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 $S_1$, $S_2$ and $S_3$:

\[
\text{if } B \text{ then } S_1 \\
\text{else } S_2 \text{ fi; } \\
S_3
\]

is equivalent to:

\[
\text{if } B \text{ then } S_1; S_3 \\
\text{else } S_2; S_3 \text{ fi}
\]
A Syntactic Transformation

For any condition (formula) $B$ and any statements $S_1$, $S_2$ and $S_3$:

```
if B then $S_1$
else $S_2$ fi;
$S_3$
```

is equivalent to:

```
if B then $S_1$; $S_3$
else $S_2$; $S_3$ fi
```

In FermaT this result can be produced by applying Absorb_Right or Expand_Forwards on the `if` statement, or Merge_Left on $S_3$.
Another Example

If $S_3$ does not modify any of the variables in $B$ then:

$$S_3;$$
$$\text{if } B \text{ then } S_1$$
$$\text{else } S_2 \text{ fi;} S_3$$

is equivalent to:

$$\text{if } B \text{ then } S_3; S_2$$
$$\text{else } S_3; S_1 \text{ fi}$$
Another Example

If $S_3$ does not modify any of the variables in $B$ then:

$$S_3;$$
$$\text{if } B \text{ then } S_1$$
$$\text{else } S_2 \text{ fi; } S_3$$

is equivalent to:

$$\text{if } B \text{ then } S_3; S_2$$
$$\text{else } S_3; S_1 \text{ fi}$$

In FermaT this result can be produced by applying Absorb_Left on the if statement, or Merge_Right on $S_3$. 
Splitting A Tautology

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

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

Adding Assertions:

$$ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } $$

is equivalent to:

$$ \text{if } B \text{ then } \{B\}; S_1 \text{ else } \{\neg B\}; S_2 \text{ fi } $$
Splitting A Tautology

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

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

Adding Assertions:

$$\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}$$

is equivalent to:

$$\text{if } B \text{ then } \{B\}; S_1 \text{ else } \{\neg B\}; S_2 \text{ fi}$$

Assertions can be introduced and propagated through the program.
Adding Assertions

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

```
while B do S od
```

is equivalent to:

```
while B do {B}; S od; {¬B}
```
A Semantic Transformation

Assignment Merging: (Merge_Left and Merge_Right on assignments)

\[ x := 2 \times x; \quad x := x + 1 \]

is equivalent to:

\[ x := 2 \times x + 1 \]

Another example:

\[ y := n \times x \]

is equivalent to:

\[ n := n - 1; \quad y := (n + 1) \times x; \quad n := n + 1 \]
Example Transformations

\[
\text{if } n = 0 \text{ then } x := 1 \\
\quad \text{else } x := x + 1 \text{ fi;}
\]

\[
x := 2 * x
\]
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
if \( n = 0 \) then \( x := 1 \)
    else \( x := x + 1 \) fi;
\( x := 2 \times x \)

Expand the if statement:
if \( n = 0 \) then \( x := 1; \ x := 2 \times x \)
    else \( x := x + 1; \ x := 2 \times x \) fi

Merge the assignments:
if \( n = 0 \) then \( x := 2 \)
    else \( x := 2 \times (x + 1) \) fi
Expanding a Call

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

\[
\begin{align*}
\text{actions } A_1 : \\
A_1 & \equiv S_1 \text{ end} \\
\ldots \\
A_1 & \equiv \ldots \text{ call } A_j \ldots \text{ end} \\
\ldots \\
A_n & \equiv S_n \text{ end endactions}
\end{align*}
\]
Expanding a Call

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

\[
\text{actions } A_1 : \\
A_1 \equiv S_1 \text{ end} \\
... \\
A_1 \equiv ... [S_j] ... \text{ end} \\
... \\
A_n \equiv S_n \text{ end endactions}
\]
Expanding a Call

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

\[
\text{actions } A_1 :
\]
\[
A_1 \equiv S_1 \text{ end}
\]
\[
\ldots
\]
\[
A_1 \equiv \ldots [S_j] \ldots \text{ end}
\]
\[
\ldots
\]
\[
A_n \equiv S_n \text{ end endactions}
\]

If there are no other calls to \( A_j \), then the action can be deleted.
Suppose we have this code in a *regular* action system:

```plaintext
if B then $S_1$; call $A$
    else $S_2$ fi;

call $A$
```
Expand and Separate

Suppose we have this code in a regular action system:

```plaintext
if B then $S_1$; call $A$
    else $S_2$ fi;
call $A$
```

Expand the `if`:

```plaintext
if B then $S_1$; call $A$; call $A$
    else $S_2$; call $A$ fi
```
Suppose we have this code in a *regular* action system:

```haskell
if B then S_1; call A
    else S_2 fi;
call A
```

Expand the `if`:

```haskell
if B then S_1; call A; call A
    else S_2; call A fi
```

Delete after the first `call`:

```haskell
if B then S_1; call A
    else S_2; call A fi
```
Suppose we have this code in a *regular* action system:

```plaintext
if B then S₁; call A
    else S₂ fi;

call A
```

Expand the `if`:

```plaintext
if B then S₁; call A; call A
    else S₂; call A fi
```

Delete after the first `call`:

```plaintext
if B then S₁; call A
    else S₂; call A fi
```

Separate:

```plaintext
if B then S₁
    else S₂ fi;

call A
```
Example:

\[
\text{if } n = 0 \text{ then } x := 1; \text{ call } A \\
\quad \text{else } y := 2 \text{ fi;}
\]

\text{call } A
Example:
\[
\text{if } n = 0 \text{ then } x := 1; \text{ call } A \\
\quad \text{else } y := 2 \text{ fi;}
\]
\textbf{call } A

Becomes:
\[
\text{if } n = 0 \text{ then } x := 1 \\
\quad \text{else } y := 2 \text{ fi;}
\]
\textbf{call } A

The first \textbf{call } A has been deleted.
Example Transformations

Forward Expansion:

\[
\text{if } x = 1 \text{ then if } y = 1 \text{ then } z := 1 \text{ else } z := 2 \text{ fi else } z := 3 \text{ fi; if } z = 1 \text{ then } p := q \text{ fi}
\]

is equivalent to:

\[
\text{if } x = 1 \text{ then if } y = 1 \text{ then } z := 1 \text{ else } z := 2 \text{ fi; if } z = 1 \text{ then } p := q \text{ fi else } z := 3; \text{ if } z = 1 \text{ then } p := q \text{ fi fi}
\]
Example Transformations

Absorb Right:

```plaintext
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:

```plaintext
if x = 1 then if y = 1 then z := 1;
    if z = 1 then p := q fi
    else z := 2;  
        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!
Absorb Left into a loop, before:

\[
\begin{align*}
\text{do} & \quad \text{do} \quad \text{if } i > n \text{ then } \text{exit(2)} \text{ fi; } \\
& \quad i := i + 1; \\
& \quad \text{if } A[i] = v \text{ then } \text{exit(1)} \text{ fi od; } \\
& \quad \text{last} := i; \\
& \quad \text{count} := \text{count} + 1; \\
& \quad \text{if } \text{count} > \text{limit} \text{ then } \text{exit(1)} \text{ fi od; } \\
& \quad \text{if } \text{count} > \text{limit} \text{ then } \text{PRINT(last)} \text{ fi }
\end{align*}
\]
Absorb Left into a loop, after:

\[
\begin{align*}
\textbf{do } \textbf{do } & \textbf{if } i > n \textbf{ then } \textbf{if } \text{count} > \text{limit} \textbf{ then PRINT(last); exit(2)} \\
& \textbf{else exit(2) fi fi;} \\
& \textbf{i := i + 1;} \\
& \textbf{if } A[i] = v \textbf{ then exit(1) fi od;} \\
\text{last} & := i; \\
\text{count} & := \text{count} + 1; \\
\textbf{if } \text{count} > \text{limit} & \textbf{ then } \textbf{if } \text{count} > \text{limit} \textbf{ then PRINT(last); exit(1)} \\
& \textbf{else exit(1) fi fi od;} 
\end{align*}
\]
Loop Inversion

\[
\textbf{do} \quad \text{Read\_A\_Record(file, record);} \\
\quad \text{if } \text{end\_of\_file?(file) then exit}(1) \text{ fi;} \\
\quad \text{Process\_Record(record) od}
\]
Loop Inversion

\[
\text{do}\quad \text{Read}\_\text{A}\_\text{Record}(\text{file}, \text{record}); \\
\quad \text{if}\ \text{end\_of\_file}(\text{file}) \text{ then } \text{exit}(1) \text{ fi}; \\
\quad \text{Process\_Record}(\text{record}) \quad \text{od}
\]

Is equivalent to:

\[
\text{Read}\_\text{A}\_\text{Record}(\text{file}, \text{record}); \\
\text{do}\quad \text{if}\ \text{end\_of\_file}(\text{file}) \text{ then } \text{exit}(1) \text{ fi}; \\
\quad \text{Process\_Record}(\text{record}); \\
\quad \text{Read}\_\text{A}\_\text{Record}(\text{file}, \text{record}) \quad \text{od}
\]
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:

\[
do \ S_1; \ S_2 \ od
\]

Is equivalent to:

\[
S_1; \ do \ S_2; \ S_1 \ od
\]

provided \( S_1 \) is a proper sequence (It has no exit statements which can leave an enclosing loop)
Loop Inversion

More Generally:

\[
\text{do } S_1; S_2 \text{ od}
\]

Is equivalent to:

\[
\text{do } S_1; \text{ do } S_2; S_1 \text{ od } + 1 \text{ od}
\]

where the +1 will increment the exit statements which terminate \text{do } S_2; S_1 \text{ od} so that they terminate the new outer loop.
Loop inversion can be used to merge two copies of a statement into one, for example:

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

simplifies to:
\textbf{do GET}(DDIN var WREC);
   \textbf{if} end_of_file?(DDIN) \textbf{then} exit(1) \textbf{fi;}
   WORKP := WREC.NUM;
   TOTAL := TOTAL + WORKP \textbf{od;}
Merging Copies

A program with repeated statements:

\[
\text{do } \ldots; \\
\text{if } \text{end_of_file}(\text{DDIN}) \\
\text{then exit}(1) \text{ fi}; \\
\text{PUT} \_\text{FIXED}(\text{RDSOUT}, \text{WPRT var result_code, os}); \\
\text{fill}(\text{WPRT[1] var WPRT[2..80]}) \od; \\
\text{PUT} \_\text{FIXED}(\text{RDSOUT}, \text{WPRT var result_code, os}); \\
\text{fill}(\text{WPRT[1] var WPRT[2..80]})
\]
Absorb into the loop:

\begin{verbatim}
  do . . .
    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;
\end{verbatim}
Absorb into the **if** statement:

```
  do . . .;
  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:

```plaintext
do ...;
    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 $S_2$ which we want to merge:

\[
\begin{align*}
\text{if } B_1 & \text{ then } S_1; S_2 \\
\text{elsif } B_2 & \text{ then } S_2 \\
\text{else } & S_3 \text{ fi}
\end{align*}
\]
Merging Copies

Here, there are two copies of $S_2$ which we want to merge:

\[
\text{if } B_1 \text{ then } S_1; \ S_2 \\
\text{elsif } B_2 \text{ then } S_2 \\
\quad \text{else } S_3 \ \text{fi}
\]

The result is:

\[
\text{if } B_1 \lor B_2 \\
\quad \text{then if } B_1 \text{ then } S_1 \ \text{fi;} \\
\quad S_2 \\
\quad \text{else } S_3 \ \text{fi}
\]
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) \lor WLAST \neq WREC.WORD
    then if end_of_file?(DDIN)
        then F_LAB140 := 1 fi;
    call LAB170 fi
The General Induction Rule

If $S$ is any statement with bounded nondeterminacy, and $S'$ is another statement such that

$$\Delta \vdash S^n \leq S'$$

for all $n < \omega$, then:

$$\Delta \vdash S \leq S'$$

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 $B_1$ and $B_2$ are any formulae such that $B_1 \Rightarrow B_2$ then:

```
while B_1 do S od;
while B_2 do S od
```

is equivalent to:

```
while 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:

\[
\text{proc } F(x) \equiv \\
\text{actions } A_1: \\
\ldots A_i \equiv S_i. \\
\ldots B_j \equiv S_{j0}; F(g_{j1}(x)); S_{j1}; F(g_{j2}(x)); \\
\ldots; F(g_{jn_j}(x)); S_{jn_j}. \\
\ldots \text{endactions.}
\]

where \( S_{j1}, \ldots, S_{jn_j} \) preserve the value of \( x \) and no \( S \) contains a call to \( F \) (i.e. all the calls to \( F \) are listed explicitly in the \( B_j \) actions) and the statements \( S_{j0}, S_{j1}, \ldots, S_{jn_j-1} \) contain no action calls.
 proc \( F'(x) \) ≡ 
 var \( L := \langle \rangle, m := 0 \): 
 actions \( A_1 \): 
 \( \ldots A_i \) ≡ \( S_i[\text{call } \hat{F}/\text{call } Z] \). 
 \( \ldots B_j \) ≡ \( S_{j0} \); 
 \( \ldots B_j \) ≡ \( S_{j0} \); 
 \( L := \langle \langle 0, g_{j1}(x) \rangle, \langle j, 1 \rangle, x \rangle, \langle 0, g_{j2}(x) \rangle, \ldots, \langle 0, g_{jn_j}(x) \rangle, \langle j, n_j \rangle, x \rangle \rangle \) + L; 
 call \( \hat{F} \). 
 \( \ldots \hat{F} \) ≡ if \( L = \langle \rangle \) then call \( Z \) 
 else \( \langle m, x \rangle \leftarrow \text{pop} L \); 
 \( \ldots \) fi fi. endactions end.
Recursive Implementation Theorem

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

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

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

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

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

2. We can find an expression $t$ (called the variant function) whose value is reduced before each occurrence of $S'$ in $S[S'/X]$. 
Recursive Implementation Theorem

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

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

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

If both these conditions are satisfied, then:

$$\Delta \vdash S' \leq (\mu X. S)$$
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:
   \[ \text{SPEC} \approx \ldots\text{SPEC}\ldots\text{SPEC}\ldots\text{SPEC}\ldots \]
Recursive Implementation

1. Start with a specification: SPEC

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

   \[ \text{SPEC } \approx \ldots \text{SPEC} \ldots \text{SPEC} \ldots \text{SPEC} \ldots \]

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

   \[ \text{SPEC } \approx \ldots \{ t < t_0 \} ; \text{SPEC} \ldots \{ t < t_0 \} ; \text{SPEC} \ldots \{ t < t_0 \} ; \text{SPEC} \ldots \]
Recursive Implementation

1. Start with a specification: SPEC

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

   SPEC ≈ ...SPEC...SPEC...SPEC...

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

   SPEC ≈ ...{t < t₀}; SPEC...{t < t₀}; SPEC...{t < t₀}; SPEC...

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

   SPEC ≈ (μX...{t < t₀}; X...{t < t₀}; X...{t < t₀}; X...)

Recursive Implementation

1. Start with a specification: SPEC

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

   
   \[
   \text{SPEC} \approx \ldots \text{SPEC} \ldots \text{SPEC} \ldots \text{SPEC} \ldots
   \]

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

   
   \[
   \text{SPEC} \approx \ldots \{t < t_0\}; \text{SPEC} \ldots \{t < t_0\}; \text{SPEC} \ldots \{t < t_0\}; \text{SPEC} \ldots
   \]

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

   
   \[
   \text{SPEC} \approx (\mu X. \ldots \{t < t_0\}; X \ldots \{t < t_0\}; X \ldots \{t < t_0\}; X \ldots)
   \]

5. If necessary, apply Recursion Removal to get an iterative procedure.
Suppose we want to develop a factorial program.
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.
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:

\[ \text{if } n = 0 \text{ then } y := n! \text{ else } y := n! \text{ fi} \]
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:

\[
\text{if } n = 0 \text{ then } y := n! \text{ else } y := n! \text{ fi}
\]

When \( n = 0 \), we know that \( n! = 1 \), so:

\[
\text{if } n = 0 \text{ then } y := 1 \text{ else } y := n! \text{ fi}
\]
Refinement Example

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

\[
\begin{align*}
y &:= n! \quad \approx \quad y := n.(n - 1)! \\
&\quad \approx \quad y := (n - 1)!; \; y := n.y \\
&\quad \approx \quad n := n - 1; \; y := n!; \; n := n + 1; \; y := n.y
\end{align*}
\]
Refinement Example

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

\[
\begin{align*}
  y &:= n! &\approx& & y &= n.(n - 1)! \\
  &\approx& & y &= (n - 1)!; & y &= n.y \\
  &\approx& & n &= n - 1; & y &= n!; & n &= n + 1; & y &= n.y
\end{align*}
\]

The specification has been transformed as follows:

\[
\text{SPEC } \approx \begin{array}{l}
\text{if } n = 0 \\
\text{then } y := 1 \\
\text{else } n := n - 1; \text{ SPEC; } n := n + 1; & y := n.y \end{array}
\]

Note that \( n \) is reduced before the copy of SPEC on the right.
Refinement Example

Apply the Recursive Implementation Theorem:

\[
\text{SPEC} \approx \text{proc } F() \equiv \text{ if } n = 0 \\
\quad \text{then } y := 1 \\
\quad \text{else } n := n - 1; \\
\quad F(); \\
\quad n := n + 1; \\
\quad y := n.y \text{ fi end}
\]

This is an executable implementation of SPEC.
Refinement Example

Apply Recursion Removal:

\[
\text{SPEC} \approx \text{\texttt{var}} \langle i := 0 \rangle : \\
\quad \text{\texttt{while}} \ n \neq 0 \ \text{do} \\
\quad \quad i := i + 1; \ n := n - 1 \ \text{od}; \\
\quad y := 1; \\
\quad \text{\texttt{while}} \ i > 0 \ \text{do} \\
\quad \quad i := i - 1; \ n; = n + 1; \ y := n.y \ \text{od} \ \text{end}
\]

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

Simplify:

\[ \text{SPEC} \approx \text{var } \langle i := n \rangle : \]
\[ n := 0; \ y := 1; \]
\[ \text{while } i > 0 \text{ do} \]
\[ i := i - 1; \ n; = n + 1; \ y := n.y \quad \text{od end} \]
Refinement Example

Simplify:

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

Let \( j = n - i + 1 \) and simplify:

\[ \text{SPEC} \approx y := 1; \]
\[ \text{for } j := 1 \text{ to } n \text{ step } 1 \]
\[ y := j.y \text{ od end} \]
Refinement Example

Simplify:
SPEC ≈ \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} \textbf{ end}

Let \( j = n - i + 1 \) and simplify:
SPEC ≈ y := 1;
    \textbf{for} j := 1 \textbf{ to} n \textbf{ step} 1
    y := j.y \textbf{ od} \textbf{ end}

A long-winded process for such a simple specification.
Refinement Example

Simplify:

`SPEC ≈ \textbf{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 ≈ 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 \textit{any} recursive procedure!
Specification of a sorting program $\text{SORT}(a, b)$ is:

$$A[a..b] := A'[a..b].(\text{sorted}(A'[a..b]) \land \text{permutation_of}(A'[a..b], A[a..b]))$$

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

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

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

Define the specification partition as:

$$\langle A[a..b], p \rangle := \langle A'[a..b], p' \rangle.(a \leq p \leq b$$

$$\land A'[a..p - 1] \leq A'[p] \leq A'[p + 1..b]$$

$$\land \text{permutation_of}(A'[a..b], A[a..b]))$$
Now \( \text{SORT}(a, b) \approx \)

\[
\text{var } \langle p := 0 \rangle : \\
\text{if } b > a \text{ then partition;} \\
\text{SORT}(a, p - 1); \\
\text{SORT}(p + 1, b) \text{ fi}
\]

Apply Recursion Introduction to get the \textit{quicksort} algorithm:

\[
\text{proc qsort}(a, b) \equiv \\
\text{var } \langle p := 0 \rangle : \\
\text{if } b > a \text{ then partition;} \\
\text{qsort}(a, p - 1); \\
\text{qsort}(p + 1, b) \text{ fi}
\]
Loop Unrolling

while B do
  if \( B_1 \) then \( S_1 \)
  elsif \( . . . \)
  elsif \( B_i \) then \( S_i \)
  \( . . . \)
  else \( S_n \) fi od

Unroll one step of the loop:

while B do
  if \( B_1 \) then \( S_1 \)
  elsif \( . . . \)
  elsif \( B_i \) then \( S_i \); if \( B \land Q \) then if \( B_1 \) then \( . . . \) fi fi
  \( . . . \)
  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 $\land$ 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 $Q = B_i$, and assume that the $B_i$ are disjoint:

```plaintext
while B do
  if $B_1$ then $S_1$
  elsif ...$
  elsif $B_i$ then $S_i$
  ...$
  else $S_n$ fi
fi od
```

becomes:

```plaintext
while B do
  if $B_1$ then $S_1$
  elsif ...$
  elsif $B_i$ then while $B \land B_i$ do $S_i$ od
  ...$
  else $S_n$ fi
fi od
```
Algorithm Derivation

Suppose we want to develop an integer exponentiation algorithm.
Suppose we want to develop an integer exponentiation algorithm. The specification is very simple:

\[ \text{EXP}(x, n) \,=_{\text{DF}}\, y := x^n \]

where \( n \) is a non-negative integer.
Algorithm Derivation

Suppose we want to develop an integer exponentiation algorithm. The specification is very simple:

\[
\text{EXP}(x, n) \overset{\text{DF}}{=} y := x^n
\]

where \( n \) is a non-negative integer.

Our derivation uses the following facts about exponentiation:
Suppose we want to develop an integer exponentiation algorithm. The specification is very simple:

\[
\text{EXP}(x, n) =_{\text{DF}} y := x^n
\]

where \( n \) is a non-negative integer.

Our derivation uses the following facts about exponentiation:

1. \( x^0 = 1 \) for all \( x \);
Algorithm Derivation

Suppose we want to develop an integer exponentiation algorithm. The specification is very simple:

\[
\text{EXP}(x, n) =_{\text{DF}} y := x^n
\]

where \( n \) is a non-negative integer.

Our derivation uses the following facts about exponentiation:

1. \( x^0 = 1 \) for all \( x \);
2. \( x^{2n} = (x \times x)^n \) and;
Suppose we want to develop an integer exponentiation algorithm. The specification is very simple:

\[
\text{EXP}(x, n) =_{\text{DF}} y := x^n
\]

where \( n \) is a non-negative integer.

Our derivation uses the following facts about exponentiation:

1. \( x^0 = 1 \) for all \( x \);
2. \( x^{2n} = (x \times x)^n \) and;
3. \( x^{n+1} = x \times x^n \)
Algorithm Derivation

Apply Splitting A Tautology and Insert Assertions:

\[
\text{EXP}(x, n) \approx \begin{cases} 
\text{if } n = 0 & \{ n = 0 \}; \ \text{EXP}(x, n) \\
\text{elsif } \text{even?}(x) & \{ n > 0 \land \text{even?}(n) \}; \ \text{EXP}(x, n) \\
\text{else} & \{ n > 0 \land \text{odd?}(n) \}; \ \text{EXP}(x, n) 
\end{cases}
\]
Algorithm Derivation

Apply Splitting_A_Tautology and InsertAssertions:

\[
\text{EXP}(x, n) \approx \begin{cases} 
\text{if } n = 0 & \text{then } \{n = 0\}; \text{ EXP}(x, n) \\
\text{elsif even}\? (x) & \text{then } \{n > 0 \land \text{even}\?(n)\}; \text{ EXP}(x, n) \\
\text{else} & \{n > 0 \land \text{odd}\?(n)\}; \text{ EXP}(x, n) 
\end{cases}
\]

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

\[
\text{if } n = 0 \text{ then } y := 1 \\
\text{elsif even}\?(n) \text{ then } \{n > 0 \land \text{even}\?(n)\}; \\
\quad \text{ EXP}(x \cdot x, n/2) \\
\text{else } \{n > 0 \land \text{odd}\?(n)\}; \\
\quad \text{ EXP}(x, n - 1); y := x \cdot y \text{ fi}
\]
Algorithm Derivation

Apply Splitting_A_Tautology and InsertAssertions:

\[ \text{EXP}(x, n) \approx \begin{cases} 
\text{if } n = 0 \text{ then } \{n = 0\}; \text{ EXP}(x, n) \\
\text{elsif } \text{even?}(x) \text{ then } \{n > 0 \land \text{even?}(n)\}; \text{ EXP}(x, n) \\
\text{else } \{n > 0 \land \text{odd?}(n)\}; \text{ EXP}(x, n) 
\end{cases} \]

Use the assertions to refine each copy of \text{EXP}(x, n):
\begin{align*}
\text{if } n = 0 & \text{ then } y := 1 \\
\text{elsif } \text{even?}(n) & \text{ then } \{n > 0 \land \text{even?}(n)\}; \\
& \quad \text{ EXP}(x \ast x, n/2) \\
\text{else } & \{n > 0 \land \text{odd?}(n)\}; \\
& \quad \text{ EXP}(x, n - 1); y := x \ast y \text{ fi}
\end{align*}

This is the elaborated specification
Apply the Recursive Implementation Theorem:

\[
\text{proc } \exp(x, n) \equiv \\
\quad \text{if } n = 0 \text{ then } y := 1 \\
\quad \text{elsif even?(n) then } \exp(x \times x, n/2) \\
\quad \quad \text{else } \exp(x, n - 1); y := x \times y \text{ fi.}
\]

This is now an executable, recursive implementation of the specification \(\text{EXP}(x, n)\)
Replace parameter $n$ by a global variable:

```
proc exp(x, n) ≡ exp1(x).

proc exp1(x) ≡
    if $n = 0$ then $y := 1$
    elsif even?(n) then $n := n/2; \ exp1(x \ast x)$
    else $n := n - 1; \ exp1(x); \ y := x \ast y$ fi.
```

Apply Recursion Removal to exp1:

```
proc exp1(x) ≡
    var $\langle L := \langle \rangle \rangle$
    actions $A$
        $A \equiv$ if $n = 0$ then $y := 1; \ call \ \hat{F}$
        elsif even?(n) then $n := n/2; \ x := x \ast x; \ call \ A$
        else $n := n - 1; \ L \\leftarrow \leftarrow x; \ call \ A$ fi.
    $\hat{F} \equiv$ if $L = \langle \rangle$ then call $Z$
        else $x \\leftarrow \leftarrow L; \ y := x \ast y; \ call \ \hat{F}$ fi. endactions end.
```
Algorithm Derivation

Restructure the regular action system:

\[
\text{proc } \exp(x, n) \equiv \\
\text{var } \langle L := \langle \rangle \rangle : \\
\quad \text{while } n \neq 0 \text{ do} \\
\quad \quad \text{if } \text{even?}(n) \text{ then } x := x \times x; \ n := n/2 \\
\quad \quad \text{else } n := n - 1; \ L \leftarrow x \ 	ext{fi od;} \\
\quad y := 1; \\
\quad \text{while } L \neq \langle \rangle \text{ do } x \leftarrow L; \ y := x \times y \ 	ext{od.}
\]

Apply Entire Loop Unrolling after the assignment \( n := n/2 \) with the condition \( n \neq 0 \land \text{even?}(n) \):
Algorithm Derivation

\begin{align*}
\text{proc } \text{exp}(x, n) \equiv \\
\text{var } \langle L := \langle \rangle \rangle : \\
\quad \text{while } n \neq 0 \text{ do} \\
\quad \quad \text{if even?}(n) \text{ then } x := x \times x; \ n := n/2; \\
\quad \quad \quad \text{while } n \neq 0 \land \text{even?}(n) \text{ do} \\
\quad \quad \quad \quad \text{if even?}(n) \text{ then } x := x \times x; \ n := n/2 \\
\quad \quad \quad \quad \quad \text{else } n := n - 1; \ L \leftarrow x \ 	ext{fi od;} \\
\quad \quad \quad \text{else } n := n - 1; \ L \leftarrow x \ 	ext{fi od;} \\
\quad \quad y := 1; \\
\quad \text{while } L \neq \langle \rangle \text{ do } x \leftarrow L; \ y := x \times y \ 	ext{od.}
\end{align*}
Algorithm Derivation

Simplify:

\[
\text{proc } \text{exp}(x, n) \equiv \\
\text{var } \langle L := \langle \rangle \rangle : \\
\quad \text{while } n \neq 0 \text{ do} \\
\quad \quad \text{if even?}(n) \text{ then while even?}(n) \text{ do } x := x \times x; \ n := n/2 \text{ od} \\
\quad \quad \text{else } n := n - 1; \ L \xleftarrow{\text{push}} x \text{ fi od; } \\
\quad y := 1; \\
\quad \text{while } L \neq \langle \rangle \text{ do } x \xleftarrow{\text{pop}} L; \ y := x \times y \text{ od.}
\]

Unroll a step after the inner \textbf{while} loop:
Algorithm Derivation

proc exp\(x, n\) ≡

var \(\langle L := \langle \rangle \rangle:\)

while \(n \neq 0\) do
  if even?(\(n\)) then while even?(\(n\)) do \(x := x \times x; \ n := n/2\) od;
    \(L \leftarrow x; \ n := n - 1\)
  else \(L \leftarrow x; \ n := n - 1\) fi od;

\(y := 1;\)

while \(L \neq \langle \rangle\) do \(x \leftarrow L; \ y := x \times y\) od.

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

\[
\text{proc } \exp(x, n) \equiv \\
\text{var } \langle L := \langle \rangle \rangle : \\
\quad \text{while } n \neq 0 \text{ do} \\
\quad \quad \text{while } \text{even?}(n) \text{ do } x := x \times x; \ n := n/2 \text{ od;} \\
\quad \quad n := n - 1; \ L \overset{\text{push}}{\leftarrow} x \text{ od;} \\
\quad y := 1; \\
\quad \text{while } L \neq \langle \rangle \text{ do } x \overset{\text{pop}}{\leftarrow} L; \ y := x \times y \text{ od.}
\]

If we move the assignment \( y := 1 \) to the front, then we can merge the bodies of the two \textbf{while} loops.

Note: The order of execution of the statements in the second \textbf{while} loop is reversed.
Algorithm Derivation

\textbf{proc} \textit{exp}(x, n) \equiv
\begin{align*}
\textbf{var} & \langle L := \langle \rangle \rangle : \\
y & := 1; \\
\textbf{while} \ n \neq 0 \ \textbf{do} \\
& \quad \textbf{while} \ \text{even?}(n) \ \textbf{do} \ x := x \times x; \ n := n/2 \ \textbf{od}; \\
& \quad n := n - 1; \ L \leftarrow \text{push} \ x; \\
& \quad x \leftarrow \text{pop} \ L; \ y := x \times y \ \textbf{od}. \\
\end{align*}

Local variable \( L \) is now redundant, since \( L \leftarrow \text{push} \ x; \ L \leftarrow \text{pop} \ x \approx \text{skip}: \)

\textbf{proc} \textit{exp}(x, n) \equiv
\begin{align*}
y & := 1; \\
\textbf{while} \ n \neq 0 \ \textbf{do} \\
& \quad \textbf{while} \ \text{even?}(n) \ \textbf{do} \ x := x \times x; \ n := n/2 \ \textbf{od}; \\
& \quad n := n - 1; \ y := x \times y \ \textbf{od}. \\
\end{align*}
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, FlagRemoval, 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_Foward, 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
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