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 (i.e., 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:

$$\text{funct } F(x) \equiv \begin{cases} 
\text{if } B(x) \text{ then } \phi(F(M(x)), F(N(x))) \\
\text{else } H(x) \end{cases}.$$ 

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

The function:

$$\text{funct } fusc(n) \equiv \begin{cases} 
\text{if } n=1 \to 1 \\
\text{if } n>1 \wedge \text{even}(n) \to fusc(n/2) \\
\text{if } n>1 \wedge \text{odd}(n) \to fusc((n-1)/2)+fusc(n+1)/2) \end{cases}.$$ 

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:

$$\text{funct } Gfusc(n) \equiv \langle fusc(n), fusc(n-1) \rangle.$$ 

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 $fusc(i-1)$:

$$\text{funct } F(m,a,b) \equiv a.fusc(m)+b.fusc(m-1).$$ 

Thus $fusc(m)=F(m,1,0)$ (for $m \geq 2$). 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:

$$\text{funct } F(m,a,b) \equiv \begin{cases} 
\text{if } m=2 \\
\text{then } a.fusc(2/2)+b.fusc(1) \\
\text{else if } \text{even}(m) \to a.fusc(m)+b.fusc(m-1) \\
\text{else } a.fusc(m)+b.fusc(m-1) \end{cases}.$$ 

Unfolding the calls of $fusc$ (using the fact that $\text{even}(m) \Rightarrow \text{odd}(m+1)$ 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 \( F(m,a,b) \) ≡
\[
\text{while } m \neq 2 \text{ do}
  \text{if even(m) then } m:=m/2; a:=a+b
  \text{else } m:=(m+1)/2; b:=a+b \text{ od;}
  a+b
\]

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

funct \( fusc(m) \) ≡
  if \( m=1 \) then \( 1 \)
  else \( a:=1; b:=0; \)
  \[\text{while } m \neq 2 \text{ do}
    \text{if even(m) then } m:=m/2; a:=a+b
    \text{else } m:=(m+1)/2; b:=a+b \text{ od;}
    a+b
  \] fi

Theorem: A generalisation of \( fib \) is:

funct \( H(m,r,s) \) ≡
  if \( B(m) \)
    then if \( B(\delta(m)) \) then \( \phi(H(\delta(m),r,s), H(\delta^2(m),r,s)) \)
      else \( r \) fi
    else \( s \) fi.
If we set $\delta(m) = m - 1$, $B(m) \iff m > 1$, $\phi(x, y) = x + y$ then $H(m, 0, 1) = \text{fib}(m)$.

We will prove that this is equivalent to:

\[
\text{func} \ G(m, r, s) \equiv \\
\text{if } B(m) \text{ then } m := \delta(m); \\
\text{while } B(m) \text{ do } m := \delta(m); \langle r, s \rangle := \langle \phi(r, s), r \rangle \text{ od; } \\
\text{else } s \text{ fi.}
\]

Provided $\delta$, $\phi$ and $B$ are determinate and have no side effects and $\phi$ is independent of $H$ and $\delta$. ($\phi$ and $B$ need not be defined everywhere though).

If any of $\delta$, $\phi$ 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 $\delta$, $\phi$ 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:

\[
a := H(\delta(m), r, s); b := H(m, r, s); c := H(\delta(m), r, s) \\
\approx a := H(\delta(m), r, s); c := H(\delta(m), r, s); b := H(m, r, s) \\
\approx a := H(\delta(m), r, s); c := a; b := H(m, r, s) \\
\approx a := H(\delta(m), r, s); b := H(m, r, s); c := a
\]

**Proof:** Note that $H(m, r, s)$ is defined (in the general case) in terms of $H(\delta(m), r, s)$ and $H(\delta^2(m), r, s)$ (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 $\text{fib}$ suggests defining an auxiliary function which returns both $H(m, r, s)$ and $H(\delta(m), r, s)$. What we actually do is define a procedure which sets the variables $\langle r, s \rangle$ to $\langle H(m, r, s), H(\delta(m), r, s) \rangle$ except when $\neg B(m)$ holds when we only set $r$ (since in this case $H(\delta(m), r, s)$ may not even be defined). We define:

\[
\text{proc } F(m) \equiv \\
\quad \text{if } B(m) \text{ then if } B(\delta(m)) \text{ then } \langle r, s \rangle := \langle H(m, r, s), H(\delta(m), r, s) \rangle \text{ fi } \\
\quad \text{else } r := s \text{ fi.}
\]

Then $H(m, r, s) \approx \text{func } H_1(m, r, s) \equiv \left[ F(m); r \right]$. 

**Claim:** $F(m) \approx F_1(m)$ where:
funct \( F_1(m) \) ≡
if \( B(m) \) then
  var \( i := 0; \)
  while \( B(\delta(m)) \) do
    \( m := \delta(m); i := i + 1 \) od;
  for \( j := 1 \) to \( i \) step 1 do
    \( (r,s) := \langle \phi(r,s), r \rangle \) od
  else \( r := s \) fi.

Proof of Claim:
We use the induction rule for recursion to prove \( F^n(m) \leq F_1(m) \) \( \forall n < \omega \).
This uses the fact that:
\[
(r,s) := (H^{n+1}(m,r,s), H^{n+1}(\delta(m),r,s))
\leq \kappa\kappa (r,s) := (\phi(H^n(\delta(m),r,s), H^n(\delta^2(m),r,s)), H^{n+1}(\delta(m),r,s))
\leq \kappa\kappa (r,s) := (H^{n+1}(\delta(m),r,s), H^n(\delta^2(m),r,s)); (r,s) := (\phi(r,s), r)
\]
where we have replaced two calls of \( H^{n+1}(\delta(m),r,s) \) by a single call.
Now apply the induction hypothesis to get:
\[
F^{n+1}(m) \leq \text{if } B(m)\text{ then } F_1(m)\text{ else } r := s \text{ fi}
\]
by loop rolling (twice).

Conversely:
if \( B(m) \leq B(m) \)
then \( i:=0; \text{DO}^n; \text{FOR} \)
else \( r:=s \)

\[
\begin{align*}
(r,s) &:= (H^n(m,r,s), H^n(\delta(m),r,s)) ; \\
\langle r,s \rangle &:= \langle \phi(r,s), r \rangle \\
\text{else} \; r &:= s \\
\end{align*}
\]
by induction hypothesis.

\[
\begin{align*}
\leq \; \text{if } B(m) \\
\text{then } \\
\hspace{1cm} \text{if } B(\delta(m)) \\
\hspace{1cm} m &:= \delta(m); \\
\langle r,s \rangle &:= (H^n(m,r,s), H^n(\delta(m),r,s)), H^{n+1}(\delta(m),r,s)) \\
\text{else} \; r &:= s \\
\end{align*}
\]
where we have refined a single call \( H^n(\delta(m),r,s) \) to two calls \( H^n(\delta(m),r,s) \) and \( H^{n+1}(\delta(m),r,s) \).

\[ \approx F^{n+1}_1(m) \] by folding.
which proves the claim.

Now since \( \phi \) is independent of \( B \) and \( \delta \) we may apply loop overlapping to \( F_1 \) to get:

\[ F_1(m) \approx G'(m) \] where

\[
\begin{align*}
\text{proc } & G'(m) \equiv \\
\text{if } B(m) \text{ then } & \text{while } B(\delta(m)) \text{ do} \\
& m := \delta(m); \langle r,s \rangle := \langle \phi(r,s), r \rangle \text{ od} \\
\text{else} \; & r := s \\
\end{align*}
\]
which may be written (using proper inversion):

\[
\begin{align*}
\text{proc } & G'(m) \equiv \\
\text{if } B(m) \text{ then } & m := \delta(m); \\
\text{while } & B(m) \text{ do} \langle r,s \rangle := \langle \phi(r,s), r \rangle; \; m := \delta(m) \text{ od} \\
\text{else} \; & r := s \\
\end{align*}
\]
Hence \( H(m,r,s) \approx \)

\[
\begin{align*}
\text{func } & H_2(m,r,s) \equiv \\
\int & G'(m); r \\
\end{align*}
\]
which leads to our=function \( G \) above.
(The subsidiary function \( G^\ast \) implements the \texttt{while} loop in the procedure \( G' \)).

If we add the extra condition that \( \phi \) and \( \delta \) are defined everywhere then we can add extra statements to \( G' \) to give:
\[\text{proc } G'(m) \equiv\]
\[\text{if } B(m) \text{ then } m := \delta(m);\]
\[\text{while } B(m) \text{ do}\]
\[\langle r, s \rangle := \langle \phi(r, s), r \rangle; \quad m := \delta(m) \text{ od;}\]
\[\langle r, s \rangle := \langle \phi(r, s), r \rangle;\]
\[r := s\]
\[\text{else } r := s \text{ fi.}\]

(since we don’t care what the final value of \(s\) is).

\[\approx \text{if } B(m) \text{ then } m := \delta(m); \quad \langle r, s \rangle := \langle \phi(r, s), r \rangle;\]
\[\text{while } B(m) \text{ do}\]
\[\langle r, s \rangle := \langle \phi(r, s), r \rangle; \quad m := \delta(m) \text{ od; } r := s.\]

since \(\phi\) is independent of \(B\) and \(\delta\).

\[\approx \text{while } B(m) \text{ do } \langle r, s \rangle := \langle \phi(r, s), r \rangle; \quad m := \delta(m) \text{ od; } r := s.\]

by loop rolling.

Thus we get the more compact (though less efficient) version of \(G\):
\[\text{funct } G_1(m, r, s) \approx\]
\[\text{if } B(m) \text{ then } G_1(\delta(m), \phi(r, s), r)\]
\[\text{else } s \text{ fi.}\]

\textbf{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:
\[\text{funct } F(n, x) \equiv\]
\[\text{if } n > 0 \text{ then } F(n-1, \phi(n, F(n-1, x)))\]
\[\text{else } x \text{ fi.}\]

This can be transformed to:
funct $F(n,x) \equiv$
\[
\sum_{c:=2^n-1 \text{ step } -1 \text{ to } 1} \phi(ntz(c)+1,x) \text{ od;}
\]
where
funct $ntz(c) \equiv$ “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:

```plaintext
proc F(n,x) \equiv
  if n>0 then F(n-1,x); x:=\phi(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:

```plaintext
proc F(n,x) \equiv F; Z. where
F \equiv if n>0 then n:=n-1; F; x:=\phi(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 $F(n,x)$:

```plaintext
proc F(n,x) \equiv F; r:=x; Z. where
F \equiv if n>0 then n:=n-1; F; x:=\phi(n+1,x); F; n:=n+1 fi.
```

The theorem on recursion removal gives the iterative version:

```plaintext
proc F(n,x) \equiv
  Stack:=\langle\rangle;
  while n>0 do n:=n-1; Stack:=1 od;
  while Stack\neq\langle\rangle do
d:=Stack;
    if d=1 then x:=\phi(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:

- $Stack:=\langle\rangle$ becomes $c:=1$
- $Stack=\langle\rangle$ becomes $c=1$
- $Stack\leftarrow d$ becomes $c:=2.c+d$
- $d:=Stack$ becomes $(c,d):=(c\div2)$
The statement \texttt{while }n>0 \texttt{ do } n:=n-1; \texttt{ Stack}←1 \texttt{ od} becomes \(c:=(c+1)2^n-1; n:=0\) 
\(i.e. \ c:=(c+1)2^n-1; n:=0\)

We get: \texttt{proc }F(n,x) \equiv 
\begin{align*}
c & := 2^{n+1}-1; n:=0; \\
\text{while } c\neq 1 \texttt{ do} \\
& (c,d):=(c\div 2); \\
& \text{if } d=1 \texttt{ then } x:=\phi(n+1,x); c:=2.c; \\
& \quad c:=(c+1)2^n-1; n:=0 \\
& \quad \text{else } n:=n+1 \texttt{ fi}; r:=x. \\
\end{align*}

Apply entire loop unroll to the \texttt{else} part:
\texttt{proc }F(n,x) \equiv 
\begin{align*}
c & := 2^{n+1}-1; n:=0; \\
\text{while } c\neq 1 \texttt{ do} \\
& (c,d):=(c\div 2); \\
& \text{if } d=1 \texttt{ then } x:=\phi(n+1,x); c:=2.c; \\
& \quad c:=(c+1)2^n-1; n:=0 \\
& \quad \text{else } n:=n+1; \\
& \quad \text{while } c\neq 1 \wedge \text{ even }(c) \texttt{ do} \\
& \quad \quad c:=c/2; n:=n+1 \texttt{ od}; r:=x. \\
\end{align*}

We now backward expand the \texttt{if}, changing the test from \(d=1\) to \texttt{ odd}(c). The statement 
\((c,d):=(c\div 2)\) 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.

\texttt{proc }F(n,x) \equiv 
\begin{align*}
c & := 2^{n+1}-1; n:=0; \\
\text{while } c\neq 1 \texttt{ do} \\
& \text{if } \texttt{ odd}(c) \texttt{ then } c:=(c-1)/2; x:=\phi(n+1,x); c:=2.c; \\
& \quad c:=(c+1)2^n-1; n:=0 \\
& \quad \text{else } c:=c/2; n:=n+1; \\
& \quad \text{while } c\neq 1 \wedge \text{ even }(c) \texttt{ do} \\
& \quad \quad c:=c/2; n:=n+1 \texttt{ od}; r:=x. \\
\end{align*}

When \(c\) is even the statements \(c:=c/2; n:=n+1; \texttt{ while } c\neq 1 \wedge \texttt{ even}(c) \texttt{ do} c:=c/2; n:=n+1 \texttt{ od}\) can be replaced by \(c:=c/2^{\text{ntz}(c)}; n:=n+\text{ntz}(c)\).

Also the assignments \(c:=(c-1)/2; c:=2.c; c:=(c+1)2^n-1\) may be merged to: \(c:=c.2^n-1\). We get:
\[
\text{proc } F(n,x) \equiv \\
\quad c:=2^{n+1}-1; \text{ } n:=0; \\
\quad \text{while } c\neq 1 \text{ do} \\
\quad \quad \text{if odd}(c) \text{ then } c:=c \cdot 2^{n_0-1}; x:=\phi(n+1,x); n:=0 \\
\quad \quad \quad \text{else } c:=c/2^{ntz(c)}; n:=n+ntz(c) \text{ fi enddo; } r:=x.
\]

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 \( 2^{n_0-1} \) 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:

\[
\text{proc } F(n,x) \equiv \\
\quad c:=2^{n+1}-1; \text{ } n:=0; \\
\quad \text{while } c\neq 1 \text{ do} \\
\quad \quad c:=c \cdot 2^{n_0-1}; x:=\phi(n+1,x); n:=0 \\
\quad \quad c:=c/2^{ntz(c)}; n:=n+ntz(c) \text{ enddo; } r:=x.
\]

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

\[
\text{proc } F(n,x) \equiv \\
\quad c:=2^{n+1}-1; \text{ } n:=0; \\
\quad \text{do} \text{ if } c=1 \text{ then exit fi; } \\
\quad \quad c:=c \cdot 2^{n_0-1}; x:=\phi(n+1,x); \\
\quad \quad n:=ntz(c); c:=c/2^{n_0} \text{ enddo; } r:=x.
\]

Since \( 2^{n+1}-1 \) is odd we can replace \( n:=0 \) by \( n:=ntz(c) \); \( c:=c/2^{n_0} \) and then apply proper inversion to the loop:

\[
\text{proc } F(n,x) \equiv \\
\quad c:=2^{n+1}-1; \\
\quad \text{do} \text{ if } c=2^{n_0} \text{ then exit fi; } \\
\quad \quad n:=ntz(c); c:=c/2^{n_0}; \\
\quad \quad \text{if } c=1 \text{ then exit fi; } \\
\quad \quad c:=c \cdot 2^{n_0-1}; x:=\phi(n+1,x) \text{ enddo; } r:=x.
\]

If \( n_0 \) is the initial value of \( n \) then we see that \( c\geq 2^{n_0} \wedge (c=2^{n_0} \Rightarrow c/2^{ntz(c)}=1) \) 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 \( c \)):

\[
\text{proc } F(n,x) \equiv \\
\quad c:=2^{n+1}-1; n_0:=n; \\
\quad \text{do} \text{ if } c=2^{n_0} \text{ then exit fi; } \\
\quad \quad n:=ntz(c); c:=c/2^{n_0}; \\
\quad \quad c:=c \cdot 2^{n_0-1}; x:=\phi(n+1,x) \text{ enddo; } r:=x.
\]
Now we can merge the assignments \( c := c/2^n \); \( c := c.2^n - 1 \) to \( c := c - 1 \) and replace \( c \) by \( c - 2^n \) since for \( c > 2^n \), \( \text{ntz}(c) = \text{ntz}(c - 2^n) \): (and \( c > 2^n \) is invariant over the loop so \( c > 2^n \) is true within the loop.

```plaintext
proc F(n,x) ≡
   c:=2^n+1-1; n_0:=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:

```plaintext
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:

```plaintext
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 \( \text{ntz}(2^n-c) = \text{ntz}(c) \) 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:

```plaintext
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” \( x_0 \) is known in advance (or can be calculated easily from \( x \)) then we can build up a table of the values of \( h \) from \( x_0 \) 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 tabh (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 tabh b ≡
  if b=a then ⟨h(a)⟩
  else h(b)::tabh (b-1) fi.
```

where $⟨x⟩$ is the sequence with one element $x$ and $x::list$ is list with $x$ added to the front ie $⟨x⟩&list$.

Applying function inversion to this gives:

```
funct tabh 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 tabh 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 $\forall i. (a≤i<y \Rightarrow z[i-a+1]=h(i))$ 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:
\[
\text{proc } \text{tab}_h(b) \equiv \\
\text{for } i := a \text{ to } b \text{ step } 1 \text{ do } \\
\{ \forall i. (a \leq j < i \Rightarrow T[j] = h(j)) \}; \\
T[i] := h(i) \text{ od.}
\]

We unfold the call \( h(i) \) using the assertion. Suppose \( h \) is:
\[
\text{funct } h(x) \equiv E_h.
\]

We get:
\[
\text{proc } \text{tab}_h(b) \equiv \\
\text{for } i := a \text{ to } b \text{ step } 1 \text{ do } \\
T[i] := E_h[T[e]/h(e)][i/x] \text{ od.}
\]

where \( T[e] \) replaces \( h(e) \) for every expression \( e \).

Now we can replace \( h(i) \) by:
\[
\text{funct } h(x) \equiv \\
\lceil \text{tab}_h(x); T[x] \rceil \text{ where } \\
\text{proc } \text{tab}_h(b) \equiv \\
\text{for } i := a \text{ to } b \text{ step } 1 \text{ do } \\
T[i] := E_h[T[e]/h(e)][i/x] \text{ od.}
\]

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 
\( \forall i. (a \leq i \leq l \Rightarrow T[i] = h(i)) \). We can then replace \( h \) by:
\[
\text{funct } h(x) \equiv \\
\lceil \text{if } l < x \\
\text{then } \text{for } i := l+1 \text{ to } x \text{ step } 1 \text{ do } \\
T[i] := E_h[T[e]/h(e)] \text{ od; } \\
l := x; \text{ od; } \\
T[x] \rceil
\]

Example:
\[
q_n \text{ denotes the number of ways a set of } n \text{ elements may be partitioned, where } q_0 = 1, \\
q_{n+1} = \sum_{0 \leq s \leq n} (n \choose s) q_{n-s}
\]

where \( (n \choose s) = \frac{n!}{s! \cdot (n-s)!} \)

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 \( q_n \) ways, choose 2 elements in \( (n+1)2 \) ways and form the other partitions in \( q_{n-1} \) ways,...

In general for \( 0 \leq s \leq n \) we form the first partition from \( s+1 \) elements in \( (n+1)s+1 \) ways and form the other partitions in \( q_{n+1-s-1} = q_{n-s} \) ways.

A simple recursive function to evaluate this is:

\[
\text{funct } \text{part}(n) \equiv \\
\text{if } n=0 \text{ then } 1 \\
\text{else } \text{var } \text{sum} := 0; \\
\text{for } s:=0 \text{ to } n \text{ step } 1 \text{ do} \\
\text{sum} := \text{sum} + \text{comb}(n,s).\text{part}(n-s) \text{ od}; \\
\text{sum} \text{ fl. where} \\
\text{funct } \text{comb}(n,s) \equiv \\
\text{fact}(n) / (\text{fact}(s).\text{fact}(n-s)), \\
\text{funct } \text{fact}(n) \equiv \\
\text{if } n=0 \text{ then } 1 \\
\text{else } n.\text{fact}(n-1) \text{ fi.}
\]

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

Note that for \( n>0 \) \text{part}(n) \text{ is defined in terms of all of } \text{part}(0) \text{ to } \text{part}(n-1) \text{ which suggests that it is a good candidate for range-of-values tabulation. If we initialise the program with } \\
l:=1; T[1]:=1 \text{ then the transformation gives:} \\
\text{funct } \text{part}(n) \equiv \\
\text{if } l<n \\
\text{then for } i:=l+1 \text{ to } n \text{ step } 1 \text{ do} \\
\text{sum} := 0; \\
\text{for } s:=0 \text{ to } i-1 \text{ step } 1 \text{ do} \\
\text{sum} := \text{sum} + \text{comb}(i-1,s).T[i-1-s] \text{ od}; \\
T[i]:=sum \text{ od fi; } \\
T[n] \text{ where} \\
\text{funct } \text{comb}(n,s) \equiv \\
\text{fact}(n) / (\text{fact}(s).\text{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 1<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] fi

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

funct part(n) ≡
  if 1<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] fi

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
  else if 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
  else if 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 ⟨m,Ack(m,n)⟩.

funct Ack(m,n) ≡ Ack*(m,n)[2]. where
funct Ack*(m,n) ≡
  if m = 0 then ⟨m,n+1⟩
  else if 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 m2 = m after the first inner call so we can replace the reference to m by m2 to give:
funct Ack(m,n) ≡
    Ack*(m,n)[2]. where

funct Ack*(m,n) ≡
    if m=0 then (m,n+1)
    elsif n=0 then
        var m1,n1; m1:=m-1; n1:=1; Ack*(m1,n1)
        n1:=n-1; (m2,n1):=Ack*(m,n1); m1:=m2-1; 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**(m,n)[2]. where

funct Ack**(m,n) ≡
    if m=0 then (m,n+1)
    elsif n=0 then
        m:=m-1; n:=1; Ack**(m,n)
    else
        n:=n-1;
        (m,n):=Ack**(m,n);
        m:=m-1; Ack**(m,n)
    fi.

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

funct Ack(m,n) ≡
    Ack**(m,n)

proc Ack**(m,n) ≡
    if m=0 then n:=n+1
    elsif n=0 then
        m:=m-1; n:=1; Ack**(m,n)
    else
        n:=n-1; Ack**(m,n);
        m:=m-1; Ack**(m,n)
    fi.

Cascade Recursion

Consider the cascade recursion (where k_1 and k_2 are different):

funct F(x) ≡
    if B(x) then 
        φ(F(k_1(x)), F(k_2(x)), E(x))
    else H(x)
    fi.

The detailed form is:
\[
\text{funct } F(x) \equiv \\
\text{if } B(x) \text{ then } \\
\begin{array}{l}
\text{var } x_1, x_2, z_1, z_2; \\
x_1 := k_1(x); z_1 := F(x_1); x_2 := k_2(x); z_2 := F(x_2); \\
\phi(z_1, z_2, E(x)) \\
\text{else } H(x) \text{ fi} \\
\end{array}
\]
x and \(z_1\) violate the condition. As above we return \(z_1\) as an additional result:

\[
\text{funct } F(x) \equiv \\
F^*(x)[2], \text{ where} \\
\text{funct } F^*(x) \equiv \\
\text{if } B(x) \text{ then } \\
\begin{array}{l}
\text{var } x_1, x_2, y_1, y_2, z_1, z_2; \\
x_1 := k_1(x); \langle y_1, z_1 \rangle := F^*(x_1); \\
x_2 := k_2(x); \langle y_2, z_2 \rangle := F^*(x_2); \\
\phi(z_1, z_2, E(x)) \\
\end{array}
\]
\text{else } \langle x, H(x) \rangle \text{ fi}
\]

Using the inverses \(k_1'\) and \(k_2'\) we replace \(x\) by \(k_1'(y_1)\) or \(k_2'(y_2)\) to disentangle \(x\):

\[
\text{funct } F(x) \equiv \\
F^*(x)[2], \text{ where} \\
\text{funct } F^*(x) \equiv \\
\text{if } B(x) \text{ then } \\
\begin{array}{l}
\text{var } x_1, x_2, y_1, y_2, z_1, z_2; \\
x_1 := k_1(x); \langle y_1, z_1 \rangle := F^*(x_1); \\
x_2 := k_2(x); \langle y_2, z_2 \rangle := F^*(x_2); \\
\langle x, \phi(z_1, z_2, E(k_2'(y_2))) \rangle \\
\end{array}
\]
\text{else } \langle x, H(x) \rangle \text{ fi}
\]

\(z_1\) 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 \(z_1\) and replacing \(x\) and the local variables by global variables and writing \(F^*\) as a procedure gives:

\[
\text{funct } F(x) \equiv \\
\text{Stack := } \langle \rangle ; F^{**}; x. \text{ where} \\
F^{**} \equiv \text{if } B(x) \text{ then } \\
\begin{array}{l}
\text{var } x; x := k_1(x); F^{**}; x := k_2(k_1'(x)); \text{Stack} \leftarrow z; F^{**}; z_1 \leftarrow \text{Stack}; x := k_1'(x); \\
z := \phi(z_1, z, E(x)) \\
\end{array}
\]
\text{else } z := H(x) \text{ fi}
\]

where \(x\) is used for \(x, y_1\) and \(y_2\) and \(z\) is used for \(z_1\) and \(z_2\).
If $k_1$ or $k_2$ do not have inverses then $x$ will need a stack as well: all the intermediate results can be put on the same stack if needed.

**Nested Recursion:**

Consider:

```plaintext
funct G(x) ≡
  if B(x) then φ( G(Ψ( G(k_1(x)), k_2(x) )), E(x))
  else H(x) fi.
```

In detailed form:

```plaintext
funct G(x) ≡
  if B(x) then
    var x_1,x_2,z_1,z_2;
    x_1:=k_1(x); z_1:=G(x_1);
    x_2:=Ψ(z_1,k_2(x)); z_2:=G(x_2);
    φ(z_2,E(x))
  else H(x) fi.
```

$z_1$ and $z_2$ are already disentangled, but in general we will not be able to reconstruct $x$ by function inversion so it required a stack:

```plaintext
funct G(x) ≡
  Stack:=⟨⟩; G; z.
  where
  G ≡ if B(x) then
    Stack←x; x:=k_1(x); G; x←Stack;
    Stack←x; x:=Ψ(z,k_2(x)); G; x←Stack; z:=φ(z,E(x))
  else z:=H(x) 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:

```plaintext
funct F(x) ≡
  if B(x) then
    F(k_1(x)) ρ F(k_2(x)) ρ E(x)
  else H(x) fi.
```
To use functional embedding we need a close relation between \( k_1 \) and \( k_2 \) (eg \( k_1(x) = k_2(k_1(x)) \)), to use range-of-values tabulation \( k_1 \) and \( k_2 \) also have to satisfy strict conditions. For many cases these conditions fail and we have to use a stack.

Since \( \rho \) is associative we can use re-bracketing on \( F \). Let \( F_1 \) be identical to \( F \), then:

\[
\text{funct } F(x) \equiv \\
\quad \text{if } B(x) \text{ then } F(k_1(x)) \rho F_1(k_2(x)) \rho E(x) \\
\quad \text{else } H(x) \end{array} \]

and re-bracketing gives (where \( e \) is an identity element for \( \rho \)):

\[
\text{funct } F(x) \equiv \\
\quad \text{funct } G(x,e). \text{ where } \\
\quad \text{funct } G(x,z) \equiv \\
\quad \quad \text{if } B(x) \text{ then } G(k_1(x), (F_1(k_2(x)) \rho E(x)) \rho z) \\
\quad \quad \text{else } H(x) \rho z \end{array} \]

The inner call of \( F \) can be replaced by \( G(x,e) \) since \( F_1 \) is still equivalent to \( F \):

\[
\text{funct } F(x) \equiv \\
\quad \text{funct } G(x,e). \text{ where } \\
\quad \text{funct } G(x,z) \equiv \\
\quad \quad \text{if } B(x) \text{ then } G(k_1(x), G(k_2(x), e) \rho E(x) \rho z)) \\
\quad \quad \text{else } H(x) \rho z \end{array} \]

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

\[
\text{funct } F(x) \equiv \\
\quad \text{funct } G(x,e). \text{ where } \\
\quad \text{funct } G(x,z) \equiv \\
\quad \quad \text{if } B(x) \text{ then } G(k_1(x), G(k_2(x), E(x)) \rho z)) \\
\quad \quad \text{else } H(x) \rho z \end{array} \]

which in detailed form is:
\[
\text{funct } F(x) \equiv \quad G(x,e). \quad \text{where}
\[
\text{funct } G(x,z) \equiv \quad \begin{cases} 
\text{if } B(x) \quad \text{then} \quad \begin{aligned}
&x_1 := k_2(x); 
z_1 := E(x) \rho z; 
&r_1 := G(x_1,z_1); 
x_2 := k_1(x); 
z_2 := r_1; 
&G(x_2,z_2) 
\end{aligned} \\
\text{else } H(x) \rho z 
\end{cases}
\]

Now only \( x \) violates the condition and, as we have assumed nothing about \( k_1 \) and \( k_2 \) we must use a stack for \( x \):

\[
\text{funct } F(x) \equiv \quad \begin{cases} 
\text{Stack} := (); \quad z := e; \quad \text{G; } z \quad \text{where} \quad 
G \equiv \begin{cases} 
\text{if } B(x) \quad \text{then} \quad \begin{aligned}
&\text{Stack} \leftarrow x; 
x := k_2(x); 
z := E(x) \rho z; 
&G; 
\end{aligned} \\
\text{else } z := H(x) \rho z 
\end{cases} 
\end{cases}
\]

If we had an inverse for \( k_2 \) we could use function inversion to remove this stack.

Recurrence Relations

The \( r \)-term recurrence:

\[
\text{funct } F(m) \equiv \quad \begin{cases} 
\text{if } m = m_0 \rightarrow H_0(m) \\
\quad \begin{cases} 
\text{m} = m_1 \rightarrow H_1(m) \\
\quad \begin{cases} 
\quad \begin{cases} 
\text{m} = m_{r-2} \rightarrow H_{r-2}(m) \\
\text{m} = m_{r-1} \rightarrow \phi(F(\text{pred}(m)), F(\text{pred}^2(m)), \ldots, F(\text{pred}^{r-1}(m)), m) 
\end{cases} 
\end{cases} 
\end{cases} 
\end{cases}
\]

\( (\text{where } m_r = \text{succ}^r(m_0) \text{ and } \text{pred}(\text{succ}(m)) = m) \), can be treated in a similar way to our derivation of a linear function for computing \( \text{fib} \), (which is in fact a \( 3 \)-term recurrence).

As with our first treatment of \( \text{fib} \) we proceed by introducing the function:

\[
\text{funct } G(m) \equiv \quad (F(\text{pred}^{r-2}(m)), F(\text{pred}^{r-3}(m)), \ldots, F(\text{pred}(m)), F(m)).
\]
which returns an \((\mathbf{r} - 1)\)-tuple of results. Unfold \(F(\mathbf{m})\) and then transform \(G(\mathbf{m})\) so that it is defined in terms of \(G(\mathbf{m} - 1)\) (taking out the cases \(\mathbf{m} = \mathbf{m}_0, \ldots, \mathbf{m} = \mathbf{m}_{r - 2}\)). The result can be transformed to give the following linear iterative form for \(F\):

\[
\text{funct } F(\mathbf{m}) \equiv \\
\text{if } \mathbf{m} = \mathbf{m}_0 \rightarrow H_0(\mathbf{m}) \\
\text{m} = \mathbf{m}_1 \rightarrow H_1(\mathbf{m}) \\
\text{...} \\
\text{m} = \mathbf{m}_{r - 2} \rightarrow H_{r - 2}(\mathbf{m}) \\
\text{m} \geq \mathbf{m}_{r - 1} \rightarrow \langle v_1, v_2, \ldots, v_{r - 1} \rangle := \langle H_0(\mathbf{m}_0), H_1(\mathbf{m}_1), \ldots, H_{r - 2}(\mathbf{m}_{r - 2}) \rangle; \\
y := \mathbf{m}_{r - 1}; \\
\text{while } y \neq \text{succ}(\mathbf{m}) \text{ do} \\
\langle v_1, v_2, \ldots, v_{r - 1} \rangle := \langle v_2, v_3, \ldots, v_{r - 1}, \phi(v_1, v_2, \ldots, v_{r - 1}, y) \rangle; \\
y := \text{succ}(y) \text{ od}; \\
\langle v_1, \ldots, v_{r - 1} \rangle.
\]

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 \(v_1, \ldots, v_{r - 1}\).

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 \(x\) to \(n!\) (\(n\) 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.
**Defn:**  
\[ n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise.} \end{cases} \]

Our specification is simply:  
\[ S \equiv x := n! \]

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

\[ \approx \text{if } n = 0 \text{ then } x := n! \text{ by splitting a tautology.} \]

\[ \text{else } x := n! \text{ fi} \]

Next we apply the definition to the two cases:

\[ \approx \text{if } n = 0 \text{ then } x := 1 \]

\[ \text{else } x := (n-1)! ; x := x \cdot n \text{ fi} \]

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

\[ \approx \text{if } n = 0 \text{ then } x := 1 \]

\[ \text{else } n := n-1 ; x := n! ; n := n+1 ; x := x \cdot n \text{ fi} \]

\[ \approx \text{if } n = 0 \text{ then } x := 1 \]

\[ \text{else } n := n-1 ; S ; n := n+1 ; x := x \cdot n \text{ fi} \]

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:

\[ S \approx F \text{ where} \]

\[ \text{proc } F \equiv \text{if } n = 0 \text{ then } x := 1 \]

\[ \text{else } n := n-1 ; F ; n := n+1 ; x := x \cdot n \text{ fi} \]

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

\[ S \approx P \]

\[ F \equiv \text{if } n = 0 \text{ then } x := 1 ; /F \]

\[ \text{else } n := n-1 ; F ; /F \]

\[ G \equiv n := n+1 ; x := x \cdot n ; /F \]

\[ /F \equiv \text{if } n = 0 \text{ then } Z \]

\[ \text{else } G \text{ fi.} \]

Remove the tail recursion in \( F \):

\[ F \equiv \text{while } n \neq 0 \text{ do } n := n-1 \text{ od; } x := 1 ; /F. \]

For integer \( n \geq 0 \) initially this is equivalent to:

\[ F \equiv n := 0 ; x := 1 ; /F. \]
Copy G into /F and remove the recursion:
/F ≡ while n≠n₀ do n:=n+1; x:=x.n od; Z.
Copy all into P and get:
P ≡ n:=n₀; n:=0; x:=1;
while n≠n₀ do n:=n+1; x:=x.n od.
which for integer n can be replaced by a for loop: (replacing n by i and n₀ by n):
P ≡ x:=1;
for i:=0 to n−1 step 1 do x:=x.(i+1) od
Which is equivalent to: (representing i+1 by i):
P ≡ 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 m,n≥0 find a largest true square, ie the largest size≥0 such that:
∃m₁,n₁. 0<m₁ ∧ m₁+size−1≤m ∧ 0<n₁ ∧ n₁+size−1≤n
∧ ∀i,j. (m₁ ≤i<m₁+size ∧ n₁ ≤j<n₁+size) ⇒ 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 size² elements of the boolean array, so a simple search for a true square of size size>0 will involve up to (m−size).(n−size). size² tests. If m=n and the whole array is true then a simple search will require:
∑₁≤size≤n(n−size).(n−size). size² tests, which is of order n⁴.

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 \((i,j)\) and the area already scanned is
\[
\{(i_1,j_1) | 1 \leq i_1 \leq i \land (j_1 < j \lor (j_1 = j \land i_1 \leq i))\}
\]

Suppose we know the position and size of the 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 \((i,j)\). This scanning method leads to the following program:

\[
\text{(where } P \iff (\text{size, row, col}) = LTS(i,j)):\n\]
\[
i:=1; j:=0; \text{size}:=0; \text{row}:=0; \text{col}:=0; \{P\};
\]
\[
\text{while } i \leq m \text{ do}
\]
\[
\text{while } j < n \text{ do}
\]
\[
\{P\};
\]
\[
j:=j+1;
\]
\[
\langle \text{size, row, col} \rangle := LTS(i,j) \text{ od};
\]
\[
\{P\};
\]
\[
i:=i+1; j:=0;
\]
\[
\langle \text{size, row, col} \rangle := LTS(i,j) \text{ od};
\]
\[
\{i=m+1 \land j=n \land P\}.
\]

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:

\[
i:=1; j:=0; \text{size}:=0; \text{row}:=0; \text{col}:=0; \{P\};
\]
\[
\text{while } i \leq m \text{ do}
\]
\[
\text{while } j < n \text{ do}
\]
\[
\{P\};
\]
\[
j:=j+1;
\]
\[
\langle \text{size, row, col} \rangle := LTS(i,j) \text{ od};
\]
\[
i:=i+1; j:=0 \text{ od};
\]
\[
\{i=m+1 \land j=n \land P\}.
\]

If we add the square \((i,j+1)\) 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 \([i-\text{size..}i, j-\text{size..}1..j+1]\) are true (where \(\text{size}\) is the size of \(LTS\)) -provided these squares are all in the array. This will
be so only when \((i, j+1)\) forms the bottom right-hand corner of a true rectangle at least \textbf{size}+1 high and \textbf{size} wide and if \((i, j+1)\) is at the bottom of a column of true elements which is at least \textbf{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 \textbf{col}(p, j) which returns the size of the true column above \((p, j)\). We can use this to implement the function \textbf{LTS} under the condition \(P\):

\[
i:=1; j:=0; \text{size}:=0; \text{row}:=0; \text{col}:=0; \{P\};
\]

\textbf{while} \(i \leq m\) \textbf{do}

\textbf{while} \(j < n\) \textbf{do}

\(j:=j+1;\)

\(p:=i;\)

\textbf{while} \(p \leq i-\text{size} \land p \geq 1 \land \text{col}(p, j) > \text{size} \) \textbf{do}

\(p:=p+1\) \textbf{od};

\textbf{if} \(p=i-\text{size}+1\) \textbf{then} \(\text{size}, \text{row}, \text{col}:=(\text{size}+1, i, j)\) \textbf{fi};

\(i:=i+1; j:=0\) \textbf{od};

\(i=m+1 \land j=n \land P\) \textbf{od};

In the case of an \(n \times n\) array which is all true this will require order \(n^3\) 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 \textbf{size} which has \((i, j)\) at the bottom right-hand corner.
2. Maintain the height of the widest true rectangle of height \textbf{size} which has \((i, j)\) 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.

\textbf{Case (A)}: For the tallest \textbf{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, \textbf{left} is the height of the leftmost column and \textbf{right} is the height of the rightmost column):

1. If \textbf{left} > \(H\) then adding a column \textbf{right} \(\geq H\) cannot change \(H\) (since all the columns are still of height \(\geq H\) and there is still some column in the rectangle with height \(H\)).
2. If \textbf{left} > \(H\) then adding a column \textbf{right} < \(H\) means that the new \(H\) is \textbf{right}.  

25
(3) If \( \text{left} = H \) then adding \( \text{right} > H \) 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 \( \text{left} = H \) then adding \( \text{right} = H \) doesn’t change \( H \).

(5) If \( \text{left} = H \) then adding \( \text{right} < H \) means that the new \( H \) is \( \text{right} \).

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 \( n^3 \)).

**Case (B):** For the widest \( \text{size} \)-high rectangle we have only two cases (again \( \text{right} \) is the height of the column added to the right of the current rectangle and \( \text{W} \) is the current width):

1. If \( \text{right} < \text{size} \) then there cannot be a \( \text{size} \)-high rectangle which includes this column so the new \( \text{W} \) is 0.
2. If \( \text{right} \geq \text{size} \) then we can add the column to the rectangle which will still be \( \text{size} \)-high so the new \( \text{W} \) is \( \text{W} + 1 \).

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

```plaintext
size:=0; row:=0; col:=0; i:=1; j:=0; H:=0; \{H=\text{TSW}(i,j) \land P\};
while i\leq m do
    while j<n do
        \{H=\text{TSW}(i,j) \land P\};
        j:=j+1;
        if H\geq size+1 \land \text{col}(i,j)\geq size+1
            then size:=size+1; row:=i; col:=j; H:=\text{TSW}(i,j)
            else H:=\text{TSW}(i,j) \land \text{od};
        i:=i+1; j:=0 \text{ od};
    \{i=m+1 \land j=n \land P\}.
and:
```

and:
size:=0; row:=0; col:=0; i:=1; j:=0; W:=0; {W=WSH(i,j) \land P};
while i\leq m do
  while j<n do
    {W=WSH(i,j) \land P};
    j:=j+1;
    if W\geq size \land col(i,j) \geq 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 \land j=n \land 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 \( n^2 \) assignments \( W:=WSH(i,j) \) only require order \( n^2 \) time. This is obviously better than the first version which required order \( n^3 \) 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) \land P};
while i\leq m do
  while j<n do
    {W=WSH(i,j) \land P};
    j:=j+1;
    if W\geq size \land col(i,j) \geq size+1
      then size:=size+1; row:=i; col:=j;
      if col(i,j)<size
        then W:=0
        else W:=W+1; while col(i,j-W)\geq size \land j-W>0 do W:=W+1 od fi;
      {W=WSH(i,j) \land P}
    else if col(i,j)<size
      then W:=0
      else W:=W+1 fi;
    {W=WSH(i,j) \land P} fi od;
  i:=i+1; j:=0 od;
  {i=m+1 \land j=n \land P}.
If we now maintain the size of all the true columns in the scanned area in an integer array \( \text{cols}[1..n] \) then the program becomes:

```plaintext
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;
            else if cols[j]<size
                then W:=0
                else W:=W+1 fi fi;
        i:=i+1; j:=0; W:=0 od;
    \{i=m+1 \& j=n \& P\}.
```

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

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 ∈ R where R is the set of real numbers, boolean satisfies the condition boolean ∈ {true,false} and so on. If x is a typed variable then we write \( I_x \) for the condition that x must satisfy. \( I_x \) is called the “global condition” for x. For any assignment to x we must have:

\[
\{I_x\}; \ x:=x'.Q \approx \{I_x\}; \ x:=x'.Q; \ \{I_x\}
\]

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 \( I_x \) holds initially but must ensure that it holds after the
assignment. We may assume that $I_x$ is included as part of the condition in any assignment to $x$ and therefore we interpret: $x:=x'.Q$ to mean $x:=x'.(I_x \land Q)$ if necessary. For example if $x$ and $y$ are 
like integer then the assignment: $x:=x'.(x>y)$ 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:

$I_0 = \text{"}r \text{ is one of the base cases\"}$

and for $n>0$: $I_n = \text{"}r \text{ is constructed from components which all satisfy } \bigvee_{i<n} I_i\text{"}$

Then the global condition for $r$ is:

$I = \bigvee_{n<\omega} I_n$

For example, if we have:

```plaintext
var list either atom like string
       or cons like (list,list)
```

then $I_0 = \exists x. (\text{list} = \langle 1,x \rangle \land I_{\text{string}}[x/\text{string}])$

$I_n = \exists x,y. (\text{list} = \langle 2,\langle x,y \rangle \rangle \land \bigvee_{i<n} I_i[x/\text{list}] \land \bigvee_{i<n} I_i[y/\text{list}])$

and $I_{\text{list}} = \bigvee_{n<\omega} I_n$. 