Restructuring and Destructuring

Martin Ward

Reader in Software Engineering

martin@gkc.org.uk

Software Technology Research Lab
De Montfort University

Restructuring Transformations

An unstructured program (action system) can be made more structured using these transformations:

- **Expand Call:** Replace an action **call** by a copy of the action body
- **Substitute and Delete:** Apply Expand_Call to all the calls of the selected action, and then delete the action (provided the action does not call itself!)
- **Remove Recursion in Action:** In a regular action system, an action which calls itself can be transformed into an action which does not call itself by introducing loops
- **Floop to While:** a suitable Floop can be transformed directly to a **while** loop. In the general case, a flag may be needed.

Restructuring Transformations

- **Merge Calls in Action:** Attempt to merge two or more calls to the same action into a single call
- **Delete Rest:** In a regular action system, no action call can return, so all the rest of the statements after an action call can be deleted
- **Delete Item:** An action which is never called is "dead code" and can be deleted
- **Simplify Action System:** Applys the above transformations to simplify an action system as much as possible
- **Simplify Item:** An action system containing a single action can be converted to a loop

The next few slides illustrate each of these transformations.

Expand Call

$$\begin{aligned}
\text{inhere} &\equiv \text{inhere}(\ \textbf{var}\ );\boxed{\textbf{call}\ \text{more}}\ \textbf{end} \\
\text{more} &\equiv \textbf{if}\ m = 1 \\
&\qquad \textbf{then}\ p := \text{number}[i];\ \text{line} := \text{line} + \text{“}\textbf{,}\ \text{”} + p\ \textbf{fi}; \\
&\quad \text{last} := \text{item}[i];\ \textbf{call}\ l\ \textbf{end}
\end{aligned}$$

becomes:

Expand Call

inhere $\equiv$ inhere(**var**); **call** more **end**

more $\equiv$ **if** $m = 1$

then $p :=$ number$[i]$; line := line ++ "**,** " ++ p **fi**;

last := item$[i]$; **call** l **end**

becomes:

inhere $\equiv$ inhere(**var**);

if $m = 1$

then $p :=$ number$[i]$; line := line ++ "**,** " ++ p **fi**;

last := item$[i]$; **call** l **end**

more $\equiv$ **if** $m = 1$

then $p :=$ number$[i]$; line := line ++ "**,** " ++ p **fi**;

last := item$[i]$; **call** l **end**

If this was the only call to more, then the action can be deleted.

Substitute and Delete

If an action does not call itself, then Substitute_and_Delete applies Expand_Call to each call of the action, and then deletes the action.

Remove Recursion in Action

more $\equiv$ **if** $m = 1$ **then** $p :=$ number$[i]$;
line $:=$ line $+\!\!+$ "**,** " $+\!\!+$ p **fi**;
last $:=$ item$[i]$;
$i := i + 1$;
if $i = (n + 1)$ **then** **call** alldone **fi**;
p_1(**var**);
call more **end**

becomes:

Remove Recursion in Action

more $\equiv$ **if** $m = 1$ **then** $p :=$ number$[i]$;
line $:=$ line $+\!\!+$ "**,** " $+\!\!+$ p **fi**;
last $:=$ item$[i]$;
$i := i + 1$;
if $i = (n + 1)$ **then** **call** alldone **fi**;
p_1(**var**);
call more **end**

becomes:

more $\equiv$ **do** **if** $m = 1$ **then** $p :=$ number$[i]$;
line $:=$ line $+\!\!+$ "**,** " $+\!\!+$ p **fi**;
last $:=$ item$[i]$;
$i := i + 1$;
if $i = (n + 1)$ **then** **call** alldone **fi**;
p_1(**var**) **od** **end**

Remove Recursion in Action

Sometimes a double loop is needed.

```
more  ≡ i := i + 1;
        if i < n + 1 then [call more]
        elsif B1?(i) then p_1( var )
        elsif B2?(i) then [call more] fi;
        p_3( var );
        call alldone end
```

Remove Recursion in Action

Sometimes a double loop is needed.

more $\equiv i := i + 1;$
  if $i < n + 1$ **then** [**call** more]
  elsif B1?(i) **then** p_1(**var**)
  elsif B2?(i) **then** [**call** more] **fi**;
  p_3(**var**);
  call alldone **end**

becomes:

more $\equiv$ **do do** $i := i + 1;$
    if $i < n + 1$ **then** [**exit**]
    elsif B1?(i) **then** p_1(**var**)
    elsif B2?(i) **then** [**exit**] **fi**;
    p_3(**var**);
    call alldone **od od end**

Take Outside Loop

$$\textbf{do if } X = 1 \textbf{ then } \boxed{Y := 1;\ X := 0};\ \textbf{exit}(2)$$
$$\textbf{elsif } X = 2$$
$$\textbf{then } \boxed{Y := 1;\ X := 0};\ \textbf{exit}(2)$$
$$\textbf{else } X := X - Y \textbf{ fi od}$$

becomes

Take Outside Loop

do if $X = 1$ **then** $\boxed{Y := 1;\ X := 0}$; **exit**(2)
  elsif $X = 2$
    then $\boxed{Y := 1;\ X := 0}$; **exit**(2)
    else $X := X - Y$ **fi od**

becomes

do do if $X = 1$ **then exit**(1)
    elsif $X = 2$
      then exit(1)
      else $X := X - Y$ **fi od**;
  $\boxed{Y := 1;\ X := 0}$; **exit**(1) **od**

Double To Single Loop

do do $i := i + 1$;
if $i < n + 1$ **then exit**
elsif B1?(i) **then** p_1(**var**); **exit**(2)
elsif B2?(i) **then exit**
else exit(2) **fi od od**

becomes:

Double To Single Loop

do do $i := i + 1;$
    if $i < n + 1$ **then exit**
    elsif B1?(i) **then** p_1(**var**); **exit**(2)
    elsif B2?(i) **then exit**
        else exit(2) **fi od od**

becomes:

do $i := i + 1;$
  if $i < n + 1$ **then skip**
  elsif B1?(i) **then** p_1(**var**); **exit**
  elsif B2?(i) **then skip**
      else exit fi od

Floop to While

do $i := i + 1;$
    if $i < n + 1$ **then skip**
    elsif B1?(i) **then** p_1(**var**); **exit**
    elsif B2?(i) **then skip**
                else exit fi od

becomes:

Floop to While

```
do i := i + 1;
    if i < n + 1 then skip
    elsif B1?(i) then p_1( var ); exit
    elsif B2?(i) then skip
                      else exit fi od
```

becomes:

```
fl_flag1 := 0;
while fl_flag1 = 0 do
    i := (i + 1);
    if i < (n + 1) then fl_flag1 := 0
    elsif B1?(i) then p_1( var ); fl_flag1 := 1
    elsif B2?(i) then fl_flag1 := 0
                      else fl_flag1 := 1 fi od;
```

Floop to While

Simpler loop:

do $i := (i + 1)$;
  if $(n + 1) \leqslant i \wedge$ B1?(i)
    then exit(1)
  elsif $(n + 1) \leqslant i \wedge \neg$B2?$(i)$
      then exit(1)
  elsif $i < (n + 1)$
      then skip fi od;

becomes:

Floop to While

Simpler loop:

do $i := (i + 1)$;
  if $(n + 1) \leqslant i \wedge$ B1?(i)
    then exit(1)
  elsif $(n + 1) \leqslant i \wedge \neg$B2?$(i)$
    then exit(1)
  elsif $i < (n + 1)$
    then skip fi od;

becomes:

$i := (i + 1)$;
while $\neg$B1?$(i) \wedge$ B2?$(i) \vee i < (n + 1)$ **do**
  $i := (i + 1)$ **od**;

Note that the statement $i := i + 1$ had to be copied.

Merge Calls in Action

$K \equiv$ **if** item$[i] \neq$ last
then $!P$ write(line **var** os);
line := "";
$m := 0$;
inhere(**var**);
[**call** more] **fi**;
[**call** more] **end**

Merge the two calls into one:

Merge Calls in Action

$$
\begin{aligned}
K \;\equiv\; & \textbf{if}\ \text{item}[i] \neq \text{last} \\
& \quad \textbf{then}\ !P\ \text{write}(\text{line}\ \textbf{var}\ \text{os}); \\
& \qquad \text{line} := \text{``''}; \\
& \qquad m := 0; \\
& \qquad \text{inhere}(\ \textbf{var}\ ); \\
& \qquad \boxed{\textbf{call}\ \text{more}}\ \textbf{fi}; \\
& \boxed{\textbf{call}\ \text{more}}\ \textbf{end}
\end{aligned}
$$

Merge the two calls into one:

$$
\begin{aligned}
K \;\equiv\; & \textbf{if}\ \text{item}[i] \neq \text{last} \\
& \quad \textbf{then}\ !P\ \text{write}(\text{line}\ \textbf{var}\ \text{os}); \\
& \qquad \text{line} := \text{``''}; \\
& \qquad m := 0; \\
& \qquad \text{inhere}(\ \textbf{var}\ )\ \textbf{fi}; \\
& \boxed{\textbf{call}\ \text{more}}\ \textbf{end}
\end{aligned}
$$

Delete Rest

In a **regular** action system, any statements immediately following a **call** can be deleted.

Similarly, any statements following an **exit** can be deleted.

$x := y$; **call** A; $x := x + 1$

becomes:

$x := y$; **call** A

Simplify Item

If a regular action system contains a single action, then there can only be two types of call:

- Calls to the action itself
- Calls to the terminating action (Z)

The action system is replaced by a double loop:

- Calls to the action itself are replaced by **exit**
- Calls to Z are replaced by **exit**(2)

In simple cases, the double loop may be converted to a single loop, or eliminated altogether (if there are no calls to the action itself).

Assembler Migration

An Intel assembler program to compute a GCD:

```
.model small
.code
            mov ax,12
            mov bx,8
compare:
            cmp ax,bx
            je  theend
            ja  greater
            sub bx,ax
            jmp compare
greater:
            sub ax,bx
            jmp compare
theend:
            nop
end
```

WSL Translation

```
var ⟨flag_z := 0, flag_c := 0⟩ :
  actions A_S_start :
  A_S_start  ≡ ax := 12;
               bx := 8;
               call compare end
  compare  ≡ if ax = bx then flag_z := 1 else flag_z := 0 fi;
             if ax < bx then flag_c := 1 else flag_c := 0 fi;
             if flag_z = 1 then call theend fi;
             if flag_z = 0 ∧ flag_c = 0
               then call greater fi;
             if bx = ax then flag_z := 1 else flag_z := 0 fi;
             if bx < ax then flag_c := 1 else flag_c := 0 fi;
             bx := bx − ax;
             call compare;
             call greater end
  greater  ≡ if ax = bx then flag_z := 1 else flag_z := 0 fi;
             if ax < bx then flag_c := 1 else flag_c := 0 fi;
             ax := ax − bx;
             call compare;
             call theend end
  theend  ≡ call Z end endactions end
```

Flag Removal

```
actions A_S_start :
A_S_start  ≡ ax := 12;
              bx := 8;
              call compare end
compare  ≡ if ax = bx
              then if ax < bx
                        then call theend
                         else call theend fi
               else if ax ⩾ bx
                        then call greater fi fi;
           bx := (bx − ax);
           call compare;
           call greater end
greater  ≡ ax := (ax − bx);
           call compare;
           call theend end
theend  ≡ call Z end endactions
```

Collapse Action System

$\mathsf{ax} := 12;$

$\mathsf{bx} := 8;$

$\textbf{do if } \mathsf{ax} = \mathsf{bx}$

$\quad\textbf{then if } \mathsf{ax} < \mathsf{bx}$

$\qquad\quad\textbf{then exit}(1)$

$\qquad\quad\textbf{else exit}(1) \textbf{ fi}$

$\quad\textbf{else if } \mathsf{ax} \geqslant \mathsf{bx}$

$\qquad\quad\textbf{then } \mathsf{ax} := (\mathsf{ax} - \mathsf{bx})$

$\qquad\quad\textbf{else } \mathsf{bx} := (\mathsf{bx} - \mathsf{ax}) \textbf{ fi fi od}$

Simplify

$\mathsf{ax} := 12;$

$\mathsf{bx} := 8;$

while $\mathsf{ax} \neq \mathsf{bx}$ **do**

  if $\mathsf{ax} \geqslant \mathsf{bx}$

    then $\mathsf{ax} := \mathsf{ax} - \mathsf{bx}$

    else $\mathsf{bx} := \mathsf{bx} - \mathsf{ax}$ **fi od**

Program Metrics

Metric	Raw WSL	Flags	Collapse	Simplify
Statements	36	18	10	6
Expressions	40	14	14	12
McCabe	9	4	4	3
Control/Data Flow	45	23	14	12
Branch–Loop	8	9	1	1
Structural	242	143	58	40

Destructuring Transformations

A *restructuring transformation* changes the structure of a program without changing the sequence of state changes which occur during the execution of the program.

Such a transformation preserves the *operational semantics* of the program.

For example:

$$\textbf{if } \mathsf{B} \textbf{ then } \mathsf{S}_1 \textbf{ else } \mathsf{S}_2 \textbf{ fi}$$

is equivalent to:

$$\textbf{if } \neg\mathsf{B} \textbf{ then } \mathsf{S}_2 \textbf{ else } \mathsf{S}_1 \textbf{ fi}$$

Destructuring Transformations

Assignment merging is *not* a restructuring transformation:

$$x := e_1;\ x := e_2$$

is equivalent to:

$$x := e_2[e_1/x]$$

For example:

$$x := 2 * x;\ x := x + 1$$

is equivalent to:

$$x := 2 * x + 1$$

The first program has two state changes, but the second has only one, so these are not *operationally* equivalent.

Destructuring Transformations

One method to prove the correctness of a proposed restructuring transformation:

1. Convert the first program to a regular action system with no structured statements
2. Convert the second program to a regular action system with no structured statements
3. Transform the two action systems to a common format

This can sometimes be easier than trying to transform one program directly into the other.

Convert to an Action System

Any program **S** is equivalent to the regular action system:

actions start :

start $\equiv$ **S**; **call** Z **end endactions**

Now, process structured statements in **S** from the top down, adding new actions to the action system as required.

Destructuring A Sequence

Suppose we have an action containing a sequence of statements:

$A_0 \;\equiv$ **S**$_1$; **S**$_2$; $\ldots$; **S**$_n$; **call** B **end**

This is equivalent to the set of actions:

$A_0 \;\equiv$ **S**$_1$; **call** A_1 **end**

$A_1 \;\equiv$ **S**$_2$; **call** A_2 **end**

$A_{n-1} \;\equiv$ **S**$_n$; **call** B **end**

Destructuring a Conditional

$A_0 \equiv$ **if B**$_1$ **then S**$_1$
elsif B$_2$ **then S**$_2$
elsif $\ldots$
else S$_n$ **fi**;
call B **end**

This is equivalent to:

$A_0 \equiv$ **if B**$_1$ **then call** A_1
elsif B$_2$ **then call** A_2
elsif $\ldots$
else call A_n **fi end**

$A_1 \equiv$ **S**$_1$; **call** B **end**

$\ldots$

$A_n \equiv$ **S**$_n$; **call** B **end**

Destructuring a While Loop

$A_0 \equiv$ **while B do S od**; **call** B **end**

This is equivalent to:

$A_0 \equiv$ **if B then call** A_1 **else call** B **fi end**

$A_1 \equiv$ **S**; **call** A_0 **end**

Destructuring Floops

To destructure an Floop, first absorb the following **call** into the loop:

$A_0 \equiv$ **do S od**; **call** B **end**

is transformed to:

$A_0 \equiv$ **do S**$'$ **od end**

where **S**$'$ is **S** with each **exit**(n) with terminal value 1 replaced by **call** B (i.e. every **exit** which could terminate the loop).

In other words, any **exit** which can terminate the outermost loop is replaced by **call** B.

Then replace the loop with a **call** to the action:

$A_0 \equiv$ **S**$'$; **call** A_0 **end**

This is the opposite of Remove_Recursion_In_Action.

Destructuring Floops

An example:

$$
\begin{array}{l}
A_0 \;\equiv\; \textbf{do}\ \text{inhere}(\ \textbf{var}\ );\\
\quad\quad \textbf{do}\ \textbf{if}\ m = 1\\
\quad\quad\quad\quad \textbf{then}\ p := \text{number}[i];\ \text{line} := \text{line} +\!\!\!+ \text{``}\textbf{,}\ \text{''} +\!\!\!+ p\ \textbf{fi};\\
\quad\quad\quad \text{last} := \text{item}[i];\ i := (i+1);\\
\quad\quad\quad \textbf{if}\ i = (n+1)\\
\quad\quad\quad\quad \textbf{then}\ !P\ \text{write}(\text{line}\ \textbf{var}\ \text{os}); \boxed{\textbf{exit}(2)}\ \textbf{fi};\\
\quad\quad\quad m := 1;\\
\quad\quad\quad \textbf{if}\ \text{item}[i] \neq \text{last}\\
\quad\quad\quad\quad \textbf{then}\ !P\ \text{write}(\text{line}\ \textbf{var}\ \text{os});\\
\quad\quad\quad\quad\quad\quad \text{line} := \text{``''};\\
\quad\quad\quad\quad\quad\quad m := 0;\\
\quad\quad\quad\quad\quad\quad \textbf{exit}(1)\ \textbf{fi}\ \textbf{od}\ \textbf{od};\\
\quad \textbf{call}\ Z\ \textbf{end}
\end{array}
$$

Destructuring Floops

Absorb the **call**:

A_0 $\equiv$ **do** inhere(**var**);
do **if** $m = 1$
then $p :=$ number$[i]$; line := line $+\!\!\!+$ "**,** " $+\!\!\!+$ p **fi**;
last := item$[i]$; $i := (i + 1)$;
if $i = (n + 1)$
then $!P$ write(line **var** os); **call** Z **fi**;
$m := 1$;
if item$[i] \neq$ last
then $!P$ write(line **var** os);
line := "";
$m := 0$;
exit(1) **fi** **od** **od** **end**

Destructuring Floops

Remove the loop:

$A_0 \equiv$ inhere(**var**);

do if $m = 1$

then $p :=$ number$[i]$; line $:=$ line $+\!\!+$ "**,** " $+\!\!+$ p **fi**;

last $:=$ item$[i]$; $i := (i + 1)$;

if $i = (n + 1)$

then $!P$ write(line **var** os); $\boxed{\textbf{call } Z}$ **fi**;

$m := 1$;

if item$[i] \neq$ last

then $!P$ write(line **var** os);

line $:=$ "";

$m := 0$;

exit(1) **fi od**;

call A_0 **end**

Destructuring Floops

Processing the inner loop.

Process the sequence and then absorb the **call**:

$A_0 \equiv$ inhere(**var**); **call** A_1 **end**

$A_1 \equiv$ **do if** $m = 1$

then $p :=$ number$[i]$; line $:=$ line $+\!\!+$ "**,** " $+\!\!+\ p$ **fi**;

last $:=$ item$[i]$; $i := (i + 1)$;

if $i = (n + 1)$

then $!P$ write(line **var** os); **call** Z **fi**;

$m := 1$;

if item$[i] \neq$ last

then $!P$ write(line **var** os);

line $:=$ "";

$m := 0$;

call A_0 **fi od end**

Destructuring Floops

Processing the inner loop. Remove the loop:

$A_0 \ \equiv$ inhere(**var**); **call** A_1 **end**

$A_1 \ \equiv$ **if** $m = 1$

  then $p :=$ number$[i]$; line := line ++ "**,** " ++ p **fi**;

 last := item$[i]$; $i := (i+1)$;

 if $i = (n+1)$

  then $!P$ write(line **var** os); **call** Z **fi**;

 $m := 1$;

 if item$[i] \neq$ last

  then $!P$ write(line **var** os);

   line := "";

   $m := 0$;

   call A_0 **fi**;

 call A_1 **end**

Loop Inversion

Some transformations can be proved correct by converting both programs to action systems and analysing the action systems using Expand_Call (and its inverse), case analysis, renaming etc.

For example, to prove that P_1:

do $\mathbf{S}_1$;
    if B then exit fi;
    $\mathbf{S}_2$ **od**

is equivalent to P_2:

$\mathbf{S}_1$;
while $\neg$**B do**
    $\mathbf{S}_2$;
    $\mathbf{S}_1$ **od**

where $\mathbf{S}_1$ and $\mathbf{S}_2$ are both *proper sequences*.

Loop Inversion

P_2 translates to this action system:

actions A_0 :

$A_0 \equiv$ **S**$_1$; **call** A_1 **end**

$A_1 \equiv$ **if B then call** Z **else call** A_2 **fi end**

$A_2 \equiv$ **S**$_2$; **call** A_3 **end**

$A_3 \equiv$ **S**$_1$; **call** A_1 **end endactions**

P_1 translates to this action system:

actions A_0 :

$A_0 \equiv$ **S**$_1$; **call** A_1 **end**

$A_1 \equiv$ **if B then call** Z **else call** A_2 **fi end**

$A_2 \equiv$ **S**$_2$; **call** A_0 **end endactions**

Loop Inversion

Expand the **call** A_0 in A_2:

$A_2 \equiv$ **S**$_2$; **S**$_1$; **call** A_1 **end**

Destructure the sequence:

$A_2 \equiv$ **S**$_2$; **call** A_3 **end**

$A_3 \equiv$ **S**$_1$; **call** A_1 **end**

The two action systems are now identical.

Loop Unrolling

To prove that the program P_1:

while B do S od

is equivalent to P_2:

while B do S; **if B** $\wedge$ **Q then S fi od**

P_1 as an action system:

actions A_0 :

$A_0 \equiv$ **while B do S od**; **call** Z **end endactions**

Destructure the action system:

actions A_0 :

$A_0 \equiv$ **if B then call** A_1 **else call** Z **fi end**

$A_1 \equiv$ **S**; **call** A_0 **end endactions**

Loop Unrolling

P_2 as an action system:

actions A_0 :

$A_0 \equiv$ **while B do S**; **if B** $\wedge$ **Q then S fi od**; **call** Z **end endactions**

Destructure the action system:

actions A_0 :

$A_0 \equiv$ **if B then call** A_1 **else call** Z **fi end**

$A_1 \equiv$ **S**; **call** A_2 **end**

$A_2 \equiv$ **if B** $\wedge$ **Q then** A_3 **else call** A_0 **fi end**

$A_3 \equiv$ **S**; **call** A_0 **end endactions**

Loop Unrolling

Consider the P_1 action system again:

actions A_0 :

$A_0 \equiv$ **if B then call** A_1 **else call** Z **fi end**

$A_1 \equiv$ **S**; **call** A_0 **end endactions**

P_2 has an action $A_3 \equiv$ **S**; **call** A_0 **end** so we add this action to P_1 and note that any call to A_1 can be replaced by a call to A_3.

In particular, A_0 is equivalent to:

$A'_0 \equiv$ **if B then call** A_3 **else call** Z **fi end**

Also, P_2 has **if B** $\wedge$ **Q then** $\ldots$ **fi** where P_1 has **call** A_0.

So replace **call** A_0 in A_1 by the equivalent statement:

if B $\wedge$ **Q then call** A'_0 **else call** A_0 **fi**

Loop Unrolling

Unroll **call** A_0' in A_1:

actions A_0 :

$A_0 \equiv$ **if B then call** A_1 **else call** Z **fi end**

$A_1 \equiv$ **S**; **if B** $\wedge$ **Q then if B then call** A_3 **else call** Z **fi**

else call A_0 **fi end**

$A_3 \equiv$ **S**; **call** A_0 **end endactions**

Simplify:

actions A_0 :

$A_0 \equiv$ **if B then call** A_1 **else call** Z **fi end**

$A_1 \equiv$ **S**; **if B** $\wedge$ **Q then call** A_3

else call A_0 **fi end**

$A_3 \equiv$ **S**; **call** A_0 **end endactions**

Loop Unrolling

Destructure:

actions A_0 :

$A_0 \equiv$ **if B then call** A_1 **else call** Z **fi end**

$A_1 \equiv$ **S**; **call** A_2 **end**

$A_2 \equiv$ **if B** $\wedge$ **Q then call** A_3 **else call** A_0 **fi end**

$A_3 \equiv$ **S**; **call** A_0 **end endactions**

This is identical to the destructured version of P_2.

Entire Loop Unrolling

To prove that P_1:

while B do S od

is equivalent to P_3:

while B do S; **while B** $\wedge$ **Q do S od od**

Convert P_3 to an action system:

actions A_0 :

$A_0 \equiv$ **if B then call** A_1 **else call** Z **fi end**

$A_1 \equiv$ **S**; **call** A_2 **end**

$A_2 \equiv$ **if B** $\wedge$ **Q then call** A_3 **else call** A_0 **fi end**

$A_3 \equiv$ **S**; **call** A_2 **end endactions**

This is the same as P_2 (which we have proved to be equivalent to P_1) except that there is a **call** A_2 in the body of A_3 instead of **call** A_0.

Entire Loop Unrolling

actions A_0 :

A_0 $\equiv$ **if B then call** A_1 **else call** Z **fi end**

A_1 $\equiv$ **S**; **call** A_2 **end**

A_2 $\equiv$ **if B** $\wedge$ **Q then call** A_3 **else call** A_0 **fi end**

A_3 $\equiv$ **S**; **call** A_2 **end endactions**

Case analysis to prove **call** A_2 is equivalent to **call** A_0 in A_3:

1. If **B** is false or **Q** is false, then **call** A_2 leads to **call** A_0
2. If **B** is true and **Q** is true, then
   (a) **call** A_2 leads, via **call** A_3, to execute **S** and **call** A_2, while
   (b) **call** A_0 leads, via **call** A_1, to execute **S** and **call** A_2

Entire Loop Unrolling

Another way to prove that **call** A_2 is equivalent to **call** A_0 in A_3 is to replace **call** A_2 by the equivalent statement:

if B $\wedge$ **Q then call** A_2
else call A_2 **fi**

Expand each call and simplify:

actions A_0 :
$A_0 \equiv$ **if B then call** A_1 **else call** Z **fi end**
$A_1 \equiv$ **S**; **call** A_2 **end**
$A_2 \equiv$ **if B** $\wedge$ **Q then call** A_3 **else call** A_0 **fi end**
$A_3 \equiv$ **S**;
if B $\wedge$ **Q then S**; **call** A_1
else call A_0 **fi end endactions**

Replace **S**; **call** A_1 by **call** A_0 (since **B** is true here). We have:

$A_3 \equiv$ **S**; **call** A_0 **end**