CHAPTER SIX

Functions and Side-Effects

Introduction

In this Chapter we extend our system to deal with functions and expressions with side effects. So far all our expressions and conditions have been terms and formulae in infinitary logic, which by their nature cannot invoke side effects; this has enabled us to keep our transformations simple. To allow side effects we introduce the new notation of "expression brackets", $\ulcorner$ and $\lrcorner$. These allow us to include statements as part of an expression, for example the following are valid expressions:

$\ulcorner$**x:=x+1; x**$\lrcorner$

$\ulcorner$**x:=x+1; x−1**$\lrcorner$

$\ulcorner$**if** **x>0** **then** **x** **else** **−x** **fi**$\lrcorner$

The first is equivalent to **x++** in C, an expression which returns the value **x+1** and has the side effect of incrementing **x**. The second is equivalent to **++x** which returns the value of **x** before the incrementation. The third expression gives the same result as $|\mathbf{x}|$ the absolute-value function.

We define our functions in terms of procedures. A function returns a value, so within a statement a function call can be used instead of a term. So a function call can replace a term in a formula, or it can return a boolean value and replace a subformula of a formula. We can even have function calls as parameters to other functions and procedures.

Defn: If **S** is a statement and **t** a term then the "generalised expression":

$$\ulcorner \mathbf{S;t} \lrcorner$$

is interpreted by replacing any assignment **a:=**$\ulcorner$**S;t**$\lrcorner$ by **S;a:=t**.
The brackets $\ulcorner$ and $\lrcorner$ are called "expression brackets".

The final component of a generalised expression can be a conditional expression of the form: **if B then** $\mathbf{t}_1$ **else** $\mathbf{t}_2$ **fi** where $\mathbf{t}_1$ abd $\mathbf{t}_2$ are expressions. In this case it is interpreted as the generalised expression $\ulcorner$**if B then r:=**$\mathbf{t}_1$ **else r:=**$\mathbf{t}_2$ **fi; r**$\lrcorner$ where **r** is a new (local) variable. Hence the "conditional assignment": **a:=if B then** $\mathbf{t}_1$ **else** $\mathbf{t}_2$ **fi** may be interpreted as: **beg r: if B then r:=**$\mathbf{t}_1$ **else r:=**$\mathbf{t}_2$ **fi; a:=r end** which may be transformed to the equivalent: **if B then a:=**$\mathbf{t}_1$ **else a:=**$\mathbf{t}_2$ **fi**. The expression may be used within a formula, for example:

$$\mathbf{if}\ \ulcorner \mathrm{S}_1;\mathrm{t}_1 \lrcorner \leqslant \ulcorner \mathrm{S}_2;\mathrm{t}_2 \lrcorner\ \mathbf{then}\ \mathrm{S}'\ \mathbf{fi}$$

becomes:

$$\mathbf{beg}\ \mathrm{r}_1,\mathrm{r}_2\text{: } \mathrm{S}_1;\ \mathrm{r}_1:=\mathrm{t}_1;\ \mathrm{S}_2;\ \mathrm{r}_2:=\mathrm{t}_2;\ \mathbf{if}\ \mathrm{r}_1 \leqslant \mathrm{r}_2\ \mathbf{then}\ \mathrm{S}'\ \mathbf{fi}\ \mathbf{end}.$$

which is equivalent to:

$$\underline{\textbf{beg}}\ \mathbf{r:}\ \mathbf{S_1;\ r{:=}t_1;\ S_2;}\ \underline{\textbf{if}}\ \mathbf{r \leqslant t_2}\ \underline{\textbf{then}}\ \mathbf{S'}\ \underline{\textbf{fi}}\ \underline{\textbf{end}}.$$

If also $\mathbf{t_1}$ is invariant over $\mathbf{S_2}$ then both variables can be eliminated:

$$\mathbf{S_1;\ S_2;}\ \underline{\textbf{if}}\ \mathbf{t_1 \leqslant t_2}\ \underline{\textbf{then}}\ \mathbf{S'}\ \underline{\textbf{fi}}.$$

We may also have generalised boolean expressions: To define these we add the constants **tt** and **ff** which represent distinct values. (ie we include **tt** and **ff** in our set of constants and **tt**$\neq$**ff** in our set Δ of formulae). Then if **S** is a statement and **Q** a formula we represent a statement of the form:

$\underline{\textbf{if}}\,\lceil \mathbf{S;Q} \rfloor\, \underline{\textbf{then}}\ \mathbf{S_1}\ \underline{\textbf{else}}\ \mathbf{S_2}\ \underline{\textbf{fi}}$ by $\mathbf{S;}\ \underline{\textbf{if}}\ \mathbf{Q}\ \underline{\textbf{then}}\ \mathbf{S_1}\ \underline{\textbf{else}}\ \mathbf{S_2}\ \underline{\textbf{fi}}$

and a loop of the form:

$\underline{\textbf{while}}\,\lceil \mathbf{S;Q} \rfloor\, \underline{\textbf{do}}\ \mathbf{S'}\ \underline{\textbf{od}}$ by the loop: $\underline{\textbf{do}}\ \mathbf{S;}\ \underline{\textbf{if}}\ \neg\mathbf{Q}\ \underline{\textbf{then}}\ \underline{\textbf{exit}}\ \underline{\textbf{fi}}\mathbf{;\ S'}\ \underline{\textbf{od}}$

which may also be written as: $\mathbf{S;}\ \underline{\textbf{while}}\ \mathbf{Q}\ \underline{\textbf{do}}\ \mathbf{S';\ S}\ \underline{\textbf{od}}$.

If we need to assign the result of a boolean expression to a (boolean) variable then we assign one of the constants **tt** or **ff** to the variable according as the expression evaluates to true or false. For example:

$\mathbf{b{:=}}\lceil \mathbf{S;Q} \rfloor$ becomes: $\mathbf{S;}\ \underline{\textbf{if}}\ \mathbf{Q}\ \underline{\textbf{then}}\ \mathbf{b{:=}tt}\ \underline{\textbf{else}}\ \mathbf{b{:=}ff}\ \underline{\textbf{fi}}$

Thus a statement of the form $\underline{\textbf{if}}\ \mathbf{b}\ \underline{\textbf{then}}\ \mathbf{S_1}\ \underline{\textbf{else}}\ \mathbf{S_2}\ \underline{\textbf{fi}}$ where **b** is a boolean variable becomes:

$\mathbf{\{b{=}tt \lor b{=}ff\};}\ \underline{\textbf{if}}\ \mathbf{b{=}tt}\ \underline{\textbf{then}}\ \mathbf{S_1}\ \underline{\textbf{else}}\ \mathbf{S_2}\ \underline{\textbf{fi}}$.

Note that expression brackets may be nested to any depth, for example:

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

may be represented as:

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

Defn: Function calls: The definitional transformation of a function call will replace the function call by a call to a procedure which assigns the value returned by the function to a variable. This variable then replaces the function call in the expression. Several calls in one expression are replaced by the same number of procedure calls and new variables. Boolean functions treated as functions which return one of the values **tt** or **ff** (representing true and false). So a boolean function call is replaced by a formula (**b**=**tt**) where **b** is a new local variable. The statement in which the function call appeared is preceded by a procedure call which sets **b** to **tt** or **ff**, depending on the result of the corresponding boolean function.

For example, the function call:

$\mathbf{a{:=}F(x){+}F(y)}\ .\ \underline{\textbf{where}}$

$\underline{\textbf{funct}}\ \mathbf{F(x)} \equiv\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathbf{t_1}\ \underline{\textbf{else}}\ \mathbf{t_2}\ \underline{\textbf{fi}}.$

is interpreted:

$\underline{\textbf{var}}$ $\mathbf{r_1,r_2}$;
$\mathbf{F(x); r_1:=r; F(y); r_2:=r;}$
$\mathbf{a:=r_1+r_2.}$ $\underline{\textbf{where}}$
$\underline{\textbf{proc}}$ $\mathbf{F(x) \equiv}$ $\underline{\textbf{if}}$ $\mathbf{B}$ $\underline{\textbf{then}}$ $\mathbf{r:=t_1}$ $\underline{\textbf{else}}$ $\mathbf{r:=t_2}$ $\underline{\textbf{fi}}$.

The statement:

$\mathbf{a:=} \lceil\underline{\textbf{while}}\ \mathbf{B(x)}\ \underline{\textbf{do}}\ \mathbf{x:=F(x)}\ \underline{\textbf{od}};\ \mathbf{x+c}\rfloor$
$\underline{\textbf{funct}}\ \mathbf{B(x) \equiv \lceil S;\ x>y\rfloor}$.
$\underline{\textbf{funct}}\ \mathbf{F(x) \equiv}\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathbf{t_1}\ \underline{\textbf{else}}\ \mathbf{t_2}\ \underline{\textbf{fi}}$.

is interpreted:

$\underline{\textbf{do}}\ \mathbf{B(x);}\ \underline{\textbf{if}}\ \mathbf{r=ff}\ \underline{\textbf{then}}\ \underline{\textbf{exit}}\ \underline{\textbf{fi}};$
$\quad\mathbf{F(x);\ x:=r}\ \underline{\textbf{od}};$
$\mathbf{a:=x+c.}\ \underline{\textbf{where}}$
$\underline{\textbf{proc}}\ \mathbf{B(x) \equiv\ S;}\ \underline{\textbf{if}}\ \mathbf{x>y}\ \underline{\textbf{then}}\ \mathbf{r:=tt}\ \underline{\textbf{else}}\ \mathbf{r:=ff}\ \underline{\textbf{fi}},$
$\underline{\textbf{proc}}\ \mathbf{F(x) \equiv}\ \underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathbf{r:=t_1}\ \underline{\textbf{else}}\ \mathbf{r:=t_2}\ \underline{\textbf{fi}}$.

More formally we define:

Defn: Generalised expressions and generalised conditions are defined as follows: If **S** is any statement (which may include generalised expressions and generalised conditions) and
$\mathbf{E}_1$, $\mathbf{E}_2$, ... are generalised expressions and
$\mathbf{B}_1$, $\mathbf{B}_2$, ... are generalised conditions and
t is a term including variables $\mathbf{x}_1, \ldots, \mathbf{x}_n$ ($\mathbf{n \geqslant 0}$) and
Q is a formula including free variables $\mathbf{x}_1, \ldots, \mathbf{x}_n$ ($\mathbf{n \geqslant 0}$) and
$\mathbf{Q}_1$, $\mathbf{Q}_2$, ... are formulae and
$\mathbf{F(x_1,...,x_n)}$ is a function and
$\mathbf{B(x_1,...,x_n)}$ is a Boolean function (see below) then:

(i) $\mathbf{t[E_1,...,E_n/x_1,...,x_n]}$
(ii) $\lceil\mathbf{S;E_1}\rfloor$
(iii) $\underline{\textbf{if}}\ \mathbf{B_1}\ \underline{\textbf{then}}\ \mathbf{E_1}\ \underline{\textbf{else}}\ \mathbf{E_2}\ \underline{\textbf{fi}}$
(iv) $\underline{\textbf{if}}\ \mathbf{B_1} \rightarrow \mathbf{E_1} \ \square \ldots \square\ \mathbf{B}_n \rightarrow \mathbf{S}_n\ \underline{\textbf{fi}}$
(v) $\mathbf{F(E_1,...,E_n)}$

are all generalised expressions and:

(i) $\mathbf{Q[E_1,...,E_n/x_1,...,x_n]}$
(ii) $\mathbf{Q[B_1,...,B_n/Q_1,...,Q_n]}$
(iii) $\lceil \mathbf{S;B_1} \rfloor$
(iv) $\mathbf{B(E_1,...,E_n)}$
are all generalised conditions. (Note that setting **n=0** shows that any term is also a generalised expression and any formula is also a generalised condition).

<u>Defn:</u> Functions and Boolean Functions:
A function $\mathbf{F(x)}$ is defined by:
<u>**funct**</u> $\mathbf{F(x) \equiv E.}$ where **E** is a generalised expression.
A Boolean function $\mathbf{B(x)}$ is defined by:
<u>**funct**</u> $\mathbf{B(x) \equiv B.}$ where **B** is a generalised condition.

<u>Interpretation of Generalised Expressions and Generalised Conditions</u>

These are interpreted by "interpretation functions" **IE**, **IC** and **IS** where:
If **E** is a generalised expression and **r** a variable which does not occur in **E** then **IE(E,r)** is a statement which sets **r** to the value which results from evaluating the expression.
If **B** is a generalised condition and **r** a variable which does not occur in **B** then **IC(B,r)** is a statement which sets **r** to one of the special values **tt** or **ff** according to whether **B** evaluates to true or false.
If **S** is any statement which includes generalised expressions (in place of terms) and/or generalised conditions (in place of formulae) then **IS(S)** is the statement formed by replacing each assignment **a:=E** (where **E** is a generalised expression) by **IE(E,a)** and replacing each statement which uses a generalised condition by an equivalent one which uses **IC** to evaluate the condition. The definitions are:

<u>Defn:</u> **IE(E,r)** is defined by:
(i) $\mathbf{IE(t[E_1,...,E_n/x_1,...,x_n],\ r)} =_{DF}$
<u>**var**</u> $\mathbf{x_1,x_2,...,x_n}$
$\mathbf{IE(E_1,x_1);\ IE(E_2,x_2);\ ...\ IE(E_n,x_n);}$
r:=t.
where the $\mathbf{x}_i$ are local variables which may have to be implemented using stacks.
(So for any ordinary term **t**: **IE(t,r) = r:=t**)
(ii) $\mathbf{IE(\lceil S;E_1 \rfloor,\ r)} =_{DF}$ $\mathbf{IS(S);\ IE(E_1,r)}$

(iii) $\mathbf{IE}(\underline{\mathbf{if}}\ \mathbf{B}_1\ \underline{\mathbf{then}}\ \mathbf{E}_1\ \underline{\mathbf{else}}\ \mathbf{E}_2\ \underline{\mathbf{fi}}, \mathbf{r}) =_{DF}$

$\mathbf{IC}(\mathbf{B}_1,\mathbf{r});\ \underline{\mathbf{if}}\ \mathbf{r}{=}\mathbf{tt}\ \underline{\mathbf{then}}\ \mathbf{IE}(\mathbf{E}_1,\mathbf{r})\ \underline{\mathbf{else}}\ \mathbf{IE}(\mathbf{E}_2,\mathbf{r})\ \underline{\mathbf{fi}}$

(iv) $\mathbf{IE}(\underline{\mathbf{if}}\ \mathbf{B}_1 \ \rightarrow \mathbf{E}_1 \ \Box \ldots \Box\ \mathbf{B}_n \ \rightarrow \mathbf{S}_n\ \underline{\mathbf{fi}}, \mathbf{r}) =_{DF}$

$\underline{\mathbf{var}}\ \mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n$

$\mathbf{IC}(\mathbf{B}_1,\mathbf{x}_1);\ \mathbf{IC}(\mathbf{B}_2,\mathbf{x}_2);\ \ldots\ \mathbf{IC}(\mathbf{B}_n,\mathbf{x}_n);$

$\underline{\mathbf{if}}\ \mathbf{x}_1 = \mathbf{tt}\ \rightarrow\ \mathbf{IE}(\mathbf{E}_1,\mathbf{r})\ \Box\ \ldots\ \Box\ \mathbf{x}_n = \mathbf{tt}\ \rightarrow\ \mathbf{IE}(\mathbf{E}_n,\mathbf{r})\ \underline{\mathbf{fi}}.$

(v) $\mathbf{IE}(\mathbf{F}(\mathbf{E}_1,\ldots,\mathbf{E}_n), \mathbf{r}) =_{DF}$

$\underline{\mathbf{var}}\ \mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n$

$\mathbf{IE}(\mathbf{E}_1,\mathbf{x}_1);\ \mathbf{IE}(\mathbf{E}_2,\mathbf{x}_2);\ \ldots\ \mathbf{IE}(\mathbf{E}_n,\mathbf{x}_n);$

$\mathbf{F}_r(\mathbf{x}_1,\ldots,\mathbf{x}_n).$

where $\mathbf{F}_r$ is the "procedural equivalent" of the function **F** which sets the global variable **r** to the result. ie if **F** is defined $\underline{\mathbf{funct}}\ \mathbf{F}(\mathbf{x}) \equiv \mathbf{E}.$ then $\mathbf{F}_r$ is defined as:
$\underline{\mathbf{proc}}\ \mathbf{F}_r(\mathbf{x}) \equiv\ \mathbf{IE}(\mathbf{E},\mathbf{r}).$

Note, however, that if there are occurrences of $\mathbf{F}(\mathbf{x})$ which are interpreted with different variables then we must choose one variable **r** for $\mathbf{F}_r$ and for the other variables define:

$\mathbf{IE}(\mathbf{F}(\mathbf{E}_1,\ldots,\mathbf{E}_n), \mathbf{s}) =_{DF}\ \underline{\mathbf{var}}\ \mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n$

$\mathbf{IE}(\mathbf{E}_1,\mathbf{x}_1);\ \mathbf{IE}(\mathbf{E}_2,\mathbf{x}_2);\ \ldots\ \mathbf{IE}(\mathbf{E}_n,\mathbf{x}_n);$

$\mathbf{F}_r(\mathbf{x}_1,\ldots,\mathbf{x}_n);\ \mathbf{s}{:=}\mathbf{r}.$

$\mathbf{IC}(\mathbf{B},\mathbf{r})$ is defined by:

(i) $\mathbf{IC}(\mathbf{Q}[\mathbf{E}_1,\ldots,\mathbf{E}_n/\mathbf{x}_1,\ldots,\mathbf{x}_n], \mathbf{r}) =_{DF}$

$\underline{\mathbf{var}}\ \mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n$

$\mathbf{IE}(\mathbf{E}_1,\mathbf{x}_1);\ \mathbf{IE}(\mathbf{E}_2,\mathbf{x}_2);\ \ldots\ \mathbf{IE}(\mathbf{E}_n,\mathbf{x}_n);$

$\underline{\mathbf{if}}\ \mathbf{Q}\ \underline{\mathbf{then}}\ \mathbf{r}{:=}\mathbf{tt}\ \underline{\mathbf{else}}\ \mathbf{r}{:=}\mathbf{ff}\ \underline{\mathbf{fi}}.$

(So for any ordinary formula **Q**: $\mathbf{IC}(\mathbf{Q},\mathbf{r}) = \underline{\mathbf{if}}\ \mathbf{Q}\ \underline{\mathbf{then}}\ \mathbf{r}{:=}\mathbf{tt}\ \underline{\mathbf{else}}\ \mathbf{r}{:=}\mathbf{ff}\ \underline{\mathbf{fi}}$)

(ii) $\mathbf{IC}(\mathbf{Q}[\mathbf{B}_1,\ldots,\mathbf{B}_n/\mathbf{Q}_1,\ldots,\mathbf{Q}_n], \mathbf{r}) =_{DF}$

$\underline{\mathbf{var}}\ \mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n$

$\mathbf{IC}(\mathbf{B}_1,\mathbf{x}_1);\ \mathbf{IC}(\mathbf{B}_2,\mathbf{x}_2);\ \ldots\ \mathbf{IC}(\mathbf{B}_n,\mathbf{x}_n);$

$\underline{\mathbf{if}}\ \mathbf{Q}\ \underline{\mathbf{then}}\ \mathbf{r}{:=}\mathbf{tt}\ \underline{\mathbf{else}}\ \mathbf{r}{:=}\mathbf{ff}\ \underline{\mathbf{fi}}.$

(iii) $\mathbf{IC}(\ulcorner \mathbf{S};\mathbf{B}_1 \lrcorner, \mathbf{r}) =_{DF}\ \mathbf{IS}(\mathbf{S});\ \mathbf{IC}(\mathbf{B}_1,\mathbf{r})$

(iv) $\mathbf{IC}(\mathbf{B}(\mathbf{E}_1,\ldots,\mathbf{E}_n/\mathbf{x}_1,\ldots,\mathbf{x}_n), \mathbf{r}) =_{DF}$

$\underline{\mathbf{var}}\ \mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n$

$\mathbf{IC}(\mathbf{B}_1,\mathbf{x}_1);\ \mathbf{IC}(\mathbf{B}_2,\mathbf{x}_2);\ \ldots\ \mathbf{IC}(\mathbf{B}_n,\mathbf{x}_n);$

$\mathbf{B}_r(\mathbf{x}_1,\ldots,\mathbf{x}_n).$

where $\mathbf{B}_r$ is the "procedural equivalent" of the boolean function **B** which sets the global variable **r** to the result. ie if **B** is defined $\underline{\mathbf{funct}}\ \mathbf{B}(\mathbf{x}) \equiv \mathbf{B}.$ then $\mathbf{B}_r$ is defined:

proc $\mathbf{B}_r(\mathbf{x}) \equiv$ **IC(B,r).**

IS(S) is defined by: (where **r** is a new variable)

(i) **IS(S)** $=_{DF}$ **S**

if **S** contains no generalised expression or generalised condition.

(ii) **IS(a:=E)** $=_{DF}$ **IE(E,a)**

(ii) **IS(if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi)** $=_{DF}$

IE(B,r); if r=tt then IS($\mathbf{S}_1$**) else IS(**$\mathbf{S}_2$**) fi**

(iii) **IS(if** $\mathbf{B}_1 \rightarrow \mathbf{S}_1 \square \dots \square \mathbf{B}_n \rightarrow \mathbf{S}_n$ **fi)** $=_{DF}$

var $\mathbf{x}_1,\mathbf{x}_2,\dots,\mathbf{x}_n$

IC($\mathbf{B}_1,\mathbf{x}_1$**); IC(**$\mathbf{B}_2,\mathbf{x}_2$**); ... IC(**$\mathbf{B}_n,\mathbf{x}_n$**);**

if $\mathbf{x}_1 = \mathbf{tt} \rightarrow$ **IS(**$\mathbf{S}_1$**)** $\square \dots \square$ $\mathbf{x}_n = \mathbf{tt} \rightarrow$ **IS(**$\mathbf{S}_n$**) fi.**

(iv) **IS(while B do S od)** $=_{DF}$

do IC(B,r); if r=ff then exit fi;

IS(S) od

(v) **IS(do** $\mathbf{B}_1 \rightarrow \mathbf{S}_1 \square \dots \square \mathbf{B}_n \rightarrow \mathbf{S}_n$ **fi)** $=_{DF}$

do var $\mathbf{x}_1,\mathbf{x}_2,\dots,\mathbf{x}_n$

IC($\mathbf{B}_1,\mathbf{x}_1$**); IC(**$\mathbf{B}_2,\mathbf{x}_2$**); ... IC(**$\mathbf{B}_n,\mathbf{x}_n$**);**

if $\neg(\mathbf{x}_1 = \mathbf{tt} \vee \dots \vee \mathbf{x}_n = \mathbf{tt})$ **then exit fi;**

if $\mathbf{x}_1 = \mathbf{tt} \rightarrow$ **IS(**$\mathbf{S}_1$**)** $\square \dots \square$ $\mathbf{x}_n = \mathbf{tt} \rightarrow$ **IS(**$\mathbf{S}_n$**) fi. od**

All other statements are defined by interpreting their components.

Transformations of Generalised Expressions

These can be derived from similar transformations of procedures and other statements, for example unfolding (replacing a function call by a copy of the body) and folding. They should be used with care because not all transformations of statements will still be valid when some of their terms are replaced by function calls. For example:

a:=t+t (where **t** is a term) may be replaced by **a:=2.t** but:

a:=G()+G() where

funct G() $\equiv$

if true $\rightarrow$ **0**

$\square$ **true** $\rightarrow$ **1 fi**

is only refined by **a:=2.G()** and not equivalent to it. This is because **a:=G()+G()** is equivalent to:

$\underline{\mathbf{var}}$ $\mathbf{r_1,r_2}$: **G**; $\mathbf{r_1}$**:=r**; **G**; $\mathbf{r_2}$**:=r**; **a:=**$\mathbf{r_1+r_2}$. **where**

proc **G** $\equiv$

if **true** $\rightarrow$ **r:=0**

$\Box$ **true** $\rightarrow$ **r:=1** **fi**

while **a:=2.G()** is equivalent to:

var $\mathbf{r_1}$: **G**; $\mathbf{r_1}$**:=r**; **a:=2.**$\mathbf{r_1}$. **where**

proc **G** $\equiv$

if **true** $\rightarrow$ **r:=0**

$\Box$ **true** $\rightarrow$ **r:=1** **fi**

The first case is equivalent to: $\mathbf{a{:=}a'.(a' = 0 \lor a' = 1 \lor a' = 2)}$ and this may be refined by (but is not equivalent to): $\mathbf{a{:=}a'.(a' = 0 \lor a' = 2)}$, which is equivalent to the second case.

[Bauer & Wossner 80] use the notation $\mathbf{(0 \Box 1)}$ for such a "nondeterministic expression". The generalised expression $(\mathbf{E}_1 \Box \ldots \Box \mathbf{E}_n)$, which represents in our notation **if** **true** $\rightarrow \mathbf{E}_1 \Box \ldots \Box$ **true** $\rightarrow \mathbf{E}_n$ **fi**, will (nondeterministically) take one of the values $\mathbf{E}_1, \ldots, \mathbf{E}_n$.

To apply one of the transformations we have derived so far to a statement containing generalised expressions and generalised conditions first transform the statement to an ordinary one (using the interpretation functions **IS**, **IC** and **IE**), then apply the transformation (making sure that it still applies), and then apply the interpretation functions "in reverse" (if possible) to get a transformed statement containing generalised expressions and generalised conditions.

Tail-Recursion

We have the following transformation for tail-recursive procedures:

proc **F(x)** $\equiv$ **if** **B** **then** $\mathbf{S_1}$; **F(g(x))**

else $\mathbf{S_2}$ **fi**.

$\approx$

proc **F(x)** $\equiv$ **while** **B** **do** $\mathbf{S_1}$; **x:=g(x)** **od**; $\mathbf{S_2}$.

Note that $\mathbf{S_1}$ and $\mathbf{S_2}$ may contain further calls of **F(x)**.

This follows from replacing the parameter by a stack:

proc **F(x)** $\equiv$ **S:=**$\langle\rangle$; **F**. **where**

F $\equiv$ (**if** **B** **then** $\mathbf{S_1}$ **x:=g(x)**; **F** **fi**; $\mathbf{S_2}$)[**S**$\leftarrow$**x**; **x:=h(x)**; **F**; **x**$\leftarrow$**S** / **F(h(x))**].

Note that we don't require **F** to preserve the value of **x** so **x** does not have to be stacked and restored for the final call **F(g(x))** which has therefore been replaced by **x:=g(x);F**.

By the tail-recursion result of the previous Chapter this becomes:

$\underline{\textbf{proc}}$ $\mathbf{F(x)} \equiv$ $\mathbf{S{:=}\langle\rangle;\ F.}$ $\underline{\textbf{where}}$

$\mathbf{F} \equiv$ $(\underline{\textbf{while}}$ $\mathbf{B}$ $\underline{\textbf{do}}$ $\mathbf{S_1\ x{:=}g(x)}$ $\underline{\textbf{od}}$; $\mathbf{S_2})[\mathbf{S{\leftarrow}x;\ x{:=}h(x);\ F;\ x{\leftarrow}S\ /\ F(h(x))}]$.

Replace stack **S** by a parameter again and copy in to get:

$\underline{\textbf{proc}}$ $\mathbf{F(x)} \equiv$ $\underline{\textbf{while}}$ $\mathbf{B}$ $\underline{\textbf{do}}$ $\mathbf{S_1;\ x{:=}g(x)}$ $\underline{\textbf{od}}$; $\mathbf{S_2}$.

There is a similar result for functions:

$\underline{\textbf{funct}}$ $\mathbf{F(x)} \equiv$ $\underline{\textbf{if}}$ $\mathbf{B}$ $\underline{\textbf{then}}\ulcorner \mathbf{S_1;\ F(g(x))}\lrcorner$

$\underline{\textbf{else}}$ $\mathbf{E}$ $\underline{\textbf{fi}}$.

where **E** is any general expression. This is equivalent to:

$\underline{\textbf{funct}}$ $\mathbf{F(x)} \equiv$ $\ulcorner\underline{\textbf{while}}$ $\mathbf{B}$ $\underline{\textbf{do}}$ $\mathbf{S_1;\ x{:=}g(x)}$ $\underline{\textbf{od}}$; $\mathbf{E}\lrcorner$.

The proof follows from transforming the functions to procedures.

<u>Example:</u> Fibonacci Numbers:

The sequence of Fibonacci numbers: **0,1,1,2,3,5,8,13,21,34,...**$\mathbf{F_n}$**,...** is defined by:

$\mathbf{F_1 = 0}$

$\mathbf{F_2 = 1}$

$\mathbf{F_{n+2} = F_{n+1}+F_n}$ for $\mathbf{n>0}$

A function to evaluate $\mathbf{F_n}$ given **n** is:

$\underline{\textbf{funct}}$ $\mathbf{fib(n)} \equiv$ $\mathbf{F_n}$.

To transform this into a program which is not defined in terms of $\mathbf{F_n}$ we first take out the cases **n=1** and **n=2**:

$\underline{\textbf{funct}}$ $\mathbf{fib(n)} \equiv$ $\underline{\textbf{if}}$ $\mathbf{n{=}1}$ $\underline{\textbf{then}}\ulcorner\mathbf{\{n{=}1\};\ F_n}\lrcorner$

$\underline{\textbf{elsf}}$ $\mathbf{n{=}2}$ $\underline{\textbf{then}}\ulcorner\mathbf{\{n{=}2\};\ F_n}\lrcorner$

$\underline{\textbf{else}}\ulcorner\mathbf{\{n>2\};\ F_n}\lrcorner$ $\underline{\textbf{fi}}$.

Now $\ulcorner\mathbf{\{n{=}1\};\ F_n}\lrcorner \approx \ulcorner\mathbf{\{n{=}1\};\ 0}\lrcorner$

and $\ulcorner\mathbf{\{n{=}2\};\ F_n}\lrcorner \approx \ulcorner\mathbf{\{n{=}2\};\ 1}\lrcorner$ from the definition of $\mathbf{F_n}$, while:

$\ulcorner\mathbf{\{n>2\};\ F_n}\lrcorner \approx \ulcorner\mathbf{\{n>2\};\ F_{n-1}+F_{n-2}}\lrcorner$

We get:

```
funct fib(n) ≡ if n=1
            then 0
       elsf n=2 then 1
          else F_{n−1}+F_{n−2} fi.
```

Finally we can replace $\mathbf{F}_{n-1}+\mathbf{F}_{n-2}$ by **fib(n−1)+fib(n−2)** since $1 \leqslant n-1 < n$ and $1 \leqslant n-2 < n$ (this follows from the theorem on recursive implementation of a specification). The result is:

```
funct fib(n) ≡ if n=1
            then 0
       elsf n=2 then 1
           else fib(n−1)+fib(n−2) fi.
```

Each call of this function with argument >2 causes two further calls, it is easy to see therefore that **fib(n)** results in $2^{n-1}-1$ function calls (for $n \geqslant 1$) so the function as it stands is hopelessly inefficient. Noting however that **fib(n)** is defined in terms of **fib(n−1)** and **fib(n−2)** (for $n>2$) prompts us to try the definition:

```
funct g(n) ≡ ⟨fib(n), fib(n−1)⟩.
```

for $n>1$.

Hence **g(2)**= ⟨**fib(2), fib(1)**⟩ = ⟨**1,0**⟩.

Unfolding **g(n+1)** (for $n>1$) gives:

g(n+1)= ⟨**fib(n+1), fib(n)**⟩
 = ⟨**fib(n)+fib(n−1), fib(n)**⟩ since $n+1>2$.

This is defined in terms of **fib(n)** and **fib(n−1)** so we can write it in terms of **g(n)**:
 = ⟨**g(n)[1]+g(n)[2], g(n)[1]**⟩
 = ⌈**beg** ⟨**a,b**⟩**:=g(n);** ⟨**a+b, a**⟩ **end**⌋

Hence we can prove (by the recursive implementation theorem again) that **g** is equivalent to:

```
funct g(n) ≡ if n=2 then ⟨1,0⟩
        else ⌈beg ⟨a,b⟩:=g(n−1); ⟨a+b, a⟩ end⌋ fi.
```

A procedural version of this, where **r:=g(n)** $\approx$ **G(n)** is:

```
proc G(n) ≡ if n=2 then r:=⟨1,0⟩
        else beg ⟨a,b⟩:=g(n−1); r:=⟨a+b, a⟩ end fi.
```

Our recursion-removal techniques can now be applied to get:

<u>proc</u> G(n) $\equiv$ **n_0:=n; r:=⟨1,0⟩;**
<u>for</u> n:=2 <u>to</u> n_0 <u>step</u> 1 <u>do</u> r:=⟨r[1]+r[2], r[1]⟩ <u>od</u>.

Hence we can write **fib** in the linear form: (replacing **r** by **a** and **b** where **r**= ⟨**a**,**b**⟩):

```
funct fib(n) ≡
  if n=0 then 0
  elsf n=1 then 1
        else ⌈ a:=1; b:=0;
           for i:=2 to n step 1 do
             ⟨a,b⟩:=⟨a+b,a⟩ od;
           a⌋ fi.
```

However, there is a still more efficient way of computing Fibonacci numbers. From the original definition it is possible to prove the equations (for **j>2**):

$\mathbf{F}_{2j} = \mathbf{F}_j^2+\mathbf{F}_{j+1}^2$

$\mathbf{F}_{2j+1} = (\mathbf{2}.\mathbf{F}_j+\mathbf{F}_{j+1}).\mathbf{F}_{j+1}$ and $\mathbf{F}_{2j-1} = (\mathbf{2}.\mathbf{F}_{j+1}-\mathbf{F}_j).\mathbf{F}_j$

Dijkstra gives proofs for these equations in [Dijkstra 79] where he challenges "proponents of program transformations" to transform the linear form of the Fibonacci numbers program into a logarithmic form which the equations suggest should be possible. We have already shown that the linear form given above is equivalent to the specification **<u>funct</u> fib(n)** $\equiv \mathbf{F}_n$. We shall now take up Dijkstra's challenge and transform this into a logarithmic order function using transformation we have already developed.

From **<u>funct</u> fib(n)** $\equiv$ $\mathbf{F}_n$. we can use the proof of the equations above to transform this directly to:

```
funct fib(n) ≡
    ⌈x:=F_n.(∀j∈ℕ. ((n=2.j ⇒ (F_n =F_j²+F_{j+1}²))
               ∧ (n=2.j+1 ⇒ (F_n = (2.F_j+F_{j+1}).F_{j+1})));
     x⌋.
```

From this we can derive the following recursive function (using the theorem for the recursive implementation of specifications):

```
funct fib2(j) ≡ if j=1 then 0
          elsf j=2 then 1
      elsf even(j) then fib2(j/2)^2+fib2(j/2+1)^2
            else (2.fib2((j−1)/2)+fib2((j−1)/2+1)).fib2((j−1)/2+1) fi.
```

Define the expression $\langle \mathbf{a} \div \mathbf{b} \rangle$ by: $\langle \mathbf{a} \div \mathbf{b} \rangle =_{DF} \langle (\mathbf{a\ div\ b}), (\mathbf{a\ mod\ b}) \rangle$.
Then if we let $\langle \mathbf{q,r} \rangle = \langle \mathbf{j} \div \mathbf{2} \rangle$ we have:

$$\mathbf{even(j)} \iff (\mathbf{q=j/2} \wedge \mathbf{r=0}) \text{ and } \mathbf{odd(j)} \iff (\mathbf{q=(j-1)/2} \wedge \mathbf{r=1}).$$

So we can write $\mathbf{fib}_2$ as:

```
funct fib2(j) ≡ if j=1 then 0
     elsf j=2 then 1
        else ⌈⟨q,r⟩:=⟨j÷2⟩;
          if r=0
          then fib2(q)^2+fib2(q+1)^2
          else (2.fib2(q)+fib2(q+1)).fib2(q+1) fi⌋fi.
```

Here we see that $\mathbf{fib}_2\mathbf{(j)}$ (for $\mathbf{j>2}$) is defined in terms of $\mathbf{fib}_2\mathbf{(q)}$ and $\mathbf{fib}_2\mathbf{(q+1)}$. This suggests that we define a new function $\mathbf{G}_2\mathbf{(j)}$ to be (for $\mathbf{j \geqslant 2}$):

```
funct G2(j) ≡ ⟨fib2(j), fib2(j−1)⟩.
```

Taking out the cases $\mathbf{j=2}$ and $\mathbf{j=3}$ gives:

```
funct G2(j) ≡ if j=2 then ⟨1,0⟩
     elsf j=3 then ⟨1,1⟩
        else ⌈{j>2}; ⟨fib2(j),fib2(j−1)⟩⌋fi.
```

Unfolding the calls to $\mathbf{fib}_2$ gives:

```
funct G2(j) ≡ if j=2 then ⟨1,0⟩
     elsf j=3 then ⟨1,1⟩
        else ⟨⌈⟨q,r⟩:=⟨j÷2⟩;
           if r=0
            then fib2(q)^2+fib2(q+1)^2
            else (2.fib2(q)+fib2(q+1)).fib2(q+1) fi⌋,
          ⌈⟨q,r⟩:=⟨(j−1)÷2⟩;
           if r=0
            then fib2(q)^2+fib2(q+1)^2
            else (2.fib2(q)+fib2(q+1)).fib2(q+1) fi⌋⟩ fi.
```

If $\langle \mathbf{q,r} \rangle = \langle \mathbf{j \div 2} \rangle$ and $\langle \mathbf{q_1,r_1} \rangle = \langle \mathbf{(j-1) \div 2} \rangle$ then

$$\mathbf{r=0} \Rightarrow (\mathbf{q_1 = q} \ \wedge \ \mathbf{r_1 = 1}) \text{ and } \mathbf{r=1} \Rightarrow (\mathbf{q_1 = q+1} \ \wedge \ \mathbf{r_1 = 0}).$$

So we may replace the outer **else** clause by:

$\lceil \langle \mathbf{q,r} \rangle := \langle \mathbf{j \div 2} \rangle$; $\underline{\textbf{if}}$ $\mathbf{r=0}$
    $\underline{\textbf{then}}$ $\langle \mathbf{fib_2(q)^2 + fib_2(q+1)^2,\ (2.fib_2(q) + fib_2(q+1)).fib_2(q+1)} \rangle$
    $\underline{\textbf{else}}$ $\langle \mathbf{(2.fib_2(q-1) + fib_2(q)).fib_2(q),\ fib_2(q)^2 + fib_2(q+1)^2} \rangle$ $\underline{\textbf{fi}} \rfloor$

$\mathbf{fib_2(q+1) = fib(q+1) = fib(q) + fib(q-1) = fib_2(q) + fib_2(q-1)}$
since **fib** and $\mathbf{fib_2}$ are equivalent.

Now $\mathbf{G_2(j)}$ is defined in terms of $\mathbf{fib_2(q)}$ and $\mathbf{fib_2(q-1)}$ (for $\mathbf{j>3}$) so we can write it in terms of $\mathbf{G_2(q)}$:

$\underline{\textbf{funct}}$ $\mathbf{G_2(j)} \equiv$ $\underline{\textbf{if}}$ $\mathbf{j=2}$ $\underline{\textbf{then}}$ $\langle \mathbf{1,0} \rangle$
  $\underline{\textbf{elsf}}$ $\mathbf{j=3}$ $\underline{\textbf{then}}$ $\langle \mathbf{1,1} \rangle$
    $\underline{\textbf{else}}$ $\lceil \langle \mathbf{q,r} \rangle := \langle \mathbf{j \div 2} \rangle$;
      $\langle \mathbf{a,b} \rangle := \mathbf{G_2(q)}$;
      $\underline{\textbf{if}}$ $\mathbf{r=0}$ $\underline{\textbf{then}}$ $\langle \mathbf{a^2 + (a+b)^2,\ (2.a+a+b).(a+b)} \rangle$
        $\underline{\textbf{else}}$ $\langle \mathbf{(2.b+a).a,\ b^2 + a^2} \rangle$ $\underline{\textbf{fi}} \rfloor \underline{\textbf{fi}}$.

Since the argument is halved for each recursive call $\mathbf{G_2(j)}$ only takes $\mathbf{log(j)}$ time.

To transform this to an iterative version we will use the standard method (a protocol stack). Convert to the procedural equivalent which sets $\langle \mathbf{a,b} \rangle$ to $\mathbf{G_2(j)}$:

$\underline{\textbf{proc}}$ $\mathbf{H(j)} \equiv$
  $\underline{\textbf{if}}$ $\mathbf{j=2}$ $\underline{\textbf{then}}$ $\mathbf{a:=1;\ b:=0}$
  $\underline{\textbf{elsf}}$ $\mathbf{j=3}$ $\underline{\textbf{then}}$ $\mathbf{a:=1;\ b:=1}$
    $\underline{\textbf{else}}$ $\langle \mathbf{q,r} \rangle := \langle \mathbf{j \div 2} \rangle$;
      $\mathbf{H(q)}$;
      $\underline{\textbf{if}}$ $\mathbf{r=0}$ $\underline{\textbf{then}}$ $\langle \mathbf{a,b} \rangle := \langle \mathbf{a^2 + (a+b)^2,\ (3.a+b).(a+b)} \rangle$
        $\underline{\textbf{else}}$ $\langle \mathbf{a,b} \rangle := \langle \mathbf{(2.b+a).a,\ b^2 + a^2} \rangle$ $\underline{\textbf{fi}}$ $\underline{\textbf{fi}}$.

Note that the only variable whose value is required after the inner call is **r** and this has the value **0** or **1** so it can be stored on a stack which may be implemented by the binary representation of an integer **s**. If we initialise **s** to **1** then we can tell that the stack is empty (when $\mathbf{s=1}$) which is true only for the outermost call. We get the iterative form:

```
proc H(j) ≡  s:=1;
        while j>3 do ⟨j,r⟩:=⟨j÷2⟩; s:=2.s+r od;
        a:=1; b:=j−2;
        while s>1 do
          ⟨s,r⟩:=⟨s÷2⟩;
          if r=0 then ⟨a,b⟩:=⟨a²+(a+b)², (3.a+b).(a+b)⟩
              else ⟨a,b⟩:=⟨(2.b+a).a, b²+a²⟩ fi od.
```

The first loop sets **j** to the most significant two bits of the binary representation of its original value (so **j=2** or **3** after the first loop). The other bits are reversed, a **1** is added in front and the result put into **s**. The second loop reads the bits of **s** from the least significant upwards up to but not including the initial **1**.

This leads to the following iterative function for **fib(n)** in time **O(log(n))**:

```
funct fib(n) ≡
  if n=1 then 0
     else ⌈ s:=1;
          while n>3 do ⟨n,r⟩:=⟨n÷2⟩; s:=2.s+r od;
            a:=1; b:=n−2;
            while s>1 do
              ⟨s,r⟩:=⟨s÷2⟩;
              if r=0 then ⟨a,b⟩:=⟨a²+(a+b)², (3.a+b).(a+b)⟩
                  else ⟨a,b⟩:=⟨(2.b+a).a, b²+a²⟩ fi od;
          a⌋ fi.
```

Although this seems at first sight to be a long sequence of transformations to derive a simple program, the transformations used will occur in many other problems. With practice several of them can be applied in one step; in fact most of the transformation could be applied automatically by a suitable programming system with a little guidance required from the programmer.

Example of Function Inversion

Consider the modulo function:

funct mod(a,b) ≡ ι**x.**$(\exists$**m. a=m.b+x** $\wedge$ $0 \leqslant x < b)$.

where ι**x.Q** means “the unique **x** such that **Q** holds” ie:

$$r := \iota x.Q \;\approx\; \langle r\rangle.\langle\rangle.(Q[r/x] \wedge \forall x.(Q \Rightarrow x=r))$$

From this we can derive:

```
funct mod(a,b) ≡ if a⩾2.b then mod(mod(a,2.b),b)
      elsf a⩾b then mod(a−b,b)
         else a fi.
```

We have $\mathbf{0} \leqslant \mathbf{mod(a,2.b)} < \mathbf{2.b}$ from the original definition, so unfolding the outer call of **mod** in the first line of the second definition gives:

```
funct mod(a,b) ≡ if a⩾2.b then if mod(a,2.b)⩾b then mod(mod(a,2.b),b)
                    else mod(a,2.b) fi
      elsf a⩾b then mod(a−b,b)
         else a fi.
```

But $\mathbf{b} \leqslant \mathbf{mod(a,2.b)} < \mathbf{2.b}$
so $\mathbf{0} \leqslant \mathbf{mod(a,2.b)-b} < \mathbf{b}$
so we get:

```
funct mod(a,b) ≡ if a⩾2.b then if mod(a,2.b)⩾b then mod(a,2.b)−b
                    else mod(a,2.b) fi
      elsf a⩾b then mod(a−b,b)
         else a fi.
```

From function inversion we get:

```
funct mod(a,b) ≡
 ⌈var r:=a, dd:=b;
 while r⩾dd do r:=r; dd:=2.dd od;
 ⌈var z:=r;
  while ⟨r,dd⟩ ≠ ⟨a,b⟩ do dd:=dd/2;
         z:= if z⩾dd then z−dd
               else z fi od;
  z⌋⌋.
```

This is simplified to:

```
funct mod(a,b) ≡
 ⌈var r:=a, dd:=b;
 while r⩾dd do dd:=2.dd od;
 while dd≠b do dd:=dd/2;
       if r⩾dd then r:=r−dd fi od;
 r⌋.
```

Here we have started from a specification of the **mod** function and used our transformations to derive an efficient program for division of numbers represented in binary notation.

Constant Propogation

A= **while** **B** **do** **c:=E$_1$; x:=E$_2$** **od**

B= **x:=f(x)** **where**
funct **f(x)** $\equiv$ **if** **B** **then** **c:=E$_1$; f(E$_2$)**
else **x** **fi.**

C= **c:=E$_1$;** **while** **B** **do** **x:=E$_2$** **od**

D= **c:=E$_1$; x:=f(x)** **where**
funct **f(x)** $\equiv$ **if** **B** **then** **f(E$_2$)**
else **x** **fi.**

(1) $\Delta \cup \{$**f** not in **B,E$_1$,E$_2$**$\} \vdash$ **A** $\approx$ **B**
(2) $\Delta \cup \{$**f** not in **B,E$_2$**$\} \vdash$ **C** $\approx$ **D**
(3) $\Delta \cup \{$**x:=E$_1$** and **c:=E$_2$** independent, **c** not in **B**,
E$_1$ defined and no side-effects, **B** invariant over **E$_1$**$\} \vdash$ **A**$\leqslant$**C**
(4) $\Delta \cup \{$**x:=E$_1$** and **c:=E$_2$** independent, **f** not in **E$_1$**, **c** not in **B**,
E$_1$ defined and no side-effects, **B** invariant over **E$_1$**$\} \vdash$ **B**$\leqslant$**D**

(1) and (2) follow from theorem for tail-recursion.

(3) follows from the induction rule for iteration. The induction step is:
while **B** **do** **c:=E$_1$; x:=E$_2$** **od**$^{n+1}$
$\approx$ **if** **B** **then** **c:=E$_1$; x:=E$_2$;** **while** **B** **do** **c:=E$_1$; x:=E$_2$** **od**^{n} **fi**
$\leqslant$ **if** **B** **then** **c:=E$_1$; x:=E$_2$; c:=E$_1$;** **while** **B** **do** **x:=E$_2$** **od** **fi**
$\approx$ **if** **B** **then** **c:=E$_1$; c:=E$_1$; x:=E$_2$;** **while** **B** **do** **x:=E$_2$** **od** **fi** by independence.
$\approx$ **if** **B** **then** **c:=E$_1$; x:=E$_2$;** **while** **B** **do** **x:=E$_2$** **od** **fi**
since **E$_1$** is defined and has no side effects.
$\approx$ **c:=E$_1$;** **if** **B** **then** **x:=E$_2$;** **while** **B** **do** **x:=E$_2$** **od** **fi** since **B** is invar over **E$_1$**.
$\approx$ **c:=E$_1$;** **while** **B** **do** **x:=E$_2$** **od** by loop rolling.

The proof of (4) is similar to that of (3) but using the induction rule for iteration.

If $\mathbf{E}_1$ is determinate then we get equivalence for (3) and (4).

Transforming a Cascade Recursion to Linear Recursion

Consider the following two functions:

$$\underline{\textbf{funct}}\ \mathbf{F}(\mathbf{x}) \equiv\ \underline{\textbf{if}}\ \mathbf{P}(\mathbf{x})\ \underline{\textbf{then}}\ \mathbf{G}(\mathbf{x})\ \underline{\textbf{else}}\ \mathbf{F}(\mathbf{F}(\mathbf{H}(\mathbf{x})))\ \underline{\textbf{fi}}.$$

$$\underline{\textbf{funct}}\ \mathbf{F}_1(\mathbf{x}) \equiv\ \underline{\textbf{if}}\ \mathbf{P}(\mathbf{x})\ \underline{\textbf{then}}\ \mathbf{G}(\mathbf{x})\ \underline{\textbf{else}}\ \mathbf{F}_1(\mathbf{G}(\mathbf{H}(\mathbf{x})))\ \underline{\textbf{fi}}.$$

In this section we use program transformations to aid in determining a set of constraints on $\mathbf{P}$, $\mathbf{G}$ and $\mathbf{H}$ such that $\mathbf{F}$ and $\mathbf{F}_1$ are equivalent.

Defn: For each $\mathbf{x}$ define $\mathbf{k}(\mathbf{x}) \in \mathbb{N}$ such that:

$$\mathbf{P}(\mathbf{H}^{k(x)}) \wedge (\forall \mathbf{i},\ \mathbf{0} \leqslant \mathbf{i} < \mathbf{k}(\mathbf{x}).\ \neg \mathbf{P}(\mathbf{H}^{i}(\mathbf{x})))$$

where this exists (it exists for all $\mathbf{x}$ on which $\mathbf{F}$ terminates).

Theorem: Under the condition:

$$(1)\ \forall \mathbf{x}.\ \mathbf{P}(\mathbf{x}) \Rightarrow \mathbf{P}(\mathbf{HG}(\mathbf{x}))$$

we have $\mathbf{F}(\mathbf{x}) = \mathbf{F}_1(\mathbf{x})$ for all $\mathbf{x}$ which satisfy $\mathbf{P}(\mathbf{x}) \vee \mathbf{P}(\mathbf{H}(\mathbf{x}))$.
Proof: Suppose $\mathbf{x}$ satisfies $\mathbf{P}(\mathbf{x}) \vee \mathbf{P}(\mathbf{H}(\mathbf{x}))$, if $\mathbf{P}(\mathbf{x})$ holds then $\mathbf{F}(\mathbf{x}) = \mathbf{F}_1(\mathbf{x}) = \mathbf{G}(\mathbf{x})$ (trivial).
So suppose $\neg \mathbf{P}(\mathbf{x})$ holds, then by the premise we must have $\mathbf{P}(\mathbf{H}(\mathbf{x}))$ holds. By defn of $\mathbf{F}$ we have $\mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{F}(\mathbf{H}(\mathbf{x}))) = \mathbf{F}(\mathbf{G}(\mathbf{H}(\mathbf{x})))$ since $\mathbf{P}(\mathbf{H}(\mathbf{x}))$.
Suppose $\mathbf{P}(\mathbf{GH}(\mathbf{x}))$ holds, then $\mathbf{F}(\mathbf{x}) = \mathbf{GGH}(\mathbf{x})$
By defn of $\mathbf{F}_1$ we have $\mathbf{F}_1(\mathbf{x}) = \mathbf{F}_1(\mathbf{GH}(\mathbf{x})) = \mathbf{GGH}(\mathbf{x})$ as required.

Conversely suppose $\neg \mathbf{P}(\mathbf{GH}(\mathbf{x}))$ holds, then $\mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{F}(\mathbf{H}(\mathbf{GH}(\mathbf{x}))))$.
By (1) $\mathbf{P}(\mathbf{H}(\mathbf{x})) \Rightarrow \mathbf{P}(\mathbf{HGH}(\mathbf{x}))$ so $\mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{GHGH}(\mathbf{x})) = \mathbf{F}((\mathbf{GH})^2(\mathbf{x}))$.
As above, if $\mathbf{P}((\mathbf{GH})^2(\mathbf{x}))$ holds then $\mathbf{F}(\mathbf{x}) = \mathbf{G}(\mathbf{GH})^2(\mathbf{x})$, otherwise $\mathbf{F}((\mathbf{GH})^3(\mathbf{x}))$, etc.

Continuing in this way we see that $\mathbf{F}(\mathbf{x}) = \mathbf{G}(\mathbf{GH})^n(\mathbf{x})$ where $\mathbf{n}$ is the smallest such that $\mathbf{P}((\mathbf{GH})^n(\mathbf{x}))$ holds. But this is the result $\mathbf{F}_1$ gives since if $\neg \mathbf{P}(\mathbf{GH}(\mathbf{x}))$ then $\mathbf{F}_1(\mathbf{x}) = \mathbf{F}_1((\mathbf{GH})^2(\mathbf{x}))$ etc.

Hence $\mathbf{F(x)}=\mathbf{F_1(x)}$ on $\{\mathbf{x}|\mathbf{P(x)} \vee \mathbf{P(H(x))}\}$ as required.

If we assume another condition; that $\mathbf{G}$ and $\mathbf{H}$ be commutative ie:

$$(2)\ \forall\mathbf{x}.\ \mathbf{GH(x)}=\mathbf{HG(x)},$$

then we can prove a more general theorem connecting $\mathbf{F}$ and $\mathbf{F_1}$:

<u>Theorem:</u> Under the conditions (1) and (2) we have:

$$\forall\mathbf{x}.\ \mathbf{F(x)} = \mathbf{F}_1^{k+1}(\mathbf{H}^k(\mathbf{x}))$$

where $\mathbf{k}=\mathbf{k(x)}$ is as defined above. (We assume that $\mathbf{F}$ is defined everywhere).

Proof: By induction on $\mathbf{k}$.

If $\mathbf{k}=\mathbf{0}$ then $\mathbf{P(x)}$ holds and the last theorem gives $\mathbf{F(x)}=\mathbf{F_1(x)}$.

So suppose $\mathbf{k}>\mathbf{0}$ and the theorem holds for all $\mathbf{x}$ st $\mathbf{k(x)}<\mathbf{k}$.

Let $\mathbf{x}$ be st $\mathbf{k(x)}=\mathbf{k}$. $\mathbf{k}>\mathbf{0} \Rightarrow \neg\mathbf{P(x)}$ holds so $\mathbf{F(x)}=\mathbf{F(F(H(x)))}$

$\mathbf{k(H(x))}=\mathbf{k\text{-}1} <\mathbf{k}$ (by definition of $\mathbf{k(x)}$) so by induction hypothesis on $\mathbf{H(x)}$:

$$\mathbf{F(H(x))}=\mathbf{F}_1^{k-1+1}(\mathbf{H}^{k-1}(\mathbf{H(x)})) = \mathbf{F}_1^k(\mathbf{H}^k(\mathbf{x}))$$

Each application of $\mathbf{F_1}$ does a number of applications of $\mathbf{G}$ and $\mathbf{H}$ in some order with one more application of $\mathbf{G}$ than $\mathbf{H}$. The order is immaterial since $\mathbf{G}$ and $\mathbf{H}$ commute.

So we can write $\mathbf{F}_1^k(\mathbf{x})=\mathbf{G}^{k+m}\mathbf{H}^m(\mathbf{x})$ for some $\mathbf{m}\in \mathbb{N}$, so

$\mathbf{F(H(x))}=\mathbf{G}^{k+m}\mathbf{H}^{k+m}(\mathbf{x})$.

Also the last application of $\mathbf{G}$ in the last application of $\mathbf{F_1}$ must have been on something for which $\mathbf{P}$ holds, ie $\mathbf{P}(\mathbf{G}^{k+m-1}\mathbf{H}^{k+m}(\mathbf{x}))$ holds ($\mathbf{k+m-1}\geqslant\mathbf{0}$ since $\mathbf{k}\geqslant\mathbf{1}$).

So by (1) $\mathbf{P}(\mathbf{HG}(\mathbf{G}^{k+m-1}\mathbf{H}^{k+m}(\mathbf{x})))$ holds, ie $\mathbf{P}(\mathbf{H}(\mathbf{G}^{k+m}\mathbf{H}^{k+m}(\mathbf{x})))$ holds. So by applying the last theorem to $\mathbf{G}^{k+m}\mathbf{H}^{k+m}(\mathbf{x})$ we get:

$$\mathbf{F(F(H(x)))} = \mathbf{F}(\mathbf{G}^{k+m}\mathbf{H}^{k+m}(\mathbf{x})) = \mathbf{F_1}(\mathbf{G}^{k+m}\mathbf{H}^{k+m}(\mathbf{x})) = \mathbf{F_1}(\mathbf{F}_1^k(\mathbf{H}^k(\mathbf{x}))) = \mathbf{F}_1^{k+1}(\mathbf{H}^k(\mathbf{x}))$$

Result proved by induction.

Finally, if we add the third condition:

$$(3)\ \forall\mathbf{x}.\ \mathbf{P(G(x))}\Rightarrow\mathbf{P(X)}$$

complete equivalence is obtained:

<u>Theorem:</u> Under conditions (1), (2) and (3) we have $\forall\mathbf{x}.\ \mathbf{F(x)}=\mathbf{F_1(x)}$.

Proof: We can calculate $\mathbf{k(x)}$ iteratively by the function:

<u>funct</u> k(x) ≡ <u>if</u> P(x) <u>then</u> 0 <u>else</u> k(H(x))+1 <u>fi</u>.

$\mathbf{F_1}$ is already in iterative form so we can write an iterative procedure to calculate $\mathbf{F}_1^{k(x)+1}(\mathbf{H}^{k(x)}(\mathbf{x}))$ which is equivalent to $\mathbf{F(x)}$.

First we note $\mathbf{P}(\mathbf{H}^{k(x)}(\mathbf{x}))$ always holds and so $\mathbf{F}_1(\mathbf{H}^{k(x)}(\mathbf{x}))=\mathbf{G}(\mathbf{H}^{k(x)}(\mathbf{x}))$. So

$$\mathbf{F}(\mathbf{x}) = \mathbf{F}_1^{k(x)+1}(\mathbf{H}^{k(x)}(\mathbf{x})) = \mathbf{F}_1^{k(x)}(\mathbf{G}\mathbf{H}^{k(x)}(\mathbf{x}))$$

The following routine sets **x** to $\mathbf{F}_1(\mathbf{x}_0)$ where $\mathbf{x}_0$ is the initial value of **x**:

```
F1 ≡ while ¬P(x) do x:=GH(x) od; x:=G(x)
```

We can use this to give an iterative routine for **F**:

```
Fκκ ≡ κκk:=0;
  while ¬P(x) do x:=H(x); k:=k+1 od; {k=k(x0) ∧ x=H^k(x0)};
  x:=G(x); {k=k(x0) ∧ x=GH^k(x0)};
  while k≠0 do F1; k:=k-1 od.
```

where we have added some assertions to make it clearer.

Copy $\mathbf{F}_1$ into **F** to get:

```
F ≡ κκκκk:=0;
  while ¬P(x) do x:=H(x); k:=k+1 od;
  x:=G(x);
  while k≠0 do
    while ¬P(x) do x:=GH(x) od;
    x:=G(x); k:=k-1 od.
```

Writing the last two statements of **F** in the form of unguarded loops and **exit**s gives:

```
F2 ≡ κκκκx:=G(x);
  do if k=0 then exit fi;
    do if P(x) then exit fi;
     x:=GH(x) od;
    x:=G(x); k:=k-1 od.
```

we can exchange the last two assignments since they are independent:

```
F2 ≡ κκκκx:=G(x);
  do if k=0 then exit fi;
    do if P(x) then exit fi;
     x:=GH(x) od;
    k:=k-1; x:=G(x) od.
```

this is of the form: **a; do t; b; a od** where **a** and **b** are proper sequences of statements (ie they contain no **exit**(**n**)s within less than **n** nested loops). So we can apply loop inversion to this to get **do a; t; b od** ie:

```
F₂ ≡ κκκκdo x:=G(x)
  if k=0 then exit fi;
  do if P(x) then exit fi;
    x:=GH(x) od;
  k:=k-1 od.
```

Since the assignment **x:=G(x)** doesn't change the value of **k** we can push it inside the **if** statement to get: **if k=0 then x:=G(x); exit else x:=G(x) fi;** and detach the **else** part (since the **if** part causes an **exit**) to get **if k=0 then x:=G(x); exit fi; x:=G(x);**.
Now use the inverse of absorption on **x:=G(x)** to take it outside the loop:

```
F₂ ≡ do if k=0 then exit fi;
  x:=G(x);
  do if P(x) then exit fi;
    x:=GH(x) od;
  k:=k-1 od;
 x:=G(x).
```

Re-writing this using **while** loops gives:

```
F₂ ≡ while k≠0 do
   x:=G(x);
   while ¬P(x) do
     x:=GH(x) od;
   k:=k-1 od;
 x:=G(x).
```

and finally copy this back into **F** to get:

```
F ≡ k:=0;
 while ¬P(x) do x:=H(x); k:=k+1 od;
 while k≠0 do
   x:=G(x);
   while ¬P(x) do
     x:=GH(x) od;
   k:=k-1 od;
 x:=G(x).

F₁ ≡ while ¬P(x) do x:=GH(x) od; x:=G(x).
```

To show that the final versions of **F** and $\mathbf{F}_1$ are equivalent we will introduce a change in the data representation: for a fixed $\mathbf{x}_0$ we represent $\mathbf{G}^i\mathbf{H}^j(\mathbf{x}_0)$ by the ordered pair of natural numbers **(i,j)**, thus our representation function is: $\Psi(\mathbf{G}^i\mathbf{H}^j(\mathbf{x}_0))=(\mathbf{i},\mathbf{j})\ \mathbf{i},\mathbf{j}\in\mathbb{N}$.

$\mathbf{P}(\mathbf{x})$ becomes $\mathbf{P}_1(\mathbf{i},\mathbf{j})$ where $\mathbf{P}_1(\mathbf{i},\mathbf{j}) \iff \mathbf{P}(\mathbf{G}^i\mathbf{H}^j(\mathbf{x}_0))$

x:=H(x) becomes **j:=j+1** and **x:=G(x)** becomes **i:=i+1**.

This representation is well-defined because **G** and **H** commute, so the result of any sequence of **G**'s and **H**'s applied to $\mathbf{x}_0$ can be expressed in the form $\mathbf{G}^i\mathbf{H}^j(\mathbf{x}_0)$ for some pair $\langle\mathbf{i},\mathbf{j}\rangle$. Under this representation **F** and $\mathbf{F}_1$ for a given $\mathbf{x}_0$ become **K; j:=j+1** and $\mathbf{K}_1$**; j:=j+1** where

$\mathbf{K}\kappa\kappa\kappa\ \equiv$ **i:=0; j:=0; k:=0;**
  while $\neg\mathbf{P}_1(\mathbf{i},\mathbf{j})$ **do** **i:=i+1; k:=k+1** **od**; $\{\mathbf{k}=\mathbf{k}(\mathbf{x}_0)\ \wedge\ \mathbf{i}=\mathbf{k}\}$;
  while $\mathbf{k}\neq\mathbf{0}$ **do**
   j:=j+1;
   while $\neg\mathbf{P}_1(\mathbf{i},\mathbf{j})$ **do** **i:=i+1; j:=j+1** **od**;
   k:=k+1 **od**.
$\mathbf{K}_1\ \equiv$ **i:=0; j:=0;** **while** $\neg\mathbf{P}_1(\mathbf{i},\mathbf{j})$ **do** **i:=i+1; j:=j+1** **od**.

where we have left out the final **j:=j+1** in both routines to simplify the exposition.
If we can show that the final values of **i** and **j** after **K** are the same as after $\mathbf{K}_1$ then we have proved the equivalence of **F** and $\mathbf{F}_1$ for all initial values $\mathbf{x}_0$.

This can be proved using the following assertions:

(1) **K** and $\mathbf{K}_1$ both finish on the diagonal of the **(i,j)** plane, ie **i=j** finally.

Proof: In $\mathbf{K}_1$ the assertion **i=j** is set up by the first assignments and remains invariant over the loop. In **K** the assertion **i-k-j=0** is set up by the first assignments and preserved in the first loop. In the inner loop the value of **i-j-k** is preserved so the assertion **i-k-j=0** is satisfied after each outer loop. On exit we have **k=0** hence **i=j**.

(2) For $\mathbf{K}_1$ $\mathbf{P}_1(\mathbf{i},\mathbf{i})$ holds on exit (since **i=j**) and $\forall\mathbf{l},\ \mathbf{0}\leqslant\mathbf{l}<\mathbf{i}.\ \neg\mathbf{P}_1(\mathbf{l},\mathbf{l})$ holds.

Proof: **i** and **j** are only incremented when $\neg\mathbf{P}_1(\mathbf{i},\mathbf{j})$ holds.
Also $\forall\mathbf{l},\ \mathbf{i}\leqslant\mathbf{l}\ \mathbf{P}_1(\mathbf{l},\mathbf{l})$ holds since $\mathbf{P}_1(\mathbf{l},\mathbf{l})\Rightarrow\mathbf{P}_1(\mathbf{l}+\mathbf{1},\mathbf{l}+\mathbf{1})$ by condition (1).
So $\mathbf{K}_1$ finishes on the first point on the diagonal such that $\mathbf{P}_1$ holds.

(3) For **K** $((\mathbf{i}>\mathbf{0}\ \wedge\ \mathbf{j}>\mathbf{0})\Rightarrow\neg\mathbf{P}_1(\mathbf{i}-\mathbf{1},\mathbf{j}-\mathbf{1}))$ is an invariant.

Proof: It trivially holds in the first loop since **j=0**. In the outer loop, after the execution of **j:=j+1** we have:
Case(i): **j=1**: **i=k>0** so $\neg\mathbf{P}_1(\mathbf{i}-\mathbf{1},\mathbf{0})$ holds (by defn of **k**) so $\neg\mathbf{P}_1(\mathbf{i}-\mathbf{1},\mathbf{j}-\mathbf{1})$ holds.
Case(ii): **j>1**: **i=k>0** so $\neg\mathbf{P}_1(\mathbf{i}-\mathbf{1},\mathbf{j}-\mathbf{2})$ holds, since the invariant holds before the assignment, so by condition (3) $\mathbf{P}_1(\mathbf{i}-\mathbf{1},\mathbf{j}-\mathbf{1}+\mathbf{1})=\mathbf{P}_1(\mathbf{i}-\mathbf{1},\mathbf{j}-\mathbf{1})$ so $\neg\mathbf{P}_1(\mathbf{i}-\mathbf{1},\mathbf{j}-\mathbf{1})$ holds.

In the inner loop the guard gives $\neg\mathbf{P}_1(\mathbf{i,j})$ so after the assignments $\neg\mathbf{P}_1(\mathbf{i{-}1,j{-}1})$ holds.
Let $\mathbf{i}_1$ be the value of $\mathbf{i}$ after $\mathbf{K}_1$.
By **(1)** and **(3)** we have after **K**: $\mathbf{i{=}j} \wedge \mathbf{P}_1(\mathbf{i,i}) \wedge \big(\mathbf{i{>}0} \Rightarrow \neg\mathbf{P}_1(\mathbf{i{-}1,i{-}1})\big)$
By **(2)**: $\mathbf{P}_1(\mathbf{i,i}) \Rightarrow \mathbf{i}\geqslant\mathbf{i}_1 \geqslant\mathbf{0}$ while $\big(\mathbf{i{>}0} \Rightarrow \neg\mathbf{P}_1(\mathbf{i{-}1,i{-}1})\big) \Rightarrow \mathbf{i{-}1{<}i}_1$. Hence $\mathbf{i{=}i}_1$ and **K** and $\mathbf{K}_1$ are equivalent.

Hence **F** and $\mathbf{F}_1$ are equivalent as required.

<u>Example:</u>

McCarthy's "91-Function" (cf [McCarthy 60]) is defined:
<u>funct</u> F(x) $\equiv$
 <u>if</u> x>100 <u>then</u> x−10 <u>else</u> F(F(x+11)) <u>fi</u>.

Here $\mathbf{P(x)} \iff \mathbf{x{>}100}$, $\mathbf{G(x){=}x{-}10}$ and $\mathbf{H(x){=}x{+}11}$. So for all **x**: $\mathbf{GH(x){=}HG(x){=}x{+}1}$. The three conditions hold since:
(1): $\forall\mathbf{x}.\ \mathbf{x{>}100} \Rightarrow \mathbf{x{+}1{>}100}$
(2): $\forall\mathbf{x}.\ \mathbf{GH(x){=}HG(x){=}x{+}1}$
(3): $\forall\mathbf{x}.\ \mathbf{x{-}10{>}100} \Rightarrow \mathbf{x{>}100}$.

So by applying the theorem **F** is equivalent to $\mathbf{F}_1$ where:
  <u>funct</u> $\mathbf{F}_1\mathbf{(x)} \equiv$
   <u>if</u> x>100 <u>then</u> x−10 <u>else</u> $\mathbf{F}_1\mathbf{(x{+}1)}$ **<u>fi</u>.**
This is equivalent to the tail-recursive procedure:
  <u>proc</u> $\mathbf{F}_1 \equiv$
   <u>if</u> x>100 <u>then</u> r:=x−10 <u>else</u> x:=x+1; $\mathbf{F}_1$ **<u>fi</u>**
If $\mathbf{x}\leqslant\mathbf{101}$ initially then this repeatedly increments **x** until it is greater than **100** and the subtracts **10**.
Hence if $\mathbf{x}\leqslant\mathbf{101}$ initially this procedure is equivalent to:
<u>proc</u> $\mathbf{F}_2 \equiv$ **r:=91.** (From which we get the name of the function!).

A rigorous proof of this uses the induction rule for recursion. We have:
{x⩽101}; <u>if</u> x>100 <u>then</u> r:=x−10 <u>else</u> r:=91 <u>fi</u>
  $\approx$ **{x⩽101}; <u>if</u> x>100 <u>then</u> {x=101}; r:=x−10 <u>else</u> r:=91 <u>fi</u>**
  $\approx$ **{x⩽101}; <u>if</u> x>100 <u>then</u> r:=91 <u>else</u> r:=91 <u>fi</u>**
  $\approx$ **{x⩽101}; r:=91.**

If we let **t** be the term: **101−x** then we have:

$\{x \leqslant 101\}$; $\underline{\textbf{if}}$ $x>100$ $\underline{\textbf{then}}$ $r:=x-10$ $\underline{\textbf{else}}$ $x:=x+1$; F_1 $\underline{\textbf{fi}}$

$\approx$ $\{x \leqslant 101\}$; $t_0:=101-x$; $\{t_0 \geqslant 0\}$;

$\underline{\textbf{if}}$ $x>100$ $\underline{\textbf{then}}$ $r:=x-10$ $\underline{\textbf{else}}$ $x:=x+1$; $\{101-x<t_0\}$; F_1 $\underline{\textbf{fi}}$

Hence by the induction rule for recursion, for $x \leqslant 101$ initially, F_1 is equivalent to: $\underline{\textbf{proc}}$ $F_2 \equiv r:=91.$ and hence **F** is equivalent to $\underline{\textbf{funct}}$ $F_3(x) \equiv 91.$