,j) fi od;
  i:=i+1; j:=0 od;
{i=m+1 ∧ j=n ∧ P}.
```

In the second version the assignment **W:=WSH(i,j)** can be carried out in constant time except for the case when **size** is increased when it requires order **n** time. However **size** can be increased no more than **n** times (for the worst case) in the whole program so the $\mathbf{n}^2$ assignments **W:=WSH(i,j)** only require order $\mathbf{n}^2$ time. This is obviously better than the first version which required order $\mathbf{n}^3$ time in the worst case.

We split the assignments **W:=WSH(i,j)** into the two cases described above:

```
size:=0; row:=0; col:=0; i:=1; j:=0; W:=0; {W=WSH(i,j) ∧ P};
while i⩽m do
  while j<n do
    {W=WSH(i,j) ∧ P};
    j:=j+1;
    if W⩾size ∧ col(i,j)⩾size+1
     then size:=size+1; row:=i; col:=j;
       if col(i,j)<size
        then W:=0
        else W:=0; while col(i,j−W)⩾size ∧ j−W>0 do W:=W+1 od fi;
       {W=WSH(i,j) ∧ P}
     else if col(i,j)<size
        then W:=0
        else W:=W+1 fi;
       {W=WSH(i,j) ∧ P} fi od;
  i:=i+1; j:=0; W:=0 od;
{i=m+1 ∧ j=n ∧ P}.
```

If we now maintain the size of all the true columns in the scanned area in an integer array **cols [1..n]** then the program becomes:

```
size:=0; row:=0; col:=0; i:=1; j:=0; W:=0;
for j:=1 to n step 1 do cols[j]:=0 od;
while i⩽m do
  while j<n do
    j:=j+1;
    if b[i,j]=true then cols[j]:=cols[j]+1
         else cols[j]:=0 fi;
    {cols[j]=col(i,j)};
    if W⩾size ∧ cols[j]⩾size+1
     then size:=size+1; row:=i; col:=j; W:=0;
       if cols[j]⩾size
        then while cols[j−W]⩾size ∧ j−W>0 do W:=W+1 od fi
     else if cols[j]<size
        then W:=0
        else W:=W+1 fi fi od;
  i:=i+1; j:=0; W:=0 od;
{i=m+1 ∧ j=n ∧ P}.
```

This gives the result in linear time (that is order $\mathbf{n \times m}$ time) and requires order $\mathbf{n+m}$ storage.

Data Types

Until now we have not made use of the concept of the types of the variables in any of our transformations, all our variables have been simple variables of unspecified type, although we have used them with stack and array operations. Many transformations are correct whatever type the variables have. We have merely stated that variables take their values from a set **D** of values which has not been further specified or given any structure. Practical programs have variables of different types which have different properties (eg. arrays, stacks, trees, sequential files etc). In this section we introduce a notation for data types, which is based on the concept of "prototypes" which are "cloned" and modified to create new variables of various types.

__Prototypes__

We assume that there exists a set of primitive __variables__ (the prototypes) called "**integer**", "**real**", "**boolean**", etc. which can be used in their own right as global variables and can also be used to define new variables of the same type by using the __**like**__ construct. Thus statements like:

integer:=integer+1
real:=real/integer
__**if**__ ¬**boolean** ∧ **integer=3** __**then**__ . . . __**fi**__

are legal statements involving these variables.

The statements:

__**var**__ **a,b** __**like**__ **integer** __**var**__ **x,y,z** __**like**__ **real** etc.

define new variables of the same type. Once a variable has been defined it can itself be used as a prototype to define other variables of the same type by using the __**like**__ construction. So:

__**var**__ **a,b** __**like**__ **integer;**
__**var**__ **c,d** __**like**__ **a**

defines four integer variables **a**,**b**,**c**, and **d**.

This removes the distinction between variable names and type names which is the cause of much confusion in strongly-typed languages such as Pascal. __All__ names are variable names and if a local variable is needed of the same type as some other variable then it can be declared without having to remember or look up the type of the variable. This reduces the number of names that someone has to keep in mind in order to understand the program which therefore reduces "memory load": which is now recognised as very important for readability. Amit [Amit 84] defines the memory load at a point in the program to be a function of the number of items the programmer needs to remember in order to understand the program, together with the distance between the current point and the last definition or use of each item.

So far as our formal system is concerned, a declaration of the type of a new variable is a "promise" that all the values assigned to that variable will satisfy a certain condition. This condition will therefore be a "global invariant" which is satisfied throughout execution of the program. For example the variable **real** satisfies the condition **real**$\in$**R** where **R** is the set of real numbers, **boolean** satisfies the condition **boolean**$\in$ {**true,false**} and so on. If **x** is a typed variable then we write $\mathbf{I}_x$ for the condition that **x** must satisfy. $\mathbf{I}_x$ is called the "global condition" for **x**. For any assignment to **x** we must have:

$$\{\mathbf{I}_x\};\ \mathbf{x{:=}x'.Q}\ \approx\ \{\mathbf{I}_x\};\ \mathbf{x{:=}x'.Q};\ \{\mathbf{I}_x\}$$

If **x** is not assigned a value when it is declared then **x** must be assigned before it is accessed and the first assignment to **x** may not assume $\mathbf{I}_x$ holds initially but must ensure that it holds after the assignment. We may assume that $\mathbf{I}_x$ is included as part of the condition in any assignment to **x** and therefore we interpret: $\mathbf{x{:=}x'.Q}$ to mean $\mathbf{x{:=}x'.}(\mathbf{I}_x \wedge \mathbf{Q})$ if necessary. For example if **x** and **y** are **like integer** then the assignment: $\mathbf{x{:=}x'.}(\mathbf{x'} > \mathbf{y})$ will only assign integer values to **x**.

Space precludes a full discussion of these ideas, see [Ward 89b] for the details. We will however discuss how recursive type definitions can be incorporated in this system since they make use of infinitary logic:

Recursive Data Structures

For the global condition of a recursively defined data structure we need an infinitely long formula. If variable **r** is recursively defined then we define:

$\mathbf{I}_0 =$ "**r** is one of the base cases"

and for $\mathbf{n{>}0}$: $\mathbf{I}_n =$ "**r** is constructed from components which all satisfy $\bigvee_{i<n}\mathbf{I}_i$"

Then the global condition for **r** is:

$$\mathbf{I} = \bigvee_{n<\omega}\mathbf{I}_n$$

For example, if we have:

var list either atom like string
or cons like ⟨list,list⟩

then $\mathbf{I}_0 = \exists \mathbf{x}.(\mathbf{list}{=}\langle \mathbf{1},\mathbf{x}\rangle \wedge \mathbf{I}_{string}[\mathbf{x}/\mathbf{string}])$

$\mathbf{I}_n = \exists \mathbf{x},\mathbf{y}.(\mathbf{list}{=}\langle \mathbf{2},\langle \mathbf{x},\mathbf{y}\rangle\rangle \wedge \bigvee_{i<n}\mathbf{I}_i[\mathbf{x}/\mathbf{list}] \wedge \bigvee_{i<n}\mathbf{I}_i[\mathbf{y}/\mathbf{list}])$

and $\mathbf{I}_{list} = \bigvee_{n<\omega}\mathbf{I}_n$.