Formal Transformations and WSL

Part Two

Martin Ward

Reader in Software Engineering

martin@gkc.org.uk

Software Technology Research Lab
De Montfort University

Types of Transformations

Types of Transformations

A *Syntactic Transformation* changes the syntax of the program but preserves the exact sequence of operations carried out by the program. Many restructuring transformations are purely syntactic.

Types of Transformations

A *Syntactic Transformation* changes the syntax of the program but preserves the exact sequence of operations carried out by the program. Many restructuring transformations are purely syntactic.

A *Semantic Transformation* may change the sequence of operations carried out by the program, but preserves the final state.

Types of Transformations

A *Syntactic Transformation* changes the syntax of the program but preserves the exact sequence of operations carried out by the program. Many restructuring transformations are purely syntactic.

A *Semantic Transformation* may change the sequence of operations carried out by the program, but preserves the final state.

A syntactic transformation preserves the operational semantics, so these transformations are also called *Operational Transformations*.

Types of Transformations

A *Syntactic Transformation* changes the syntax of the program but preserves the exact sequence of operations carried out by the program. Many restructuring transformations are purely syntactic.

A *Semantic Transformation* may change the sequence of operations carried out by the program, but preserves the final state.

A syntactic transformation preserves the operational semantics, so these transformations are also called *Operational Transformations*.

A semantic transformation preserves the denotational semantics.

A Syntactic Transformation

For any condition (formula) **B** and any statements $\mathbf{S}_1$, $\mathbf{S}_2$ and $\mathbf{S}_3$:

if B **then** $\mathbf{S}_1$
 else $\mathbf{S}_2$ **fi**;
$\mathbf{S}_3$

is equivalent to:

if B **then** $\mathbf{S}_1$; $\mathbf{S}_3$
 else $\mathbf{S}_2$; $\mathbf{S}_3$ **fi**

A Syntactic Transformation

For any condition (formula) **B** and any statements $\mathbf{S}_1$, $\mathbf{S}_2$ and $\mathbf{S}_3$:

if B then $\mathbf{S}_1$
    else $\mathbf{S}_2$ **fi**;
$\mathbf{S}_3$

is equivalent to:

if B then $\mathbf{S}_1$; $\mathbf{S}_3$
    else $\mathbf{S}_2$; $\mathbf{S}_3$ **fi**

In FermaT this result can be produced by applying Absorb_Right or Expand_Forwards on the **if** statement, or Merge_Left on $\mathbf{S}_3$

Another Example

If $\mathbf{S}_3$ does not modify any of the variables in **B** then:

$\mathbf{S}_3$;
if B then $\mathbf{S}_1$
else $\mathbf{S}_2$ **fi**; $\mathbf{S}_3$

is equivalent to:

if B then $\mathbf{S}_3$; $\mathbf{S}_2$
else $\mathbf{S}_3$; $\mathbf{S}_1$ **fi**

Another Example

If $\mathbf{S}_3$ does not modify any of the variables in **B** then:

$$\begin{array}{l} \mathbf{S}_3; \\ \textbf{if B then } \mathbf{S}_1 \\ \qquad \textbf{else } \mathbf{S}_2 \textbf{ fi}; \ \mathbf{S}_3 \end{array}$$

is equivalent to:

$$\begin{array}{l} \textbf{if B then } \mathbf{S}_3; \ \mathbf{S}_2 \\ \qquad \textbf{else } \mathbf{S}_3; \ \mathbf{S}_1 \textbf{ fi} \end{array}$$

In FermaT this result can be produced by applying Absorb_Left on the **if** statement, or Merge_Right on $\mathbf{S}_3$

Splitting A Tautology

For any statement **S** and any condition **B**:

$$\textbf{S} \approx \textbf{if B then S else S fi}$$

Adding Assertions:

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

is equivalent to:

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

Splitting A Tautology

For any statement **S** and any condition **B**:

$$\textbf{S} \approx \textbf{if B then S else S fi}$$

Adding Assertions:

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

is equivalent to:

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

Assertions can be introduced and propagated through the program.

Adding Assertions

For any statement **S** and any condition **B**:

$$\textbf{while B do S od}$$

is equivalent to:

$$\textbf{while B do } \{\textbf{B}\};\ \textbf{S od};\ \{\neg\textbf{B}\}$$

A Semantic Transformation

Assignment Merging: (Merge_Left and Merge_Right on assignments)

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

is equivalent to:

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

Another example:

$$y := n * x$$

is equivalent to:

$$n := n - 1;\ y := (n + 1) * x;\ n := n + 1$$

Example Transformations

$$\textbf{if } n = 0 \textbf{ then } x := 1$$
$$\textbf{else } x := x + 1 \textbf{ fi};$$
$$x := 2 * x$$

Example Transformations

if $n = 0$ **then** $x := 1$
  else $x := x + 1$ **fi**;
$x := 2 * x$

Expand the **if** statement:

if $n = 0$ **then** $x := 1;\ x := 2 * x$
  else $x := x + 1;\ x := 2 * x$ **fi**

Example Transformations

if $n = 0$ **then** $x := 1$
else $x := x + 1$ **fi**;
$x := 2 * x$

Expand the **if** statement:

if $n = 0$ **then** $x := 1;\ x := 2 * x$
else $x := x + 1;\ x := 2 * x$ **fi**

Merge the assignments:

if $n = 0$ **then** $x := 2$
else $x := 2 * (x + 1)$ **fi**

Expanding a Call

In an action system, any **call** can be replaced by a copy of the body of the action called:

actions A_1 :

$A_1 \equiv$ **S**$_1$ **end**

$\ldots$

$A_1 \equiv \ldots \boxed{\textbf{call } A_j} \ldots$ **end**

$\ldots$

$A_n \equiv$ **S**$_n$ **end endactions**

Expanding a Call

In an action system, any **call** can be replaced by a copy of the body of the action called:

actions A_1 :

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

$\ldots$

$A_1 \;\equiv \ldots \boxed{\mathbf{S}_j} \ldots$ **end**

$\ldots$

$A_n \;\equiv$ **S**$_n$ **end endactions**

Expanding a Call

In an action system, any **call** can be replaced by a copy of the body of the action called:

actions A_1 :
$A_1 \ \equiv$ **S**$_1$ **end**
$\ldots$
$A_1 \ \equiv \ldots \boxed{\mathbf{S}_j} \ldots$ **end**
$\ldots$
$A_n \ \equiv$ **S**$_n$ **end endactions**

If there are no other calls to A_j, then the action can be deleted

Expand and Separate

Suppose we have this code in a *regular* action system:

if B **then** $\mathbf{S}_1$; **call** A

  else $\mathbf{S}_2$ **fi**;

call A

Expand and Separate

Suppose we have this code in a *regular* action system:

if B **then** $\mathbf{S}_1$; **call** A
 else $\mathbf{S}_2$ **fi**;
call A

Expand the **if**:

if B **then** $\mathbf{S}_1$; **call** A; **call** A
 else $\mathbf{S}_2$; **call** A **fi**

Expand and Separate

Suppose we have this code in a *regular* action system:

if **B** **then** **S**$_1$; **call** A
  else **S**$_2$ **fi**;
call A

Expand the **if**:

if **B** **then** **S**$_1$; **call** A; **call** A
  else **S**$_2$; **call** A **fi**

Delete after the first **call**:

if **B** **then** **S**$_1$; **call** A
  else **S**$_2$; **call** A **fi**

Expand and Separate

Suppose we have this code in a *regular* action system:

if B then S$_1$; **call** A
  else S$_2$ **fi**;
call A

Expand the **if**:

if B then S$_1$; **call** A; **call** A
  else S$_2$; **call** A **fi**

Delete after the first **call**:

if B then S$_1$; **call** A
  else S$_2$; **call** A **fi**

Separate:

if B then S$_1$
  else S$_2$ **fi**;
call A

Expand and Separate

Example:

if $n = 0$ **then** $x := 1;$ **call** A

else $y := 2$ **fi**;

call A

Expand and Separate

Example:

if $n = 0$ **then** $x := 1$; **call** A

else $y := 2$ **fi**;

call A

Becomes:

if $n = 0$ **then** $x := 1$

else $y := 2$ **fi**;

call A

The first **call** A has been deleted.

Example Transformations

Forward Expansion:

if $x = 1$ **then if** $y = 1$ **then** $z := 1$ **else** $z := 2$ **fi**
else $z := 3$ **fi**;
if $z = 1$ **then** $p := q$ **fi**

is equivalent to:

if $x = 1$ **then if** $y = 1$ **then** $z := 1$ **else** $z := 2$ **fi**;
if $z = 1$ **then** $p := q$ **fi**
else $z := 3$;
if $z = 1$ **then** $p := q$ **fi fi**

Example Transformations

Absorb Right:

if $x = 1$ **then if** $y = 1$ **then** $z := 1$ **else** $z := 2$ **fi**
    else $z := 3$ **fi**;
if $z = 1$ **then** $p := q$ **fi**

is equivalent to:

if $x = 1$ **then if** $y = 1$ **then** $z := 1$;
        if $z = 1$ **then** $p := q$ **fi**
      else $z := 21$;
        if $z = 1$ **then** $p := q$ **fi fi**;
    else $z := 3$;
      if $z = 1$ **then** $p := q$ **fi fi**

This transformation is also called Merge Left!

Example Transformations

Absorb Left into a loop, before:

do do if $i > n$ **then** $\boxed{\textbf{exit}(2)}$ **fi**;
$\quad i := i + 1$;
$\quad$ **if** $A[i] = v$ **then exit**(1) **fi od**;
last $:= i$;
count $:=$ count $+ 1$;
if count $>$ limit **then** $\boxed{\textbf{exit}(1)}$ **fi od**;
if count $>$ limit **then** PRINT(last) **fi**

Example Transformations

Absorb Left into a loop, after:

do do if $i > n$ **then if** count $>$ limit **then** PRINT(last); **exit**(2)
else exit(2) **fi fi**;
$i := i + 1$;
if $A[i] = v$ **then exit**(1) **fi od**;
last $:= i$;
count $:=$ count $+ 1$;
if count $>$ limit **then if** count $>$ limit **then** PRINT(last); **exit**(1)
else exit(1) **fi fi od**;

Loop Inversion

do Read_A_Record(file, record);
    if end_of_file?(file) **then** **exit**(1) **fi**;
    Process_Record(record) **od**

Loop Inversion

```
do Read_A_Record(file, record);
   if end_of_file?(file) then exit(1) fi;
   Process_Record(record) od
```

Is equivalent to:

```
Read_A_Record(file, record);
do if end_of_file?(file) then exit(1) fi;
   Process_Record(record);
   Read_A_Record(file, record) od
```

Loop Inversion

```
do Read_A_Record(file, record);
   if end_of_file?(file) then exit(1) fi;
   Process_Record(record) od
```

Is equivalent to:

```
Read_A_Record(file, record);
do if end_of_file?(file) then exit(1) fi;
   Process_Record(record);
   Read_A_Record(file, record) od
```

Which is equivalent to:

```
Read_A_Record(file, record);
while ¬end_of_file?(file) do
   Process_Record(record);
   Read_A_Record(file, record) od
```

Loop Inversion

In general:

$$\textbf{do } \mathbf{S}_1;\ \mathbf{S}_2 \textbf{ od}$$

Is equivalent to:

$$\mathbf{S}_1;\ \textbf{do } \mathbf{S}_2;\ \mathbf{S}_1 \textbf{ od}$$

provided $\mathbf{S}_1$ is a *proper sequence* (It has no **exit** statements which can leave an enclosing loop)

Loop Inversion

More Generally:

$$\textbf{do } \mathsf{S}_1;\ \mathsf{S}_2 \textbf{ od}$$

Is equivalent to:

$$\textbf{do } \mathsf{S}_1;\ \textbf{do } \mathsf{S}_2;\ \mathsf{S}_1 \textbf{ od} + 1 \textbf{ od}$$

where the $+1$ will increment the **exit** statements which terminate **do** S_2; S_1 **od** so that they terminate the new outer loop.

Loop Inversion

Loop inversion can be used to merge two copies of a statement into one, for example:

GET(DDIN **var** WREC);
do **if** end_of_file?(DDIN) **then** **exit**(1) **fi**;
    WORKP := WREC.NUM;
    TOTAL := TOTAL + WORKP;
    GET(DDIN **var** WREC) **od**;

simplifies to:

do GET(DDIN **var** WREC);
    if end_of_file?(DDIN) **then** **exit**(1) **fi**;
    WORKP := WREC.NUM;
    TOTAL := TOTAL + WORKP **od**;

Merging Copies

A program with repeated statements:

do $\ldots$;
    if end_of_file(DDIN)
      then **exit**(1) **fi**;
    PUT_FIXED(RDSOUT, WPRT **var** result_code, os);
    fill(WPRT[1] **var** WPRT[2..80]) **od**;
PUT_FIXED(RDSOUT, WPRT **var** result_code, os);
fill(WPRT[1] **var** WPRT[2..80])

Merging Copies

Absorb into the loop:

do $\ldots;$

  if end_of_file$($DDIN$)$

   then PUT_FIXED$($RDSOUT$,$ WPRT **var** result_code$,$ os$);$

     fill$($WPRT$[1]$ **var** WPRT$[2..80]);$

     exit(1) **fi**$;$

  PUT_FIXED$($RDSOUT$,$ WPRT **var** result_code$,$ os$);$

  fill$($WPRT$[1]$ **var** WPRT$[2..80])$ **od**$;$

Merging Copies

Absorb into the **if** statement:

do $\ldots$;

if end_of_file(DDIN)

then PUT_FIXED(RDSOUT, WPRT **var** result_code, os);

fill(WPRT[1] **var** WPRT[2..80]);

exit(1)

else PUT_FIXED(RDSOUT, WPRT **var** result_code, os);

fill(WPRT[1] **var** WPRT[2..80]) **fi** **od**;

Merging Copies

Separate Left:

do $\ldots;$

  PUT_FIXED(RDSOUT, WPRT **var** result_code, os);

  fill(WPRT[1] **var** WPRT[2..80]);

  if end_of_file(DDIN)

    then **exit**(1) **od**;

Merging Copies

Here, there are two copies of $\mathbf{S}_2$ which we want to merge:

if $\mathbf{B}_1$ **then** $\mathbf{S}_1$; $\mathbf{S}_2$
elsif $\mathbf{B}_2$ **then** $\mathbf{S}_2$
else $\mathbf{S}_3$ **fi**

Merging Copies

Here, there are two copies of $\mathbf{S}_2$ which we want to merge:

if $\mathbf{B}_1$ **then** $\mathbf{S}_1$; $\mathbf{S}_2$
elsif $\mathbf{B}_2$ **then** $\mathbf{S}_2$
else $\mathbf{S}_3$ **fi**

The result is:

if $\mathbf{B}_1 \vee \mathbf{B}_2$
then if $\mathbf{B}_1$ **then** $\mathbf{S}_1$ **fi**;
$\mathbf{S}_2$
else $\mathbf{S}_3$ **fi**

An Example

if end_of_file?(DDIN)
  then F_LAB140 $:= 1$; **call** LAB170 **fi**;
if WLAST $\neq$ WREC.WORD
  then **call** LAB170 **fi**

Absorb:

if end_of_file?(DDIN)
  then F_LAB140 $:= 1$; **call** LAB170
elsif WLAST $\neq$ WREC.WORD
    then **call** LAB170 **fi**

Join Cases:

if end_of_file?(DDIN) $\vee$ WLAST $\neq$ WREC.WORD
  then **if** end_of_file?(DDIN)
      then F_LAB140 $:= 1$ **fi**;
    call LAB170 **fi**

The General Induction Rule

If $\mathbf{S}$ is any statement with bounded nondeterminacy, and $\mathbf{S}'$ is another statement such that

$$\Delta \vdash \mathbf{S}^n \ \leq \ \mathbf{S}'$$

for all $n < \omega$, then:

$$\Delta \vdash \mathbf{S} \ \leq \ \mathbf{S}'$$

Here, "bounded nondeterminacy" means that in each specification statement there is a finite number of possible values for the assigned variables.

Loop Merging

If **S** is any statement and $\mathbf{B}_1$ and $\mathbf{B}_2$ are any formulae such that $\mathbf{B}_1 \Rightarrow \mathbf{B}_2$ then:

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

is equivalent to:

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

General Recursion Removal

Suppose we have a recursive procedure whose body is a regular action system in the following form:

proc $F(x) \ \equiv$
  actions A_1 :
  $\ldots A_i \ \equiv$ $\mathbf{S}_i$.
  $\ldots B_j \ \equiv$ $\mathbf{S}_{j0}$; $F(g_{j1}(x))$; $\mathbf{S}_{j1}$; $F(g_{j2}(x))$;
      $\ldots$; $F(g_{jn_j}(x))$; $\mathbf{S}_{jn_j}$.
  $\ldots$ **endactions.**

where $\mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j}$ preserve the value of x and no $\mathbf{S}$ contains a call to F (i.e. all the calls to F are listed explicitly in the B_j actions) and the statements $\mathbf{S}_{j0}$, $\mathbf{S}_{j1}$, $\ldots$, $\mathbf{S}_{jn_j-1}$ contain no action calls.

General Recursion Removal

$$
\begin{array}{l}
\textbf{proc } F'(x) \equiv \\
\quad \textbf{var } L := \langle\rangle, m := 0 : \\
\qquad \textbf{actions } A_1 : \\
\qquad \ldots A_i \equiv \textbf{S}_i[\textbf{call } \hat{F}/\textbf{call } Z]. \\
\qquad \ldots B_j \equiv \textbf{S}_{j0}; \\
\qquad\qquad\qquad L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \\
\qquad\qquad\qquad\qquad \ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!\!+} L; \\
\qquad\qquad\qquad \textbf{call } \hat{F}. \\
\qquad \ldots \hat{F} \equiv \textbf{if } L = \langle\rangle \\
\qquad\qquad\qquad \textbf{then call } Z \\
\qquad\qquad\qquad \textbf{else } \langle m, x\rangle \xleftarrow{\text{pop}} L; \\
\qquad\qquad\qquad\qquad \textbf{if } m = 0 \rightarrow \textbf{call } A_1 \\
\qquad\qquad\qquad\qquad \square \ldots \square\ m = \langle j, k\rangle \\
\qquad\qquad\qquad\qquad\quad \rightarrow \textbf{S}_{jk}[\textbf{call } \hat{F}/\textbf{call } Z];\ \textbf{call } \hat{F} \\
\qquad\qquad\qquad\qquad \ldots \textbf{ fi fi. endactions end.}
\end{array}
$$

Recursive Implementation Theorem

Suppose we have a statement $\mathbf{S}'$ which we wish to transform into the recursive procedure $(\mu X.\mathbf{S})$. This is possible whenever:

Recursive Implementation Theorem

Suppose we have a statement $\mathbf{S}'$ which we wish to transform into the recursive procedure $(\mu X.\mathbf{S})$. This is possible whenever:

1. The statement $\mathbf{S}'$ is refined by $\mathbf{S}[\mathbf{S}'/X]$. In other words, if we replace recursive calls in $\mathbf{S}$ by copies of $\mathbf{S}'$ then we get a refinement of $\mathbf{S}'$; and

Recursive Implementation Theorem

Suppose we have a statement $\mathbf{S}'$ which we wish to transform into the recursive procedure $(\mu X.\mathbf{S})$. This is possible whenever:

1. The statement $\mathbf{S}'$ is refined by $\mathbf{S}[\mathbf{S}'/X]$. In other words, if we replace recursive calls in $\mathbf{S}$ by copies of $\mathbf{S}'$ then we get a refinement of $\mathbf{S}'$; and

2. We can find an expression $\mathbf{t}$ (called the *variant function*) whose value is reduced before each occurrence of $\mathbf{S}'$ in $\mathbf{S}[\mathbf{S}'/X]$.

Recursive Implementation Theorem

Suppose we have a statement $\mathbf{S}'$ which we wish to transform into the recursive procedure $(\mu X.\mathbf{S})$. This is possible whenever:

1. The statement $\mathbf{S}'$ is refined by $\mathbf{S}[\mathbf{S}'/X]$. In other words, if we replace recursive calls in $\mathbf{S}$ by copies of $\mathbf{S}'$ then we get a refinement of $\mathbf{S}'$; and

2. We can find an expression $\mathbf{t}$ (called the *variant function*) whose value is reduced before each occurrence of $\mathbf{S}'$ in $\mathbf{S}[\mathbf{S}'/X]$.

If both these conditions are satisfied, then:

$$\Delta \vdash \mathbf{S}' \ \leq \ (\mu X.\mathbf{S})$$

Recursive Implementation

Recursive Implementation

1. Start with a specification: SPEC

Recursive Implementation

1. Start with a specification: SPEC

2. Transform to a program containing copies of the specification:

$$\mathsf{SPEC} \approx \ldots\mathsf{SPEC}\ldots\mathsf{SPEC}\ldots\mathsf{SPEC}\ldots$$

Recursive Implementation

1. Start with a specification: SPEC
2. Transform to a program containing copies of the specification:

$$\mathsf{SPEC} \;\approx\; \ldots\mathsf{SPEC}\ldots\mathsf{SPEC}\ldots\mathsf{SPEC}\ldots$$

3. Show that the variant expression is reduced before each copy:

$$\mathsf{SPEC} \;\approx\; \ldots\{\mathbf{t} < t_0\};\ \mathsf{SPEC}\ldots\{\mathbf{t} < t_0\};\ \mathsf{SPEC}\ldots\{\mathbf{t} < t_0\};\ \mathsf{SPEC}\ldots$$

Recursive Implementation

1. Start with a specification: SPEC
2. Transform to a program containing copies of the specification:

$$\mathsf{SPEC} \;\approx\; \ldots\mathsf{SPEC}\ldots\mathsf{SPEC}\ldots\mathsf{SPEC}\ldots$$

3. Show that the variant expression is reduced before each copy:

$$\mathsf{SPEC} \;\approx\; \ldots\{\mathbf{t} < t_0\};\ \mathsf{SPEC}\ldots\{\mathbf{t} < t_0\};\ \mathsf{SPEC}\ldots\{\mathbf{t} < t_0\};\ \mathsf{SPEC}\ldots$$

4. Apply the Recursive Implementation transformation to get a recursive procedure:

$$\mathsf{SPEC} \;\approx\; (\mu X.\ldots\{\mathbf{t} < t_0\};\ X\ldots\{\mathbf{t} < t_0\};\ X\ldots\{\mathbf{t} < t_0\};\ X\ldots)$$

Recursive Implementation

1. Start with a specification: SPEC
2. Transform to a program containing copies of the specification:

$$\mathsf{SPEC} \;\approx\; \ldots \mathsf{SPEC} \ldots \mathsf{SPEC} \ldots \mathsf{SPEC} \ldots$$

3. Show that the variant expression is reduced before each copy:

$$\mathsf{SPEC} \;\approx\; \ldots \{\mathbf{t} < t_0\};\ \mathsf{SPEC} \ldots \{\mathbf{t} < t_0\};\ \mathsf{SPEC} \ldots \{\mathbf{t} < t_0\};\ \mathsf{SPEC} \ldots$$

4. Apply the Recursive Implementation transformation to get a recursive procedure:

$$\mathsf{SPEC} \;\approx\; (\mu X. \ldots \{\mathbf{t} < t_0\};\ X \ldots \{\mathbf{t} < t_0\};\ X \ldots \{\mathbf{t} < t_0\};\ X \ldots)$$

5. If necessary, apply Recursion Removal to get an iterative procedure.

Refinement Example

Suppose we want to develop a factorial program.

Refinement Example

Suppose we want to develop a factorial program.

The specification is very simple.

Define SPEC to be the statement:

$$y := n!$$

where n is a non-negative integer.

Refinement Example

Suppose we want to develop a factorial program.

The specification is very simple.

Define SPEC to be the statement:

$$y := n!$$

where n is a non-negative integer.

Transform this into an **if** statement:

$$\textbf{if } n = 0 \textbf{ then } y := n! \textbf{ else } y := n! \textbf{ fi}$$

Refinement Example

Suppose we want to develop a factorial program.

The specification is very simple.

Define SPEC to be the statement:

$$y := n!$$

where n is a non-negative integer.

Transform this into an **if** statement:

$$\textbf{if } n = 0 \textbf{ then } y := n! \textbf{ else } y := n! \textbf{ fi}$$

When $n = 0$, we know that $n! = 1$, so:

$$\textbf{if } n = 0 \textbf{ then } y := 1 \textbf{ else } y := n! \textbf{ fi}$$

Refinement Example

If $n > 0$ then $n! = n.(n-1)!$, so:

Refinement Example

If $n > 0$ then $n! = n.(n-1)!$, so:

$$\begin{aligned} y := n! \quad &\approx \quad y := n.(n-1)! \\ &\approx \quad y := (n-1)!;\ y := n.y \\ &\approx \quad n := n-1;\ y := n!;\ n := n+1;\ y := n.y \end{aligned}$$

Refinement Example

If $n > 0$ then $n! = n.(n-1)!$, so:

$$
\begin{aligned}
y := n! \quad &\approx \quad y := n.(n-1)! \\
&\approx \quad y := (n-1)!;\ y := n.y \\
&\approx \quad n := n-1;\ y := n!;\ n := n+1;\ y := n.y
\end{aligned}
$$

The specification has been transformed as follows:

SPEC $\approx$ **if** $n = 0$
then $y := 1$
else $n := n-1$; SPEC; $n := n+1$; $y := n.y$ **fi**

Note that n is reduced before the copy of SPEC on the right.

Refinement Example

Apply the Recursive Implementation Theorem:

$$\begin{aligned}\text{SPEC} \approx \textbf{proc } F() \equiv \textbf{ if } & n = 0 \\ \textbf{then } & y := 1 \\ \textbf{else } & n := n - 1; \\ & F(); \\ & n := n + 1; \\ & y := n.y \textbf{ fi end}\end{aligned}$$

This is an executable implementation of SPEC.

Refinement Example

Apply Recursion Removal:

$$\begin{aligned}
\text{SPEC} \approx\ & \textbf{var}\ \langle i := 0 \rangle : \\
& \quad \textbf{while}\ n \neq 0\ \textbf{do} \\
& \qquad i := i + 1;\ n := n - 1\ \textbf{od}; \\
& \quad y := 1; \\
& \quad \textbf{while}\ i > 0\ \textbf{do} \\
& \qquad i := i - 1;\ n;\ = n + 1;\ y := n.y\ \textbf{od}\ \textbf{end}
\end{aligned}$$

(Here, i represents the number of recursive calls still pending.)

Refinement Example

Simplify:

SPEC $\approx$ **var** $\langle i := n \rangle$:

$n := 0;\ y := 1;$

while $i > 0$ **do**

$i := i - 1;\ n;\ = n + 1;\ y := n.y$ **od end**

Refinement Example

Simplify:

$$\text{SPEC} \approx \textbf{var } \langle i := n \rangle : \\ n := 0;\ y := 1; \\ \textbf{while } i > 0 \textbf{ do} \\ i := i - 1;\ n;\ = n + 1;\ y := n.y \textbf{ od end}$$

Let $j = n - i + 1$ and simplify:

$$\text{SPEC} \approx y := 1; \\ \textbf{for } j := 1 \textbf{ to } n \textbf{ step } 1 \\ y := j.y \textbf{ od end}$$

Refinement Example

Simplify:

SPEC $\approx$ **var** $\langle i := n \rangle$:

$n := 0;\ y := 1;$

while $i > 0$ **do**

$i := i - 1;\ n;\ = n + 1;\ y := n.y$ **od end**

Let $j = n - i + 1$ and simplify:

SPEC $\approx$ $y := 1;$

for $j := 1$ **to** n **step** 1

$y := j.y$ **od end**

A long-winded process for such a simple specification.

Refinement Example

Simplify:

SPEC $\approx$ **var** $\langle i := n \rangle$:

$n := 0;\ y := 1;$

while $i > 0$ **do**

$i := i - 1;\ n;\ = n + 1;\ y := n.y$ **od end**

Let $j = n - i + 1$ and simplify:

SPEC $\approx$ $y := 1;$

for $j := 1$ **to** n **step** 1

$y := j.y$ **od end**

A long-winded process for such a simple specification.

But the transformations apply to *any* recursive procedure!

Sorting Example

Specification of a sorting program SORT(a, b) is:

$$A[a..b] := A'[a..b].(\mathsf{sorted}(A'[a..b]) \ \wedge \ \mathsf{permutation\_of}(A'[a..b], A[a..b]))$$

If $a \geqslant b$ then $A[a..b]$ is already sorted.

Otherwise, permute the elements of A so that there is an element $A[p]$ such that:

$$A[a..p-1] \leqslant A[p] \leqslant A[p+1..b]$$

Define the specification partition as:

$$\begin{aligned}\langle A[a..b], p\rangle := \langle A'[a..b], p'\rangle.(a \leqslant p \leqslant b \\ &\wedge \ A'[a..p-1] \leqslant A'[p] \leqslant A'[p+1..b] \\ &\wedge \ \mathsf{permutation\_of}(A'[a..b], A[a..b]))\end{aligned}$$

Sorting Example

Now $\mathsf{SORT}(a, b) \ \approx$

var $\langle p := 0 \rangle$:

  if $b > a$ **then** partition;

   $\mathsf{SORT}(a, p - 1)$;

   $\mathsf{SORT}(p + 1, b)$ **fi**

Apply Recursion Introduction to get the *quicksort* algorithm:

proc $\mathsf{qsort}(a, b) \ \equiv$

  var $\langle p := 0 \rangle$:

   if $b > a$ **then** partition;

    $\mathsf{qsort}(a, p - 1)$;

    $\mathsf{qsort}(p + 1, b)$ **fi**

Loop Unrolling

while B **do**

  if B_1 **then** S_1

  elsif $\ldots$

  elsif B_i **then** S_i

  $\ldots$

        else S_n **fi od**

Unroll one step of the loop:

while B **do**

  if B_1 **then** S_1

  elsif $\ldots$

  elsif B_i **then** S_i; **if** B $\wedge$ Q **then if** B_1 **then** $\ldots$ **fi fi**

  $\ldots$

        else S_n **fi od**

We can unroll simultaneously at multiple terminal positions.

Entire Loop Unrolling

while B do

  if B$_1$ **then S**$_1$

  elsif ...

  elsif B$_i$ **then S**$_i$

  ...

        else S$_n$ **fi od**

Unroll multiple loop steps:

while B do

  if B$_1$ **then S**$_1$

  elsif ...

  elsif B$_i$ **then S**$_i$; **while B** $\wedge$ **Q do if B**$_1$ **then** ... **fi od**

  ...

        else S$_n$ **fi od**

We can unroll simultaneously at multiple terminal positions.

Entire Loop Unrolling

For example, let $\mathbf{Q} = \mathbf{B}_i$, and assume that the $\mathbf{B}_i$ are disjoint:

while $\mathbf{B}$ **do**

    if $\mathbf{B}_1$ **then** $\mathbf{S}_1$

    elsif ...

    elsif $\mathbf{B}_i$ **then** $\mathbf{S}_i$

    ...

            else $\mathbf{S}_n$ **fi od**

becomes:

while $\mathbf{B}$ **do**

    if $\mathbf{B}_1$ **then** $\mathbf{S}_1$

    elsif ...

    elsif $\mathbf{B}_i$ **then while** $\mathbf{B} \wedge \mathbf{B}_i$ **do** $\mathbf{S}_i$ **od**

    ...

            else $\mathbf{S}_n$ **fi od**

Algorithm Derivation

Suppose we want to develop an integer exponentiation algorithm.

Algorithm Derivation

Suppose we want to develop an integer exponentiation algorithm.

The specification is very simple:

$$\mathsf{EXP}(x, n) \;=_{\mathsf{DF}}\; y := x^n$$

where n is a non-negative integer.

Algorithm Derivation

Suppose we want to develop an integer exponentiation algorithm.

The specification is very simple:

$$\mathsf{EXP}(x, n) \;=_{\mathsf{DF}}\; y := x^n$$

where n is a non-negative integer.

Our derivation uses the following facts about exponentiation:

Algorithm Derivation

Suppose we want to develop an integer exponentiation algorithm.

The specification is very simple:

$$\mathsf{EXP}(x, n) \;=_{\mathsf{DF}}\; y := x^n$$

where n is a non-negative integer.

Our derivation uses the following facts about exponentiation:

1. $x^0 = 1$ for all x;

Algorithm Derivation

Suppose we want to develop an integer exponentiation algorithm.

The specification is very simple:

$$\mathsf{EXP}(x, n) \ =_{\mathsf{DF}} \ y := x^n$$

where n is a non-negative integer.

Our derivation uses the following facts about exponentiation:

1. $x^0 = 1$ for all x;
2. $x^{2n} = (x * x)^n$ and;

Algorithm Derivation

Suppose we want to develop an integer exponentiation algorithm.

The specification is very simple:

$$\mathsf{EXP}(x, n) \;=_{\text{DF}}\; y := x^n$$

where n is a non-negative integer.

Our derivation uses the following facts about exponentiation:

1. $x^0 = 1$ for all x;
2. $x^{2n} = (x * x)^n$ and;
3. $x^{n+1} = x * x^n$

Algorithm Derivation

Apply Splitting_A_Tautology and Insert_Assertions:

$$\begin{aligned}\mathsf{EXP}(x,n) \;\approx\; & \textbf{if } n = 0 \textbf{ then } \{n = 0\};\ \mathsf{EXP}(x,n) \\ & \textbf{elsif } \mathsf{even?}(x) \textbf{ then } \{n > 0 \wedge \mathsf{even?}(n)\};\ \mathsf{EXP}(x,n) \\ & \qquad\qquad\qquad\ \textbf{else } \{n > 0 \wedge \mathsf{odd?}(n)\};\ \mathsf{EXP}(x,n) \textbf{ fi}\end{aligned}$$

Algorithm Derivation

Apply Splitting_A_Tautology and Insert_Assertions:

$$\begin{aligned}\text{EXP}(x,n) \;\approx\; & \textbf{if } n=0 \textbf{ then } \{n=0\};\ \text{EXP}(x,n)\\ & \textbf{elsif } \text{even?}(x) \textbf{ then } \{n>0 \wedge \text{even?}(n)\};\ \text{EXP}(x,n)\\ & \qquad\qquad\qquad \textbf{else } \{n>0 \wedge \text{odd?}(n)\};\ \text{EXP}(x,n) \textbf{ fi}\end{aligned}$$

Use the assertions to refine each copy of $\text{EXP}(x,n)$:

$$\begin{aligned}& \textbf{if } n=0 \textbf{ then } y:=1\\ & \textbf{elsif } \text{even?}(n) \textbf{ then } \{n>0 \wedge \text{even?}(n)\};\\ & \qquad\qquad\qquad\qquad \text{EXP}(x*x, n/2)\\ & \qquad\qquad\qquad \textbf{else } \{n>0 \wedge \text{odd?}(n)\};\\ & \qquad\qquad\qquad\qquad \text{EXP}(x, n-1);\ y:=x*y \textbf{ fi}\end{aligned}$$

Algorithm Derivation

Apply Splitting_A_Tautology and Insert_Assertions:

$$\begin{aligned}\mathsf{EXP}(x,n) \;\approx\; & \textbf{if } n=0 \textbf{ then } \{n=0\};\ \mathsf{EXP}(x,n) \\ & \textbf{elsif } \mathsf{even?}(x) \textbf{ then } \{n>0 \wedge \mathsf{even?}(n)\};\ \mathsf{EXP}(x,n) \\ & \qquad\qquad\quad \textbf{else } \{n>0 \wedge \mathsf{odd?}(n)\};\ \mathsf{EXP}(x,n) \textbf{ fi}\end{aligned}$$

Use the assertions to refine each copy of $\mathsf{EXP}(x,n)$:

$$\begin{aligned}& \textbf{if } n=0 \textbf{ then } y:=1 \\ & \textbf{elsif } \mathsf{even?}(n) \textbf{ then } \{n>0 \wedge \mathsf{even?}(n)\}; \\ & \qquad\qquad\qquad\quad \mathsf{EXP}(x*x, n/2) \\ & \qquad\qquad\quad\;\; \textbf{else } \{n>0 \wedge \mathsf{odd?}(n)\}; \\ & \qquad\qquad\qquad\quad \mathsf{EXP}(x, n-1);\ y := x*y \textbf{ fi}\end{aligned}$$

This is the **elaborated specification**

Algorithm Derivation

Apply the Recursive Implementation Theorem:

proc exp(x, n) $\equiv$

  if $n = 0$ **then** $y := 1$

  elsif even?(n) **then** exp$(x * x, n/2)$

        else exp$(x, n - 1)$; $y := x * y$ **fi.**

This is now an executable, recursive implementation of the specification EXP(x, n)

Algorithm Derivation

Replace parameter n by a global variable:

$$
\begin{array}{l}
\textbf{proc } \mathsf{exp}(x,n) \;\equiv\; \mathsf{exp1}(x)\textbf{.} \\
\textbf{proc } \mathsf{exp1}(x) \;\equiv \\
\quad \textbf{if } n=0 \textbf{ then } y:=1 \\
\quad \textbf{elsif } \mathsf{even?}(n) \textbf{ then } n:=n/2;\; \mathsf{exp1}(x*x) \\
\qquad\qquad\qquad\qquad \textbf{else } n:=n-1;\; \mathsf{exp1}(x);\; y:=x*y \textbf{ fi.}
\end{array}
$$

Apply Recursion Removal to exp1:

$$
\begin{array}{l}
\textbf{proc } \mathsf{exp1}(x) \;\equiv \\
\quad \textbf{var } \langle L:=\langle\rangle\rangle : \\
\qquad \textbf{actions } A : \\
\qquad A \;\equiv \textbf{if } n=0 \textbf{ then } y:=1;\; \textbf{call } \hat{F} \\
\qquad\qquad \textbf{elsif } \mathsf{even?}(n) \textbf{ then } n:=n/2;\; x:=x*x;\; \textbf{call } A \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } n:=n-1;\; L \xleftarrow{\mathsf{push}} x;\; \textbf{call } A \textbf{ fi.} \\
\qquad \hat{F} \;\equiv \textbf{if } L=\langle\rangle \textbf{ then call } Z \\
\qquad\qquad\qquad\qquad \textbf{else } x \xleftarrow{\mathsf{pop}} L;\; y:=x*y;\; \textbf{call } \hat{F} \textbf{ fi. endactions end.}
\end{array}
$$

Algorithm Derivation

Restructure the regular action system:

$$
\begin{array}{l}
\textbf{proc } \mathsf{exp}(x, n) \ \equiv \\
\quad \textbf{var } \langle L := \langle\rangle\rangle : \\
\qquad \textbf{while } n \neq 0 \textbf{ do} \\
\qquad\quad \textbf{if } \mathsf{even?}(n) \textbf{ then } x := x * x;\ n := n/2 \\
\qquad\qquad\qquad\qquad \textbf{else } n := n - 1;\ L \xleftarrow{\mathsf{push}} x \textbf{ fi od}; \\
\qquad y := 1; \\
\qquad \textbf{while } L \neq \langle\rangle \textbf{ do } x \xleftarrow{\mathsf{pop}} L;\ y := x * y \textbf{ od.}
\end{array}
$$

Apply Entire Loop Unrolling after the assignment $n := n/2$ with the condition $n \neq 0 \wedge \mathsf{even?}(n)$:

Algorithm Derivation

$$
\begin{array}{l}
\textbf{proc } \mathsf{exp}(x, n) \ \equiv \\
\quad \textbf{var } \langle L := \langle\rangle\rangle : \\
\quad\quad \textbf{while } n \neq 0 \textbf{ do} \\
\quad\quad\quad \textbf{if } \mathsf{even?}(n) \textbf{ then } x := x * x;\ n := n/2; \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{while } n \neq 0 \ \wedge\ \mathsf{even?}(n) \textbf{ do} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{if } \mathsf{even?}(n) \textbf{ then } x := x * x;\ n := n/2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else } n := n - 1;\ L \xleftarrow{\mathsf{push}} x \textbf{ fi od}; \\
\quad\quad\quad\quad\quad\quad \textbf{else } n := n - 1;\ L \xleftarrow{\mathsf{push}} x \textbf{ fi od}; \\
\quad\quad y := 1; \\
\quad\quad \textbf{while } L \neq \langle\rangle \textbf{ do } x \xleftarrow{\mathsf{pop}} L;\ y := x * y \textbf{ od.}
\end{array}
$$

Algorithm Derivation

Simplify:

$$
\begin{array}{l}
\textbf{proc } \mathsf{exp}(x, n) \ \equiv \\
\quad \textbf{var } \langle L := \langle\rangle\rangle : \\
\qquad \textbf{while } n \neq 0 \textbf{ do} \\
\qquad\quad \textbf{if } \mathsf{even?}(n) \textbf{ then while } \mathsf{even?}(n) \textbf{ do } x := x * x;\ n := n/2 \textbf{ od} \\
\qquad\qquad\qquad\qquad \textbf{else } n := n - 1;\ L \xleftarrow{\mathsf{push}} x \textbf{ fi od}; \\
\qquad y := 1; \\
\qquad \textbf{while } L \neq \langle\rangle \textbf{ do } x \xleftarrow{\mathsf{pop}} L;\ y := x * y \textbf{ od.}
\end{array}
$$

Unroll a step after the inner **while** loop:

Algorithm Derivation

$$
\begin{array}{l}
\textbf{proc } \mathrm{exp}(x, n) \ \equiv \\
\quad \textbf{var } \langle L := \langle\rangle\rangle : \\
\qquad \textbf{while } n \neq 0 \textbf{ do} \\
\qquad\quad \textbf{if } \mathrm{even?}(n) \textbf{ then while } \mathrm{even?}(n) \textbf{ do } x := x * x;\ n := n/2 \textbf{ od}; \\
\qquad\qquad\qquad\qquad\qquad\qquad L \xleftarrow{\text{push}} x;\ n := n - 1 \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } L \xleftarrow{\text{push}} x;\ n := n - 1 \textbf{ fi od}; \\
\qquad y := 1; \\
\qquad \textbf{while } L \neq \langle\rangle \textbf{ do } x \xleftarrow{\text{pop}} L;\ y := x * y \textbf{ od.}
\end{array}
$$

Separate common code out of the **if** statement. The test is now redundant, since the inner **while** loop is equivalent to **skip** when n is odd:

Algorithm Derivation

$$
\begin{array}{l}
\textbf{proc } \mathsf{exp}(x, n) \;\equiv \\
\quad \textbf{var } \langle L := \langle\rangle\rangle : \\
\quad\quad \textbf{while } n \neq 0 \textbf{ do} \\
\quad\quad\quad \textbf{while } \mathsf{even?}(n) \textbf{ do } x := x * x;\; n := n/2 \textbf{ od}; \\
\quad\quad\quad n := n - 1;\; L \xleftarrow{\mathsf{push}} x \textbf{ od}; \\
\quad\quad y := 1; \\
\quad\quad \textbf{while } L \neq \langle\rangle \textbf{ do } x \xleftarrow{\mathsf{pop}} L;\; y := x * y \textbf{ od.}
\end{array}
$$

If we move the assignment $y := 1$ to the front, then we can merge the bodies of the two **while** loops.

Note: The order of execution of the statements in the second **while** loop is reversed.

Algorithm Derivation

proc exp(x, n) $\equiv$

  var $\langle L := \langle\rangle\rangle$:

    $y := 1;$

    while $n \neq 0$ **do**

      while even?(n) **do** $x := x * x;\ n := n/2$ **od**;

      $n := n - 1;\ L \xleftarrow{\text{push}} x;$

      $x \xleftarrow{\text{pop}} L;\ y := x * y$ **od.**

Local variable L is now redundant, since $L \xleftarrow{\text{push}} x;\ L \xleftarrow{\text{pop}} x\ \approx$ **skip**:

proc exp(x, n) $\equiv$

  $y := 1;$

  while $n \neq 0$ **do**

    while even?(n) **do** $x := x * x;\ n := n/2$ **od**;

    $n := n - 1;\ y := x * y$ **od.**

Classes of Transformations

- Simplify
- Move
- Delete
- Join
- Reorder/Separate
- Rewrite
- Use/Apply
- Abstraction
- Refinement

Classes of Transformations

- Simplify: The selected item is transformed into simpler code
  - Eg: Simplify, Delete_Comments, Fix_Assembler, Reduce_Loop, Flag_Removal, Syntactic_Slice

Classes of Transformations

- Simplify: The selected item is transformed into simpler code
  - Eg: Simplify, Delete_Comments, Fix_Assembler, Reduce_Loop, Flag_Removal, Syntactic_Slice
- Move: The selected item is moved (and remains selected)
  - Eg: Move_To_Left, Move_To_Right, Take_Out_Of_Loop

Classes of Transformations

- Simplify: The selected item is transformed into simpler code
  - Eg: Simplify, Delete_Comments, Fix_Assembler, Reduce_Loop, Flag_Removal, Syntactic_Slice
- Move: The selected item is moved (and remains selected)
  - Eg: Move_To_Left, Move_To_Right, Take_Out_Of_Loop
- Delete: The selected item is deleted, or parts of the item are deleted
  - Eg: Delete_Item, Delete_All_Assertions, Delete_All_Redundant, Delete_All_Skips

Classes of Transformations

- Join: Items are absorbed into or combined with the selected item, or the selected item is merged into some other item
  - Eg: Absorb_Left, Absorb_Right, Merge_Left, Merge_Right, Expand_Forward, Join_All_Cases

Classes of Transformations

- Join: Items are absorbed into or combined with the selected item, or the selected item is merged into some other item
  - Eg: Absorb_Left, Absorb_Right, Merge_Left, Merge_Right, Expand_Forward, Join_All_Cases
- Reorder/Separate: The order of components in the selected item is changed, or code is taken out of the item
  - Eg: Reverse_Order, Separate_Exit_Code, Separate_Left, Separate_Right

Classes of Transformations

- Join: Items are absorbed into or combined with the selected item, or the selected item is merged into some other item
  - Eg: Absorb_Left, Absorb_Right, Merge_Left, Merge_Right, Expand_Forward, Join_All_Cases
- Reorder/Separate: The order of components in the selected item is changed, or code is taken out of the item
  - Eg: Reverse_Order, Separate_Exit_Code, Separate_Left, Separate_Right
- Rewrite: The selected item is transformed in some way, with surrounding code unchanged.
  - Eg: Collapse_Action_System, Else_If_To_Elsif, Elsif_To_Else_If, Floop_To_While, Combine_Wheres, Replace_With_Value, While_To_Floop, Double_To_Single_Loop

Classes of Transformations

- Use/Apply: The selected item (eg an assertion) is used to transform later code. For example, the fact that an assertion appears at this point is used to simplify subsequent tests
  - Eg: Apply_To_Right, Delete_What_Follows, Use_Assertion

Classes of Transformations

- Use/Apply: The selected item (eg an assertion) is used to transform later code. For example, the fact that an assertion appears at this point is used to simplify subsequent tests
  - Eg: Apply_To_Right, Delete_What_Follows, Use_Assertion
- Abstraction: This transformation is informally an abstraction operation: eg replacing a statement by an equivalent specification statement
  - Eg: Prog_to_Spec, Raise_Abstraction

Classes of Transformations

- Use/Apply: The selected item (eg an assertion) is used to transform later code. For example, the fact that an assertion appears at this point is used to simplify subsequent tests
  - Eg: Apply_To_Right, Delete_What_Follows, Use_Assertion
- Abstraction: This transformation is informally an abstraction operation: eg replacing a statement by an equivalent specification statement
  - Eg: Prog_to_Spec, Raise_Abstraction
- Refinement: This transformation is informally a refinement operation: eg refining a specification statement into an equivalent statement
  - Eg: Refine_Spec