) CHAPTER SEVEN

) Linear Recusion

Introduction

In this chapter we consider transformations of linear recursive functions, ie functions whose body contains a single recursive call. We will provide proofs for the common techniques of linear recursion removal within our system: we follow the terminology of Bauer & Wossner [Bauer & Wossner 80] in discussing these techniques. They used an applicative language with no side effects and algebraic definition of specifications, we use an imperative kernel language with applicative constructs added. We extend their transformations to deal with functions and expressions with side effects and general specifications, expressed using first order logic. By using side-effects where necessary we can transate any linear recursive function (or procedure) into the following form:

```
funct L(m) ≡  if B(m) then $\phi$(L(k(m)),E(m))
        else H(m) fi.
```

Here **B(m)** is a boolean function and ϕ, **k**, **E** and **H** are functions which do not call **L** (so there is only the single recursive call of **L**). We will use the same letters for their procedural equivalents taking **b** to be the variable assigned by the procedural equivalent of **B(m)** (so we use **IC(B,b)** for the procedural equivalent of **B**). Let **r** be the variable assigned by the other procedural equivalents. Thus a procedural equivalent of **L** is:

```
proc $L_1$(m) ≡
  B(m);
  if b=tt then $L_1$(k(m)); r:=$\phi$(r,E(m))
    else r:=H(m) fi.
```

We add a global stack **S** to replace the parameter **m** since it is needed after the recursive call:

```
proc $L_1$(m) ≡  S:=⟨⟩; $L_1$. where
  $L_1$ ≡  B(m);
    if b=tt then S←m; m:=k(m); $L_1$; m←S; r:=$\phi$(r,E(m))
      else r:=H(m) fi.
```

Using the direct method of recursion removal, L_1 is equivalent to:

```
proc $L_1$(m) ≡
  S:=⟨⟩;
  do B(m);
   if b=ff then exit fi;
   S←m; m:=k(m) od;
  r:=H(m);
```

```
do if S= ⟨⟩ then exit fi;
 m←S; r:=$\phi$(r,E(m)) od.
```
We now represent the stack **S** by an array of the same name together with a variable **i** which indicates the top of the stack. **S:=⟨⟩** becomes **i:=0**, the statement **S←m** becomes **i:=i+1; S[i]:=m**, the statement **m←S** becomes **m:=S[i]; i:=i−1** and the test **S= ⟨⟩** becomes **i=0**. If we then replace **i** by **j** in the second loop we see that **j** takes the values **i**,. . . ,**1** and we can replace the loop by a **for** loop. We also apply proper inversion to the first loop and convert it to a **while** loop:
```
proc $L_1$(m) ≡
  i:=0; B(m);
  while b=tt do
   i:=i+1; S[i]:=m; m:=k(m); B(m) od;
  r:=H(m);
  for j:=i to 1 step −1 do
   r:=$\phi$(r,E(S[j])) od.
```

We will now examine various transformation techniques which, under certain circumstances, will enable us to dispense with the stack **S**.

The Technique of Re-Bracketing

Suppose we can find a function Ψ which satisfies:

$$\forall \mathbf{r,s,t}.\ \phi(\phi(\mathbf{r,s}),\mathbf{t}) = \phi(\mathbf{r},\Psi(\mathbf{s,t}))$$

ie **z:=**ϕ**(r,s); z:=**ϕ**(z,t)** $\approx$ **z:=**Ψ**(s,t); z:=**ϕ**(r,z)**.

Suppose also that (for procedural equivalents on different variables):
Ψ is independent of ϕ, **H** and **E**.
E is independent of ϕ, **H** and **E**.
m:=k(m); b:=B(m) is independent of **z:=**Ψ**(E(m),z)**
(So, for example: ϕ**(a,b);** Ψ**(c,d)** $\approx$ Ψ**(c,d);** ϕ**(a,b)** and so on.)
By "**E** is independent of **E**" we mean that for new distinct variables $\mathbf{r}_1$, $\mathbf{r}_2$, **a**, **b**:

$$\Delta \vdash \mathbf{r}_1\mathbf{:=E(a);}\ \mathbf{r}_2\mathbf{:=E(b)} \approx \mathbf{r}_2\mathbf{:=E(b);}\ \mathbf{r}_1\mathbf{:=E(a)}.$$

Then **L** is equivalent to:
```
funct $L_2$(m) ≡
  if B(m) then G(E(m),k(m))
    else H(m) fi. where
```

```
        funct G(z,m) ≡
          if B(m) then G($\Psi$(E(m),z),k(m))
            else $\phi$(H(m),z) fi.
```

A procedural equivalent of this is:

```
      proc $L_2$(m) ≡
        if B(m) then z:=E(m); m:=k(m);
                while B(m) do
                  z:=$\Psi$(E(m),z); m:=k(m) od;
                  r:=$\phi$(H(m),z)
              else r:=H(m) fi.
```

Proof: First we take out the "**i=0**" case from $\mathbf{L}_1$ (by unrolling the first step of the loop):

```
      proc $L_1$(m) ≡
        i:=0; B(m);
        if b=tt
        then i:=i+1; S[i]:=m; m:=k(m); B(m);
          while b=tt do
            i:=i+1; S[i]:=m; m:=k(m); B(m) od;
          r:=H(m);
          for j:=i to 2 step −1 do
            r:=$\phi$(r,E(S[j])) od;
          r:=$\phi$(r,E(S[1])).
        else r:=H(m) fi.
```

Claim:

If **i>0** then for any **X** which is a combination of **H**, **E** and ϕ applied to **m**: **A** $\approx$ **B** where

```
A = z:=X;
  for j:=i to 2 step −1 do
    z:=$\phi$(z,E(S[j])) od;
  z:=$\phi$(z,E(S[1])) .
```

```
B = z:=E(S[1]);
  for j:=2 to i step 1 do
    z:=$\Psi$(E(S[j]),z) od;
    z:=$\phi$(X,z).
```

Proof of Claim: By induction on **i** using the independence conditions.

Applying this to $\mathbf{L}_1$ (with **X=H(m)**) gives:

```
proc L_1(m) ≡
  i:=0; B(m);
  if b=tt
   then i:=1; S[1]:=m; m:=k(m); B(m);
     while b=tt do
       i:=i+1; S[i]:=m; m:=k(m); B(m) od;
     z:=E(S[1]);
     for j:=2 to i step 1 do
       z:=Ψ(E(S[j]),z) od;
     r:=φ(H(m),z)
   else r:=H(m) fi.
```

Notice that here the first loop (the **while**) fills the array **S** from **2** onwards with **i** counting the number of places filled. The second loop (the **for**) does the same number of iterations and reads the values assigned to **S** in the same order. We would like to "overlap" the two loops so that after each iteration of the first loop we immidiately carry out the corresponding iteration of the second loop. Then each element of the array will be assigned and accessed (and finished with) before the next element is assigned. This means that the array can be collapsed to a simple variable (for which, for convenience, we will use **m**). This gives:

```
proc L_1(m) ≡
  B(m);
  if b=tt
   then z:=E(m); m:=k(m); B(m);
     while b=tt do
       z:=Ψ(E(m),z); m:=k(m); B(m) od;
     r:=φ(H(m),z)
   else r:=H(m) fi.
```

This is the procedure $\mathbf{L}_2$ above.

All that remains is to prove the Lemma:

Lemma: Loop Overlapping:

Suppose statements **while B do i:=i+1; $\mathbf{S}_1$ od** and $\mathbf{S}_2$ are independent, except for arrays which have their **i**th element assigned in $\mathbf{S}_1$ and **j**th element accessed in $\mathbf{S}_2$, and neither $\mathbf{S}_1$ nor $\mathbf{S}_2$ assign to **i** or **j**.
Then the two loops:

i:=1; while B do i:=i+1; $\mathbf{S}_1$ od; for j:=2 to i step 1 do $\mathbf{S}_2$ od = **i:=1; DO; FOR**

are equivalent to the single loop:

$$\mathbf{i{:=}1;\ \underline{while}\ B\ \underline{do}\ i{:=}i{+}1;\ S_1;\ S_2[i/j]\ \underline{od}} = \mathbf{DO'}.$$

Proof: The trick is to move the iterations of the second loop up one at a time so that they "overlap" the iterations of the first loop.

<u>Claim:</u> For all $\mathbf{1 \leqslant q} < \omega$:

$\mathbf{\{i{=}1\};\ DO;\ FOR}$

$\quad \approx\ \mathbf{\{i{=}1\};\ \underline{while}\ B\ \wedge\ i{<}q\ \underline{do}\ i{:=}i{+}1;\ S_1;\ S_2[i/j]\ \underline{od};}$

$\quad \mathbf{\underline{while}\ B\ \underline{do}\ i{:=}i{+}1;\ S_1\ \underline{od};}$

$\quad \mathbf{\underline{for}\ j{:=}q{+}1\ \underline{to}\ i\ step\ 1\ \underline{do}\ S_2\ \underline{od}}$

$\quad = \mathbf{\{i{=}1\};\ DO'_{\mathit{q}};\ DO;\ FOR_{\mathit{q}}}$

Here we have "overlapped" up to the first $\mathbf{q{-}1}$ iterations in the first loop, and then carried out the rest of the iterations (if any) separately in the following two loops.

Proof of Claim: By induction on $\mathbf{q}$. For $\mathbf{q{=}1}$ the first loop is void so the result follows. Suppose the result holds for $\mathbf{q}$. Note that $\mathbf{i \leqslant q}$ is invariant over $\mathbf{DO'_{\mathit{q}}}$ and $\big(\mathbf{B \Rightarrow i{=}q}\big) \wedge \big(\mathbf{i{<}q \Rightarrow \neg B}\big)$ holds after $\mathbf{DO'_{\mathit{q}}}$. Also note that by loop merging $\mathbf{DO'_{\mathit{q+1}}} \approx \mathbf{DO'_{\mathit{q}};\ DO'_{\mathit{q+1}}}$.

Consider the cases: $\mathbf{B}$ holds after $\mathbf{DO'_{\mathit{q}}}$ and $\neg\mathbf{B}$ holds after $\mathbf{DO'_{\mathit{q}}}$:

If $\neg\mathbf{B}$ holds after $\mathbf{DO'_{\mathit{q}}}$ then $\mathbf{DO'_{\mathit{q+1}}} \approx \mathbf{skip}$ and $\mathbf{DO} \approx \mathbf{skip}$ and as $\mathbf{i \leqslant q}$ we have $\mathbf{FOR_{\mathit{q}}} \approx \mathbf{skip}$ and $\mathbf{FOR_{\mathit{q+1}}} \approx \mathbf{skip}$.

So $\mathbf{\{i{=}1\};\ DO;\ FOR}$

$\quad \approx\ \mathbf{\{i{=}1\};\ DO'_{\mathit{q}};\ DO;\ FOR_{\mathit{q}}}$ by induction hypothesis

$\quad \approx\ \mathbf{\{i{=}1\};\ DO'_{\mathit{q}};\ DO'_{\mathit{q+1}};\ DO;\ FOR_{\mathit{q+1}}}$

$\quad \approx\ \mathbf{\{i{=}1\};\ DO'_{\mathit{q+1}};\ DO;\ FOR_{\mathit{q+1}}}$ as required.

If $\mathbf{B}$ holds after $\mathbf{DO'_{\mathit{q}}}$ then we must have $\mathbf{i{=}q}$ and unrolling the first step of $\mathbf{DO}$ gives:

$\mathbf{\{i{=}1\};\ DO;\ FOR}$

$\quad \approx\ \mathbf{\{i{=}1\};\ DO'_{\mathit{q}};\ \{B\ \wedge\ i{=}q\};\ i{:=}i{+}1;\ S_1;\ \{i{=}q{+}1\};}$

$\quad \mathbf{DO;\ \{i \geqslant q{+}1\};\ FOR_{\mathit{q}}}$

since $\mathbf{DO}$ increases $\mathbf{i}$ and preserves $\mathbf{q}$.

So we can unroll the first step of $\mathbf{FOR_{\mathit{q}}}$ to give:

$\quad \approx\ \mathbf{\{i{=}1\};\ DO'_{\mathit{q}};\ \{B\ \wedge\ i{=}q\};\ i{:=}i{+}1;\ S_1;\ \{i{=}q{+}1\};}$

$\quad \mathbf{DO;\ S_2[q{+}1/j];\ FOR_{\mathit{q+1}}}$

By the premise $\mathbf{DO}$ and $\mathbf{S_2}$ are independent and $\mathbf{q}$ does not occur in $\mathbf{DO}$ so we can interchange the order of $\mathbf{DO}$ and $\mathbf{S_2[q{+}1/j]}$ to give:

$\quad \approx\ \mathbf{\{i{=}1\};\ DO'_{\mathit{q}};\ \{B\ \wedge\ i{=}q\};\ \underline{while}\ B\ \wedge\ i{<}q{+}1\ \underline{do}\ i{:=}i{+}1;\ S_1;\ S_2[i/j]\ \underline{od};}$

$\quad \mathbf{DO;\ FOR_{\mathit{q+1}}}$

$\quad \approx\ \mathbf{\{i{=}1\};\ DO'_{\mathit{q+1}};\ DO;\ FOR_{\mathit{q+1}}}$ by loop merging.

which proves the result for all **q**.

Hence by assertion insertion we have for all $\mathbf{1 \leqslant q < \omega}$:
$\mathbf{\{i{=}1\};\ DO;\ FOR;\ \{i{<}q\}} \;\approx\; \mathbf{\{i{=}1\};\ DO'_q;\ DO;\ FOR_q;\ \{i{<}q\}}$

Now:
$\mathbf{\{i{=}1\};\ DO'_q;\ DO;\ FOR_q;\ \{i{<}q\}}$
$\quad\approx\; \mathbf{\{i{=}1\};\ DO'_q;\ \{i{<}q\};\ DO;\ FOR_q}$ since **i** is invariant over **DO** and $\mathbf{FOR}_q$
$\quad\approx\; \mathbf{\{i{=}1\};\ DO'_q;\ \{i{<}q \wedge \neg B\};\ DO;\ FOR_q}$ since $\mathbf{i{<}q \Rightarrow \neg B}$ holds after $\mathbf{DO'_q}$.
$\quad\approx\; \mathbf{\{i{=}1\};\ DO'_q;\ \{i{<}q \wedge \neg B\};\ FOR_q}$
$\quad\approx\; \mathbf{\{i{=}1\};\ DO'_q;\ \{i{<}q \wedge \neg B\};\ DO'}$
$\quad\approx\; \mathbf{\{i{=}1\};\ DO'_q;\ DO';\ \{i{<}q\}}$
$\quad\approx\; \mathbf{\{i{=}1\};\ DO';\ \{i{<}q\}}$ by loop merging.
So the claim is proved.

Hence for all $\mathbf{1 \leqslant q < \omega}$:
$\mathbf{\{i{=}1\};\ DO;\ FOR;\ \{i{<}q\}} \;\approx\; \mathbf{\{i{=}1\};\ DO';\ \{i{<}q\}}$ where **q** only occurs in the assertions.

So $\kappa\{\mathbf{i{=}1}\} \cup \Delta \vdash \mathbf{DO;\ FOR;\ \{i{<}q\}} \;\approx\; \mathbf{DO';\ \{i{<}q\}}$ for all $\mathbf{q} < \omega$
so $\{\mathbf{i{=}1}\} \cup \Delta \vdash \bigvee_{q<\omega}\mathbf{WP(DO;\ FOR;\ \{i{<}q\},\ G(w))} \iff \bigvee_{q<\omega}\mathbf{WP(DO';\ \{i{<}q\},\ G(w))}$
so $\{\mathbf{i{=}1}\} \cup \Delta \vdash \bigvee_{q<\omega}\mathbf{WP(DO;\ FOR,\ (i{<}q) \wedge G(w))} \iff \bigvee_{q<\omega}\mathbf{WP(DO',\ (i{<}q) \wedge G(w))}$
so $\{\mathbf{i{=}1}\} \cup \Delta \vdash \mathbf{WP(DO;\ FOR,\ \bigvee_{q<\omega}(i{<}q) \wedge G(w))} \iff \mathbf{WP(DO',\ \bigvee_{q<\omega}(i{<}q) \wedge G(w))}$
by continuity of **WP**.
But $\bigvee_{q<\omega}(\mathbf{i{<}q}) \iff \mathbf{true}$
so $\{\mathbf{i{=}1}\} \cup \Delta \vdash \mathbf{WP(DO;\ FOR,\ G(w))} \iff \mathbf{WP(DO',\ G(w))}$
ie $\Delta \vdash \mathbf{\{i{=}1\};\ DO;\ FOR} \;\approx\; \mathbf{\{i{=}1\};\ DO'}$ and the result is proved.
This completes the proof of the Lemma.

Example: Suppose we have the following function **R** which takes a list as its first argument and returns another list:

```
funct R(a,y) ≡
  if B(a,y) then H(a,y)
    else R(tl(a),y)&⟨hd(a)⟩ fi.
```

Here we have $\mathbf{E(a,y)} = \langle\mathbf{hd(a)}\rangle$, $\mathbf{k(a,y)} = \langle\mathbf{tl(a),y}\rangle$ and $\phi\mathbf{(p,q)=p\&q}$.
ϕ is associative so we can apply the transformation by taking $\Psi = \phi$ to give:

```
funct R_2(a,y) ≡
  if B(a,y) then H(a,y)
    else ⌈ z:=⟨hd(a)⟩; a:=tl(a);
        while ¬B(a,y) do
                z:=hd(a)::z; a:=tl(a) od;
              H(a,y)&z ⌋ fi.
```
where we have replaced **⟨hd(a)⟩&z** by **hd(a)::z** (also called **cons(hd(a),z)**).

In a totally functional (or "applicative") style we need a sub-function to represent the **while** loop:
```
funct R_3(a,y) ≡
  if B(a,y) then H(a,y)
    else G(tl(a),⟨hd(a)⟩) fi. where
  funct G(a,z) ≡
    if B(a,y) then H(a,y)&z
      else G(tl(a),hd(a)::z) fi.
```

Note that if **B(a,y)** holds initially then **G(a,⟨⟩) = H(a,y)&⟨⟩ = H(a,y)** while if **B(a,y)** fails initially then **G(a,⟨⟩) = G(tl(a),hd(a)::z)** so we may eliminate the first **if** to give:
```
funct R_3(a,y) ≡
  G(a,⟨⟩). where
  funct G(a,z) ≡
    if B(a,y) then H(a,y)&z
       else G(tl(a),hd(a)::z) fi.
```

This transformation would be described as "adding an accumulating parameter **z**" by devotees of functional programming.

The Technique of Operand Commutation

Suppose we can find a function Ψ which satisfies:

$$\forall r,s,t.\phi(\Psi(r,s),E(t))= \Psi(\phi(r,E(t)),s)$$

ie $z{:=}\Psi(r,s);\ z{:=}\phi(z,E(t)) \ \approx\ z{:=}\phi(r,E(t));\ z{:=}\Psi(z,s)$.
and also: $\forall r.\Psi(H(m_0),r)= \phi(H(m_0),r)$ where m_0 is the "argument on termination" (see below).
Suppose also that: **m:=k(m); b:=B(m)** is independent of **r:=Ψ(r,E(m))**.

Then **L** is equivalent to:

```
funct L_3(m) ≡
  G(H(m_0),z). where
  funct G(z,m) ≡
    if B(m) then G($\Psi$(z,E(m)),k(m))
       else z fi.
```

The "argument on termination" $\mathbf{m}_0$ is the final value put onto the stack. With this technique the other values are used in the order they are put on the stack except the final value which will be needed first. So if this value can be determined in advance we can apply "loop overlapping" and do away with the stack.
Recall our procedural equivalent for **L**:

```
proc L_1(m) ≡
  i:=0; B(m);
  while b=tt do
   i:=i+1; S[i]:=m; m:=k(m); B(m) od;
  r:=H(m);
  for j:=i to 1 step −1 do
   r:=$\phi$(r,E(S[j])) od.
```

__Claim:__

```
A = r:=H(m);  ≈  B = r:=H(m)
  for j:=i to 1 step −1 do                  for j:=1 to i step 1 do
    r:=$\phi$(r,E(S[j])) od.                        r:=$\Psi$(r,E(S[j])) od
```

Note that this only needs to be true when **m** is an argument on termination, for example $\neg\mathbf{B(m)}$ is a sufficient condition.

__Proof__ __of__ __Claim:__ Let $\mathbf{b{=}H(m_0)}$ where $\mathbf{m}_0$ is the initial value of **m**.
The proof is by induction on **i**; the result is trivial for $\mathbf{i \leqslant 0}$.
We introduce the abbreviations: For any integers **p**,**q** let

```
$\phi$[p..q] =_DF for j:=p to q step −1 do
                  r:=$\phi$(r,E(S[j])) od
```

and

```
$\Psi$[p..q] =_DF for j:=p to q step 1 do
                  r:=$\Psi$(r,E(S[j])) od.
```

So we wish to prove: **r:=b;** ϕ**[i..1]** $\approx$ **r:=b;** Ψ**[1..i]**.

Base Step: $\mathbf{i=1}$:
From the premise $\phi(\mathbf{b,x}) = \Psi(\mathbf{b,x})$ we get:
$\mathbf{A} \approx \mathbf{r{:=}b;\ r{:=}}\phi\mathbf{(r,E(S[1]))} \approx \mathbf{r{:=}}\phi\mathbf{(b,E(S[1])} \approx \mathbf{r{:=}}\Psi\mathbf{(b,E(S[1])} \approx \mathbf{B}$.

Induction Step: Suppose $\mathbf{i>1}$ and the result holds for smaller $\mathbf{i}$:
Claim 1: For $\mathbf{-1 \leqslant l \leqslant i-1}$:
$\mathbf{r{:=}b;}\ \phi\mathbf{[i-1..0]} \approx \mathbf{r{:=}b;}\ \phi\mathbf{[l..0];}\ \Psi\mathbf{[l+1..i-1]}$.
The case $\mathbf{l=i-1}$ is trivial so suppose $\mathbf{-1 \leqslant l < i-1}$ and the result holds for $\mathbf{l+1}$ and all smaller $\mathbf{i}$:
$\mathbf{r{:=}b;}\ \phi\mathbf{[i-1..0]} \approx \mathbf{r{:=}b;}\ \phi\mathbf{[l+1..0];}\ \Psi\mathbf{[l+2..i-1]}$ by induction hypothesis.
$\approx \mathbf{r{:=}b;\ r{:=}}\phi\mathbf{(r,E(S[l+1]));}\ \phi\mathbf{[l..0];}\ \Psi\mathbf{[l+2..i-1]}$ since $\mathbf{0 \leqslant l+1}$.
$\approx \mathbf{r{:=}b;\ r{:=}}\Psi\mathbf{(r,E(S[l+1]));}\ \phi\mathbf{[l..0];}\ \Psi\mathbf{[l+2..i-1]}$ by premise.
Claim 2: For any $\mathbf{c}$ and any $\mathbf{l}$:
$\mathbf{r{:=}}\Psi\mathbf{(r,c);}\ \phi\mathbf{[l..0]} \approx \phi\mathbf{[l..0];\ r{:=}}\Psi\mathbf{(r,c)}$.
To prove this we prove:
$\mathbf{r{:=}}\Psi\mathbf{(r,c);}\ \phi\mathbf{[l..0]} \approx \phi\mathbf{[l..k];\ r{:=}}\Psi\mathbf{(r,c);}\ \phi\mathbf{[k-1..0]}$ for all $\mathbf{0 \leqslant k \leqslant l+1}$.
by induction on $\mathbf{k}$.

Hence: $\mathbf{r{:=}b;}\ \phi\mathbf{[i-1..0]}$
$\approx \mathbf{r{:=}b;}\ \phi\mathbf{[l..0];\ r{:=}}\Psi\mathbf{(r,E(S[l+1]));}\ \Psi\mathbf{[l+2..i-1]}$ by Claim 2.
$\approx \mathbf{r{:=}b;}\ \phi\mathbf{[l..0];}\ \Psi\mathbf{[l+1..i-1]}$
which proves Claim 1.

So putting $\mathbf{l= -1}$ gives:
$\mathbf{r{:=}b;}\ \phi\mathbf{[i-1..0]}$
$\approx \mathbf{r{:=}b;}\ \phi\mathbf{[-1..0];}\ \Psi\mathbf{[0..i-1]}$
$\approx \mathbf{r{:=}b;}\ \Psi\mathbf{[0..i-1]}$ since $\phi\mathbf{[-1..0]} \approx \mathbf{skip}$.

Using this claim we get the version:

```
proc L_1(m) ≡
  i:=0; B(m);
  while b=tt do
   i:=i+1; S[i]:=m; m:=k(m); B(m) od;
  r:=H(m);
  for j:=1 to i step 1 do
   r:=Ψ(r,E(S[j])) od.
```

We cannot apply loop overlapping directly to this because of the assignment $\mathbf{r{:=}H(m)}$ between the two loops (Note that **m** is assigned in each iteration of the first loop and **r** is accessed in the first iteration of the second loop and we need the final value of **m** to calculate the first value of **r**). If we can find or calculate the value assigned to **r** between the loops then loop overlapping can be applied since this assignment to **r** can be moved to before the first loop. Suppose we have a statement $\mathbf{S'}$ which is such that:

```
L_1(m)  ≈  S'; i:=0; B(m);
   while b=tt do
     i:=i+1; S[i]:=m; m:=k(m); B(m) od;
   {m=m_0}; r:=H(m);
   for j:=1 to i step 1 do
     r:=Ψ(r,E(S[j])) od.
```

Then we can replace the assignment **r:=H(m)** by $\mathbf{r{:=}H(m_0)}$ and then move it to before the first loop (since $\mathbf{m_0}$ and **r** are not used in that loop). This enables us to apply loop overlapping which gives:

```
L_1(m)  ≈  S'; r:=H(m_0); i:=0; B(m);
   while b=tt do
     i:=i+1; S[i]:=m; m:=k(m); B(m); r:=Ψ(r,E(S[i])) od.
```

We can move **r:=Ψ(r,E(S[i]))** to before **m:=k(m); B(m)** (by independence) and then replace **S[i]** by **m**. Then we can remove **S** and **i** to get:

```
L_1(m)  ≈  S'; r:=H(m_0); B(m);
   while b=tt do
     r:=Ψ(r,E(m)); m:=k(m); B(m) od.
```

In the case where **H** does not depend on **m** (eg if **H(m)** is a constant $\mathbf{c_0}$) then we don't need the argument on termination since we know $\mathbf{H(m){=}c_0}$ at the end of the first loop so we replace **r:=H(m)** by $\mathbf{r{:=}c_0}$ and move the assignment to the beginning to give:

```
L_1(m)  ≈  r:=c_0; B(m);
   while b=tt do
     r:=Ψ(r,E(m)); m:=k(m); B(m) od.
```

Another special case is where there is only one value of **m** such that **B(m)** returns **ff** in variable **b**, that is if: $\mathbf{B(m)} \approx \mathbf{B(m); \{b{=}ff \Rightarrow m{=}m_0\}}$. Inserting this assertion at the end of the loop body gives:

```
L_1(m)  ≈  i:=0; B(m); {b=ff ⇒ m=m_0};
   while b=tt do
     i:=i+1; S[i]:=m; m:=k(m); B(m); {b=ff ⇒ m=m_0} od;
   {b=ff ∧ (b=ff ⇒ m=m_0)};
   {m=m_0}; r:=H(m);
   for j:=1 to i step 1 do
     r:=Ψ(r,E(S[j])) od.
```

Thus the argument on termination must be $\mathbf{m_0}$, which can be assigned to **r** at the beginning.

For the general case, if **k** and **B** are determinate and have no side-effects then we can determine $\mathbf{m_0}$ by "precomputation". We double the first **while** loop (saving and restoring the initial value of **m**) to get:

```
L_1(m) ≈ m′:=m; i:=0; B(m);
    while b=tt do
      i:=i+1; S[i]:=m; m:=k(m); B(m) od;
    m_0:=m;
    m:=m′; i:=0; B(m);
    while b=tt do
      i:=i+1; S[i]:=m; m:=k(m); B(m) od;
    {m=m_0}; r:=H(m);
    for j:=1 to i step 1 do
      r:=Ψ(r,E(S[j])) od.
```

Move the assignment to **r** to after the first copy of the **while** loop, replace **m** in the first loop by $\mathbf{m_0}$ (and remove $\mathbf{m'}$). Now we can apply loop overlapping and remove the stack (and the variable **i**) to get: $\mathbf{L_1(m)} \approx$ $\mathbf{m_0}$**:=m; B(**$\mathbf{m_0}$**);**

```
    while b=tt do
      m_0:=k(m_0); B(m_0) od;
    B(m); r:=H(m);
    while b=tt do
      r:=Ψ(r,E(m)); m:=k(m); B(m) od.
```

This does away with the stack without us needing to know the argument on termination in advance but at the expense of computing the sequence of values of **m** twice over. However, if **k** and **B** are efficient compared with Ψ then the resulting procedure will be only slightly less efficient than the original function but using a small and fixed amount of store instead of a variable amount.

To summarise: the functional forms of these different versions are:

```
funct L_3(m) ≡
 ⌈S′; G(m,H(m_0))⌋. where
  funct G(m,z) ≡
   if B(m) then G(k(m),Ψ(z,E(m)))
      else z fi.
```

If $\mathbf{H(m)}=\mathbf{c}_0$ for all **m** then we have the version:

```
funct L_3(m) ≡
  G(m,c_0). where
  funct G(m,z) ≡
    if B(m) then G(k(m),Ψ(z,E(m)))
       else z fi.
```

Example: The Cooper Transformation:

Defn: ρ is right-commutative if $(\mathbf{a}\ \rho\ \mathbf{b})\ \rho\ \mathbf{c} = (\mathbf{a}\ \rho\ \mathbf{c})\ \rho\ \mathbf{b}$.

Note that + and **.** are commutative and associative and so are right-commutative, but so also are − and / since $(\mathbf{a}-\mathbf{b})-\mathbf{c} = (\mathbf{a}-\mathbf{c})-\mathbf{b}$ and $(\mathbf{a}/\mathbf{b})/\mathbf{c} = (\mathbf{a}/\mathbf{c})/\mathbf{b}$.

If ρ is right-commutative and **a**, **b** are integers then we have the following transformation (called the "Cooper transformation"):

```
funct F(m,n) ≡  if m>0 then F(m−1,n) $\rho$ m
        else n fi.
 ≈
funct F(m,n) ≡  if m>0 then F(m−1, n $\rho$ m)
        else n fi.
```

This was described by Cooper in [Cooper 66]. A variant of it was used in [Darlington & Burstall 76] and a special case derived in [Morris 71].

This can be seen as a special case of our result.
Let **m** be an integer and set:
$\mathbf{B(m)} \iff \mathbf{a>0}$,
$\mathbf{E(m)} = \mathbf{m}$,
$\mathbf{H(m)} = \mathbf{n}$ (which does not depend on **m**), $\mathbf{k(m)} = \mathbf{m-1}$ and
$\phi(\mathbf{x,y}) = \mathbf{x}\ \rho\ \mathbf{y}$.
Then since ρ is right-commutative we may set $\Psi = \phi$ and:

```
          funct F(m,n) ≡
            L(m). where
            funct L(m) ≡  if m>0 then L(m−1) $\rho$ m
                   else n fi.
```

From this our transformation gives:

```
funct F(m,n) ≡
  L_3(m). where
  funct L_3(m) ≡
    G(m,n). where
    funct G(m,z) ≡
      if m>0 then G(m−1, z $\rho$ m)
         else z fi.
```
which we can unfold to:
```
funct F(m,n) ≡
  G(m,n). where
  funct G(m,z) ≡
    if m>0 then G(m−1, z $\rho$ m)
       else z fi.
```
which is equivalent to:
```
funct F(m,n) ≡  if m>0 then F(m−1, n $\rho$ m)
         else n fi.
```

If **k** and **B** are determinate and have no side-effects then we have the version:
```
funct L_3(m) ≡
  G(m,H(F(m))) where
  funct F(n) ≡
    if B(n) then F(k(n))
       else n fi ,
  funct G(m,z) ≡
    if B(m) then G(k(m),$\Psi$(z,E(m)))
       else z fi.
```
A special case is when ϕ does not depend on the second argument, ie when ϕ**(r,s)**= γ**(r)**. Here right-commutivity holds automatically and we can take $\Psi = \phi$ ie Ψ**(r,s)**= γ**(r)**. We get for example:
```
funct L_3(m) ≡
  G(m,c_0). where
  funct G(m,z) ≡
    if B(m) then G(k(m),$\gamma$(z))
       else z fi.
```
This transformation can be found in [Morris 71].

The Technique of Function Inversion

If there exists an inverse for the function **k** on at least a certain part of its domain then we can avoid the need for the stack by reconstructing the stacked values from the argument on termination.
We have the following procedural version of **L**:

```
proc $L_1(m) \equiv$ S:=$\langle\rangle$; $L_1$. where
  $L_1 \equiv$ B(m);
    if b=tt then S←m; m:=k(m); $L_1$; m←S; r:=$\phi$(r,E(m))
      else r:=H(m) fi.
```

If **k** has inverse **k′** which is such that we can insert the assertions:

```
proc $L_1(m) \equiv$ S:=$\langle\rangle$; $L_1$. where
  $L_1 \equiv$ B(m);
    if b=tt then S←m; m:=k(m); {hd(S)=k′(m)}; $L_1$;
         {hd(S)=k′(m)}; m←S; r:=$\phi$(r,E(m))
       else r:=H(m) fi.
```

then we can replace **{hd(S)=k′(m)}; m←S** by **m:=k′(m); S:=tl(S)**. This means that the stack is not needed so we get:

```
proc $L_1(m) \equiv$ $L_1$. where
  $L_1 \equiv$ B(m);
    if b=tt then m:=k(m); $L_1$; m:=k′(m); r:=$\phi$(r,E(m))
      else r:=H(m) fi.
```

In order to remove the recursion in the usual way we add a count (**i**) to enable us to distinguish the outer call from inner calls. The direct method of recursion removal then gives:

```
proc $L_1(m) \equiv$
    i:=0;
   B(m);
   while b=tt do
     m:=k(m); i:=i+1; B(m) od;
   r:=H(m);
   while i>0 do
     i:=i−1; m:=k′(m); r:=$\phi$(r,E(m)) od.
```

If both **k** and **B** are defined and determinate and **Dom(L)** is the required domain for **L** then a sufficient condition on **k′** for this to work is:

$$\forall x \in \{k^i(m) | i \geqslant 0 \wedge m \in \mathrm{Dom}(L) \wedge \forall j.0 \leqslant j \leqslant i \Rightarrow B(k^j(m))\},\ k'(k(x))=x$$

For example if **k** is doubling of integers then it does not have an inverse for all integers but the inverse (halving of even integers) exists wherever it is needed so this is a suitable candidate for function inversion.

If the sequence **m**, **k(m)**, $\mathbf{k}^2$**(m)**,... does not repeat so long as **B(**$\mathbf{k}^i$**(m))** holds then if we use **y** instead of **m** in the loops we have **i=0** $\iff$ **y=m** and we can therefore remove the count by changing the test on the second loop to **y=m**:

```
proc L$_1$(m) $\equiv$
   y:=m; B(y);
   while b=tt do
     y:=k(y); B(y) od;
   r:=H(y);
   while y$\neq$m do
     y:=k$'$(y); r:=$\phi$(r,E(y)) od.
```

Sometimes however, particularly if **m** (and therefore **y** also) is a large data structure, the test of equality **y≠m** can be much more expensive than testing if an integer is zero. Then leaving in **i** will give a more efficient version.
A functional version is:

```
funct L$_4$(m) $\equiv$
   $\lceil$m$_0$:=P(m); R(m$_0$,H(m$_0$))$\rfloor$. where
  funct P(n) $\equiv$
   if B(n) then P(k(n))
      else n fi,
  funct R(y,z) $\equiv$
   if y$\neq$m then $\lceil$y:=k$'$(y); G(y,$\phi$(z,E(y)))$\rfloor$
      else z fi.
```

If in addition the argument on termination $\mathbf{m}_0$ can be determined easily and **B** and **k** are defined and determinate then the first loop can be replaced by **y:=**$\mathbf{m}_0$:

```
proc L$_1$(m) $\equiv$
   y:=m$_0$; r:=H(m$_0$);
   while y$\neq$m do
     m:=k$'$(m); r:=$\phi$(r,E(m)) od.
```

This has the functional version:

```
funct $L_4$(m) ≡
  ⌈ r:=H($m_0$); y:=$m_0$;
  while y≠m do
    y:=k′(y); r:=$\phi$(r,E(y)) od;
  r⌋.
```

Another kind of function inversion can be used when sequences are being processed from one end: we may be able to transform the function to process the sequence from the other end. For example:

```
funct L(m) ≡
  if m≠ ⟨⟩ then $\phi$(L(tl(m)),E(hd(m)))
      else $c_0$ fi.
```

which may be transformed to:

```
funct $L_1$(m) ≡
  G(m,$c_0$). where
  funct G(y,z) ≡
    if y≠ ⟨⟩ then G(upper(y), $\phi$(z,E(bot(y))))
        else z fi.
```

where **bot(S)** $=_{DF}$ **S[ℓ(S)]** and **upper(S)** $=_{DF}$ **S[1..ℓ(S)$-$1]**.

The procedural version of **L** is:

```
proc $L_1$(m) ≡
  i:=0;
  while m≠ ⟨⟩ do
    i:=i+1; S[i]:=m; m:=tl(m) od;
  r:=$c_0$;
  for j:=i to 1 step −1 do
    r:=$\phi$(r,E(hd(S[j]))) od.
```

Note that we only want the **hd** of each element on the stack so we only have to store **hd(m)**:

```
proc $L_1$(m) ≡
  i:=0;
  while m≠ ⟨⟩ do
    i:=i+1; S[i]:=hd(m); m:=tl(m) od;
  r:=$c_0$;
  for j:=i to 1 step −1 do
    r:=$\phi$(r,E(S[j])) od.
```

The effect of the first loop is to set **i** to ℓ**(m)**, and **S** to **m** and **m** to $\langle\rangle$. **m** is not needed after the first loop so we can use it instead of **S** in the second loop by replacing **S[j]** with **m[j]**.

```
proc L_1(m) ≡
  i:=ℓ(m);
  r:=c_0;
  for j:=i to 1 step −1 do
    r:=φ(r,E(m[j])) od.
```

This **for** loop processes the elements of **m** in reverse order so since **m** is not needed after the loop,we can represent **m** by **m[1..j]**, then mEj] becomes **bot(m)**:

```
proc L_1(m) ≡
  i:=ℓ(m);
  r:=c_0;
  for j:=i to 1 step −1 do
    r:=φ(r,E(bot(m))); m:=upper(m) od.
```

Now within the loop **j>i** $\iff$ **m**= $\langle\rangle$ so we can replace the **for** by a **while** and remove **i** and **j**:

```
proc L_1(m) ≡
  r:=c_0;
  while m≠ ⟨⟩ do
    r:=φ(r,E(bot(m))); m:=upper(m) od.
```

This leads to the functional equivalent given above.

Primitive Recursive Functions

The general "Induction Scheme" for a primitive recursive function **f** is:

$$\mathbf{f(m,0) = g(m)}$$
$$\mathbf{f(m,n{+}1) = h(f(m,n),m,n)}$$

This can be expressed (using the theorem on recursive implementation of specifications) as the function:

```
funct f(m,n) ≡
  if n>0 then h(f(m,n−1),m,n−1)
     else g(m) fi.
```

Here $\mathbf{k(\langle m,n\rangle) = \langle m,n{-}1\rangle}$ and $\mathbf{E(\langle m,n\rangle) = \langle m,n{-}1\rangle}$ and ϕ =**h**.
$\mathbf{k'}$ exists and $\mathbf{m}_0$ =**0** so we can apply function inversion to get:

```
proc f(m,n) ≡
  r:=g(m); y:=0;
  while y≠n do
    y:=y+1; r:=h(r,k,y−1) od.
```
which can be transformed to a **for** loop:
```
proc f(m,n) ≡
  r:=g(m);
  for y:=0 to n−1 do
    r:=h(r,k,y) od.
```

Hence the class of primitive recursive functions is not wider than the class of functions defined by repetitive routines and systems (cf [Rice 65]).

There is in fact a completely general transformation of a linear recursive function like **L** to an iterative form which does not use any stacks, (provided **B** and **k** are defined and determinate and have no side effects) but this is very inefficient and therefore only of theoretical interest. The trick is to recompute the values **S[j]** each time they are needed (using the fact that $\mathbf{S[j]}=\mathbf{k}^{j-1}\mathbf{(m)}$) and so do away with the stack. We add a function **kj(m,j)** which computes $\mathbf{k}^{j}\mathbf{(m)}$ and then replace the accesses to the stack by calls to this function:
```
proc L_5(m) ≡
  i:=0; m_0:=m; B(m_0);
  while b=tt do
   i:=i+1; m_0:=k(m_0); B(m_0) od;
  r:=H(m_0);
  for j:=i to 1 step −1 do
    r:=$\phi$(r,E(kj(m,j−1))) od. where
  funct kj(m,j) ≡
    if j=0 then m
       else kj(k(m),j−1) fi.
```
We may convert function **kj** to a procedure returning the result in a new variable $\mathbf{r}_1$. This procedure can be converted to a **for** loop and copied in to the body of $\mathbf{L}_5$:
```
proc L_5(m) ≡
  i:=0; m_0:=m; B(m_0);
  while b=tt do
   i:=i+1; m_0:=k(m_0); B(m_0) od;
  r:=H(m_0);
```

```
for j:=i to 1 step −1 do
  $r_1$:=m;
  for l:=j−1 to 1 step −1 do
    $r_1$:=k($r_1$) od;
  r:=$\phi$(r,E($r_1$)) od.
```

EXAMPLES:

We now give some examples of applying these techniques to simple recursive functions.

Exponentiation

Suppose we wish to design a function to evaluate $\mathbf{x}^n$ for positive integers **n** using only addition and multiplication. Thus we want a function equivalent to:

$$\underline{\textbf{funct}}\ \mathbf{expt(x,n)} \equiv \mathbf{x}^n.$$

which is implemented in terms of addition and multiplication.

Noting that $\mathbf{x}^0 = \mathbf{1}$ and $\mathbf{x}^{n+1} = \mathbf{x.x}^n$ we split **expt** into two cases:

```
funct expt(x,n) ≡
  if n=0 then ⌈{n=0}; $x^n$⌋
    else ⌈{n>0}; $x^n$⌋.
```

Then apply the formulas given:

```
funct expt(x,n) ≡
  if n=0 then ⌈{n=0}; 1⌋
    else ⌈{n>0}; $x.x^{n-1}$⌋.
```

Then by the theorem on recursive implementation of a specification this is equivalent to:

```
funct expt(x,n) ≡
  if n=0 then 1
    else x.expt(x,n−1).
```

The argument on termination (**n=0**) is known and multiplication is commutative and associative so we can use either re-bracketing or operand commutation. Putting:

$\mathbf{k(x,n)} = \langle \mathbf{x,n-1} \rangle$, $\mathbf{E(x,n)=x}$, $\mathbf{H(x,n)=1}$, $\phi\mathbf{(a,b)=a.b}$, $\mathbf{B(x,n)} \iff \mathbf{n \neq 0}$

gives:

```
funct expt(x,n) ≡
  if B(x,n) then $\phi$(expt(k(x,n)), E(x,n))
    else H(x,n) fi.
```

Re-bracketing and operand commutation both give the same result namely:

```
proc expt(x,n) ≡
  if n≠0 then z:=x; n:=n−1;
      while n≠0 do
       z:=x.z; n:=n−1 od;
      z:=z.1
    else z:=1 fi.
```

which by loop rolling and using the the fact that **1.x=x**, this is equivalent to:

```
proc expt(x,n) ≡
  z:=1;
  while n≠0 do
    z:=x.z; n:=n−1 od.
```

The function **k** has an inverse **k′(x,n)=** $\langle$**x+1,n**$\rangle$, so function inversion gives:

```
proc expt(x,n) ≡
  y:=n; while y≠0 do y:=y−1 od;
  z:=1;
  while y≠n do
    y:=y+1; z:=x.z od.
```

The first loop simplifies to **y:=0** whence the second loop can be written as a **for** loop:

```
proc expt(x,n) ≡
  z:=1;
  for y:=0 to n step 1 do
    z:=x.z od.
```

A more efficient version of **expt** can be derived by using the fact that $\mathbf{x}^{2.n} = \big(\mathbf{x.x}\big)^n$. Taking out the cases for even and non-zero **n** we get the function:

```
funct expt(x,n) ≡
  if n=0 → 1
   □ n≠0 → x.expt(x,n−1)
   □ n≠0 ∧ even(n) → expt(x.x,n/2) fi.
```

For $\mathbf{n{\neq}0}$ and **even(n)**, $\mathbf{n/2}$ will reduce **n** by more than **1** so we should choose the third alternative wherever possible. We can ensure this by strengthening the guard on the second alternative to $\mathbf{n{\neq}0} \wedge \neg$**even(n)** which is $\mathbf{n{\neq}0} \wedge$ **odd(n)**. This can be written as simply **odd(n)** since **odd(n)**$\Rightarrow$**n**$\neq$**0**. We get:

```
funct expt(x,n) ≡
  if n=0  →  1
   □ odd(n)  →  x.expt(x,n−1)
   □ n≠0 ∧ even(n)  →  expt(x.x,n/2) fi.
```

This is deterministic so can be written as:

```
funct expt(x,n) ≡
  if n=0 then 1
  elsf even(n) then expt(x.x,n/2)
      else x.expt(x,n−1) fi.
```

Re-arrange the tests:

```
funct expt(x,n) ≡
  if n≠0 ∧ even(n) then expt(x.x,n/2)
        else if n=0 then 1
             else x.expt(x,n−1) fi.
```

The first call is in a terminal position so we can apply tail-recursion to get:

```
funct expt(x,n) ≡
⌈while n≠0 ∧ even(n) do x:=x.x; n:=n/2 od;
  if n=0 then 1
    else x.expt(x,n−1) fi ⌋.
```

The assertion $\mathbf{n{\neq}0}$ is invariant over the loop ($\mathbf{n{\neq}0} \Rightarrow \mathbf{n/2{\neq}0}$) so we take out the $\mathbf{n{=}0}$ case and remove the test from the loop:

```
funct expt(x,n) ≡
  if n=0 then 1
    else ⌈ while even(n) do x:=x.x; n:=n/2 od;
      x.expt(x,n−1) ⌋ fi.
```

This is now in the same form as our standard linear recursion (note the use of side-effecting expressions):

DO = while even(n) do x:=x.x; n:=n/2 od
k(x,n)=⌈DO; ⟨x,n−1⟩⌋
E(x,n)=⌈DO; x⌋= k(x,n)[1]
ϕ**(a,b)= a.b**

Function inversion cannot now be used (without needing as stack) since the function **k** is not one to one (for example **k(2,4)**=**k(4,2)**=**k(16,1)**= ⟨**16,0**⟩).

Re-bracketing:
This gives the procedural equivalent:

```
proc expt(x,n) ≡
  if n≠0 then  z:=E(x,n); ⟨x,n⟩:=k(x,n);
      while n≠0 do
            z:=z.E(x,n); ⟨x,n⟩:=k(x,n) od;
      z:=1.z
    else z:=1 fi.
```

Now **z:=E(x,n); ⟨x,n⟩:=k(x,n)**
$\approx$ **z:=k(x,n)[1]; ⟨x,n⟩:=k(x,n)**
k is determinate so we can replace the two calls by a single one:
$\approx$ $\langle x_1,n_1\rangle$:=**k(x,n)**; z:=x_1; ⟨x,n⟩:=$\langle x_1,n_1\rangle$
$\approx$ $\langle x_1,n_1\rangle$:=⟨x,n⟩; **DO**[x_1,n_1/x,n]; n_1:=n_1-1; z:=x_1; ⟨x,n⟩:=$\langle x_1,n_1\rangle$
$\approx$ **DO; n:=n−1; z:=x**
Similarly **z:=z.E(x,n); ⟨x,n⟩:=k(x,n)** $\approx$ **DO; n:=n−1; z:=z.x**

So our procedure becomes:

```
proc expt(x,n) ≡
  if n≠0 then  DO; n:=n−1; z:=x;
          while n≠0 do
            DO; n:=n−1; z:=z.x od
    else z:=1 fi.
```

Insert **z:=1** at the beginning, replace **z:=x** by **z:=z.x** and **z:=1** by **skip** and then roll the first step of the loop to get:

```
proc expt(x,n) ≡
  z:=1;
  while n≠0 do
    DO; n:=n−1; z:=z.x od.
```

The result is:

```
proc expt(x,n) ≡
  z:=1;
  while n≠0 do
    while even(n) do
      x:=x.x; n:=n/2 od;
    n:=n−1; z:=z.x od.
```

Operand commutation:

The problem is in determining the argument on termination, clearly **n=0** but the value of **x** is not directly available. However this does not matter as **H** is independent of its arguments. **H(x,n)=1** so put $\mathbf{c}_0$ =**1**. We get:

```
proc expt(x,n) ≡
  z:=1;
  while n≠0 do
    z:=z.E(x,n); m:=k(m) od.
```

This gives the same result as re-bracketing.