Transformational Programming

Martin Ward

Reader in Software Engineering

martin@gkc.org.uk

Software Technology Research Lab
De Montfort University

Waterfall Model

1. **Requirements Elicitation:** Analyse the problem domain and determine what the program is required to do

Waterfall Model

1. **Requirements Elicitation:** Analyse the problem domain and determine what the program is required to do
2. **Design:** Develop the overall structure of the program

Waterfall Model

1. **Requirements Elicitation:** Analyse the problem domain and determine what the program is required to do
2. **Design:** Develop the overall structure of the program
3. **Implementation:** Write source code to implement the design in a particular programming language

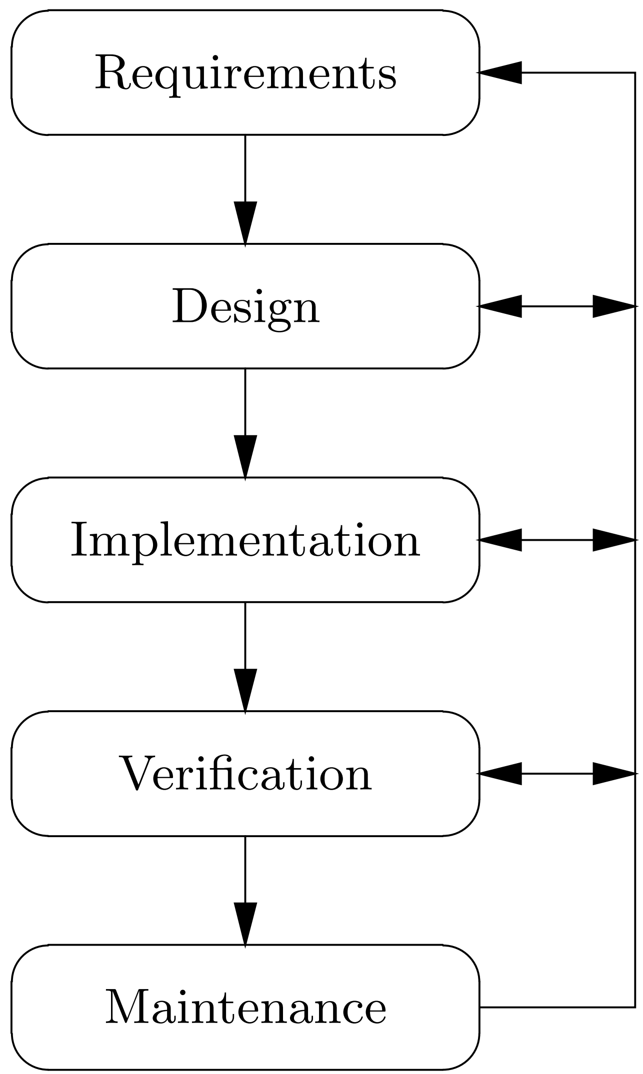
Waterfall Model

1. **Requirements Elicitation:** Analyse the problem domain and determine what the program is required to do
2. **Design:** Develop the overall structure of the program
3. **Implementation:** Write source code to implement the design in a particular programming language
4. **Verification:** Run tests and debugging

Waterfall Model

1. **Requirements Elicitation:** Analyse the problem domain and determine what the program is required to do
2. **Design:** Develop the overall structure of the program
3. **Implementation:** Write source code to implement the design in a particular programming language
4. **Verification:** Run tests and debugging
5. **Maintenance:** Any modifications required after delivery to correct faults, improve performance, or adapt the product to a modified environment

Waterfall Model



Proving Correctness

Program testing can be used to show the presence of bugs, but never to show their absence — Dijkstra 1970.

Proving Correctness

Program testing can be used to show the presence of bugs, but never to show their absence — Dijkstra 1970.

To prove that a program is correct we need two things:

Proving Correctness

Program testing can be used to show the presence of bugs, but never to show their absence — Dijkstra 1970.

To prove that a program is correct we need two things:

1. A precise mathematical specification which defines what the program is supposed to do, and

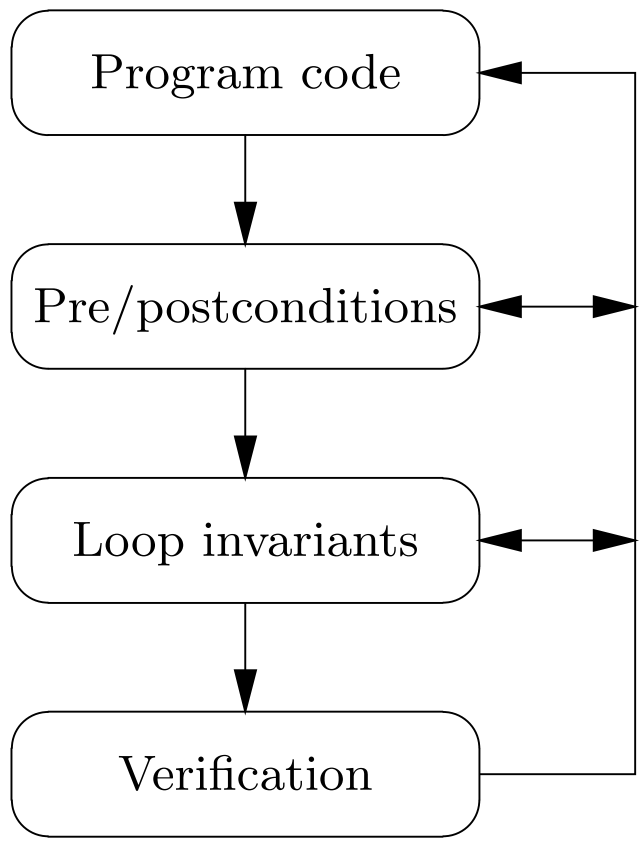
Proving Correctness

Program testing can be used to show the presence of bugs, but never to show their absence — Dijkstra 1970.

To prove that a program is correct we need two things:

1. A precise mathematical specification which defines what the program is supposed to do, and
2. A mathematical proof that the program satisfies the specification

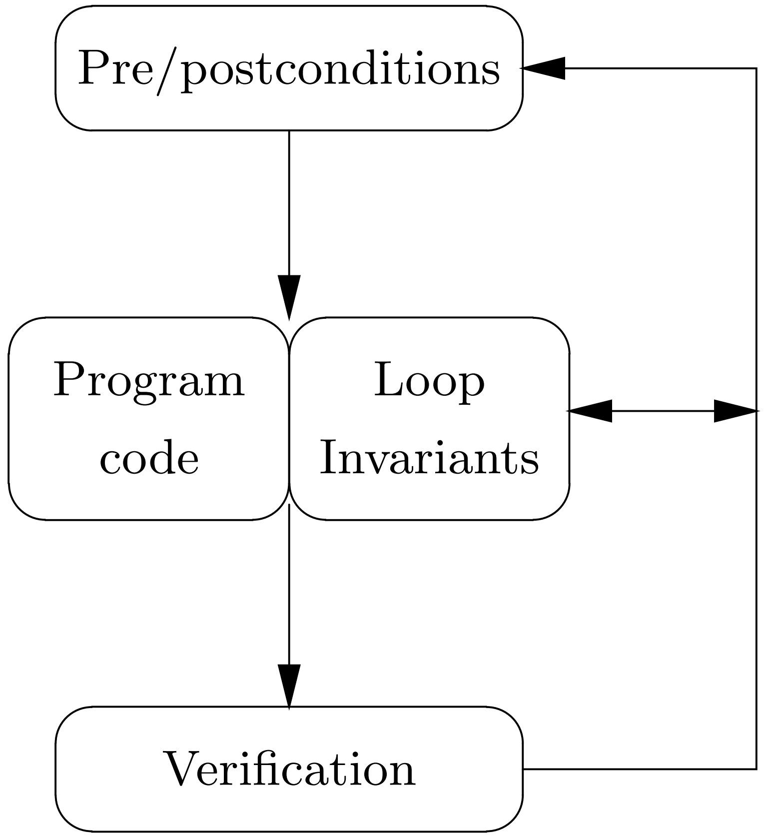
Program Verification



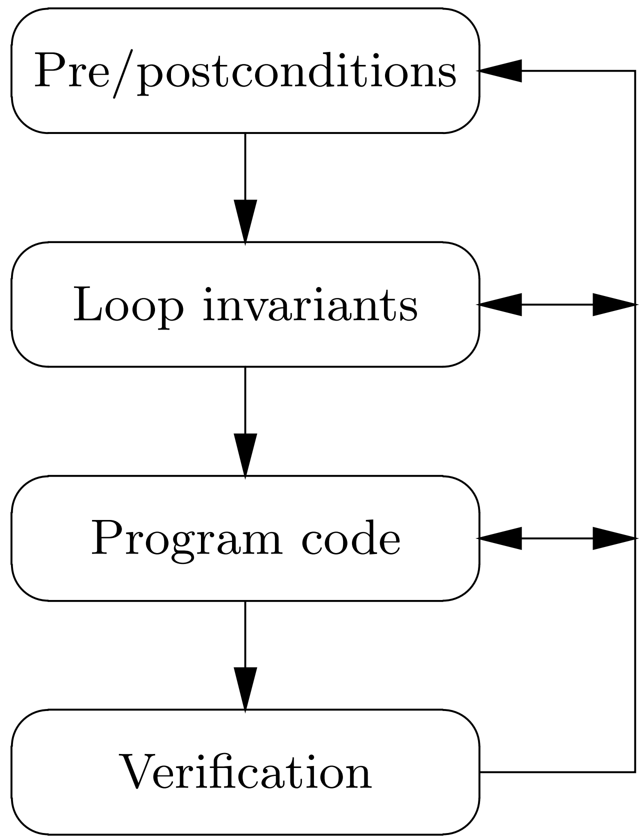
Verification of a Loop

1. Determine the loop termination condition;
2. Determine the loop body;
3. Determine a suitable loop invariant;
4. Prove that the loop invariant is preserved by the loop body;
5. Determine a variant function for the loop;
6. Prove that the variant function is reduced by the loop body (thereby proving termination of the loop);
7. Prove that the combination of the invariant plus the termination condition satisfies the specification for the loop.

Dijkstra's Approach



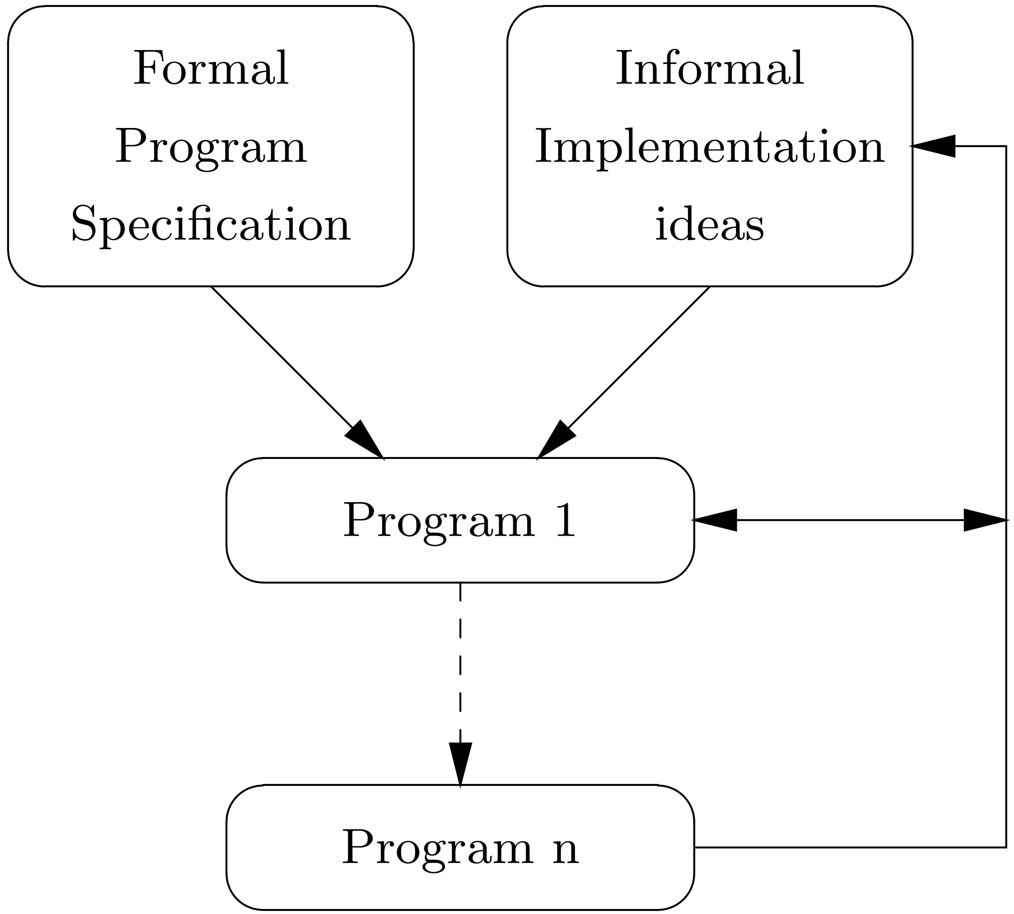
Invariant Based Programming



Common Factors

- All these development methods require the invention of *loop invariants*.
- In all these methods, the final step is *Verification*
- The program under development is not guaranteed to be correct until verification is complete
- Introducing a loop requires developing a loop invariant and variant expression

Transformational Programming



Transformational Programming Stages

1. **Formal Specification:** A WSL specification statement.

Transformational Programming Stages

1. **Formal Specification:** A WSL specification statement.
2. **Elaboration:** Elaborate the specification statement by taking out simple cases

Transformational Programming Stages

1. **Formal Specification:** A WSL specification statement.
2. **Elaboration:** Elaborate the specification statement by taking out simple cases
3. **Divide and Conquer:** Tackle the general case via some form of "divide and conquer" strategy Use the informal ideas to guide the selection of transformations

Transformational Programming Stages

1. **Formal Specification:** A WSL specification statement.
2. **Elaboration:** Elaborate the specification statement by taking out simple cases
3. **Divide and Conquer:** Tackle the general case via some form of "divide and conquer" strategy Use the informal ideas to guide the selection of transformations
4. **Recursion Introduction:** Apply the Recursive Implementation Theorem to produce a recursive program with no remaining copies of the specification

Transformational Programming Stages

1. **Formal Specification:** A WSL specification statement.
2. **Elaboration:** Elaborate the specification statement by taking out simple cases
3. **Divide and Conquer:** Tackle the general case via some form of "divide and conquer" strategy Use the informal ideas to guide the selection of transformations
4. **Recursion Introduction:** Apply the Recursive Implementation Theorem to produce a recursive program with no remaining copies of the specification
5. **Recursion Removal:** Apply the Generic Recursion Removal Theorem

Transformational Programming Stages

1. **Formal Specification:** A WSL specification statement.
2. **Elaboration:** Elaborate the specification statement by taking out simple cases
3. **Divide and Conquer:** Tackle the general case via some form of "divide and conquer" strategy Use the informal ideas to guide the selection of transformations
4. **Recursion Introduction:** Apply the Recursive Implementation Theorem to produce a recursive program with no remaining copies of the specification
5. **Recursion Removal:** Apply the Generic Recursion Removal Theorem
6. **Optimisation:** As required.

Formal Specification

A formal specification defines precisely what the program is required to accomplish, without necessarily giving any indication as to how the task is to be accomplished.

A formal specification for a factorial program could be written as:

$$r := n!$$

A formal specification for the Quicksort algorithm for sorting the array $A[a\,..\,b]$ is the statement SORT(a, b):

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

Formal Specification

The *form* of the specification should mirror the real-world nature of the requirements. Construct suitable abstractions such that local changes to the requirements involve local changes to the specification.

The *notation* used for the specification should permit unambiguous expression of requirements and support rigorous analysis to uncover contradictions and omissions.

Elaboration

The *Elaboration* stage is the process of applying transformations to take out the simple cases.

Typically, this uses Splitting a Tautology to duplicate the specification, then insert assertions, then use the assertions to refine the appropriate copy of the specification to the trivial implementation.

For the factorial program, the simplest case is $0! = 1$, so we split on the test $n = 0$:

if $n = 0$ **then** $r := n!$ **else** $r := n!$ **fi**

and simplify the case where $n = 0$:

if $n = 0$ **then** $r := 1$ **else** $r := n!$ **fi**

Elaboration

For the sort function, the simplest case is when $a \geqslant b$. In this case, the array has zero or one elements and is therefore already sorted.

Elaboration

For the sort function, the simplest case is when $a \geqslant b$. In this case, the array has zero or one elements and is therefore already sorted.

SORT(a, b) is transformed to:

if $a \geqslant b$ **then** SORT(a, b)
else SORT(a, b) **fi**

Elaboration

For the sort function, the simplest case is when $a \geqslant b$. In this case, the array has zero or one elements and is therefore already sorted.

SORT(a, b) is transformed to:

if $a \geqslant b$ **then** SORT(a, b)
else SORT(a, b) **fi**

Add assertions:

if $a \geqslant b$ **then** $\{a \geqslant b\}$; SORT(a, b)
else $\{a < b\}$; SORT(a, b) **fi**

Elaboration

For the sort function, the simplest case is when $a \geqslant b$. In this case, the array has zero or one elements and is therefore already sorted.

SORT(a, b) is transformed to:

if $a \geqslant b$ **then** SORT(a, b)
else SORT(a, b) **fi**

Add assertions:

if $a \geqslant b$ **then** $\{a \geqslant b\}$; SORT(a, b)
else $\{a < b\}$; SORT(a, b) **fi**

Use the assertions:

if $a \geqslant b$ **then** $\{a \geqslant b\}$; **skip**
else $\{a < b\}$; SORT(a, b) **fi**

Divide and Conquer

Use the informal implementation ideas to direct the selection of transformations.

Divide and Conquer

Use the informal implementation ideas to direct the selection of transformations.

For the factorial program the "idea" is to use the definition of the factorial function when $n > 0$:

$$n! = n.(n-1)!$$

Divide and Conquer

Use the informal implementation ideas to direct the selection of transformations.

For the factorial program the "idea" is to use the definition of the factorial function when $n > 0$:

$$n! = n.(n-1)!$$

Applying this idea to the elaborated specification we get:

if $n = 0$ **then** $r := 1$
else $r := (n-1)!;\ r := n.r$ **fi**

Divide and Conquer

Use the informal implementation ideas to direct the selection of transformations.

For the factorial program the "idea" is to use the definition of the factorial function when $n > 0$:

$$n! = n.(n-1)!$$

Applying this idea to the elaborated specification we get:

if $n = 0$ **then** $r := 1$
else $r := (n-1)!;\ r := n.r$ **fi**

The statement $r := (n-1)!$ is expanded into three statements:
$n := n - 1;\ r := n!;\ n := n + 1$

Note that this contains a copy of the original specification.

Divide and Conquer

For Quicksort the implementation idea has two stages:

Divide and Conquer

For Quicksort the implementation idea has two stages:

1. Partition the array around a pivot element: so that all the elements less than the pivot go on one side and the larger elements go on the other side

Divide and Conquer

For Quicksort the implementation idea has two stages:

1. Partition the array around a pivot element: so that all the elements less than the pivot go on one side and the larger elements go on the other side

2. Sort the two sub-arrays sorted using copies of the original specification statement.

Divide and Conquer

For Quicksort the implementation idea has two stages:

1. Partition the array around a pivot element: so that all the elements less than the pivot go on one side and the larger elements go on the other side

2. Sort the two sub-arrays sorted using copies of the original specification statement.

This leads to the program:

if $a < b$ **then** Partition(a, b, m);
SORT$(a, m - 1)$;
SORT$(m + 1, b)$ **fi**

In this case, the elaborated specification contains two copies of the original specification

Recursion Introduction

Apply the Recursive Implementation Theorem to produce a recursive procedure.

The Recursive Implementation Theorem can be applied when:

1. The elaborated specification is a refinement of the original specification; and
2. There exists a variant function which is reduced before each copy of the original specification

If both these conditions are satisfied, then the elaborated specification can be transformed into a recursive procedure, with each copy of the original specification replaced by a recursive call.

Recursion Introduction

For the factorial program, the elaborated specification is:

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

else $n := n - 1;\ r := n!;\ n := n + 1;\ r := n.r$ **fi**

This is equivalent to, and contains one copy of, the original specification $r := n!$

Also, the value of n is a non-negative integer and is reduced before the copy of the specification.

If an elaborated specification is equivalent to the original specification and there is an expression whose value is reduced before each copy of the specification, then it can be refined to a recursive procedure with the internal copies of the specification replaced by recursive calls.

Recursion Introduction

For the factorial program we get this recursive procedure:

proc fact $\equiv$

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

else $n := n - 1$; fact; $n := n + 1$; $r := n.r$ **fi**

The value of the variant function must be "smaller" in terms of a well-founded partial order on some set of values. Typically this will be a non-negative integer, but other possibilities include a subset order and a lexical order on a list of integers.

Recursion Introduction

More formally:

If $\preccurlyeq$ is a well-founded partial order on some set Γ and **t** is a term giving values in Γ and t_0 is a variable which does not occur in **S** or **S**′ then if

$$\{\mathbf{P} \wedge \mathbf{t} \preccurlyeq t_0\};\ \mathbf{S}\ \leq\ \mathbf{S}'[\{\mathbf{P} \wedge \mathbf{t} \prec t_0\};\ \mathbf{S}/X])$$

then $\{\mathbf{P}\};\ \mathbf{S}\ \leq\ \textbf{proc}\ X\ \equiv\ \mathbf{S}'\ \textbf{end}$

Here, **S** is the original specification which is elaborated to $\mathbf{S}'[\mathbf{S}/X]$.

P is any required precondition: if no precondition is needed, then let **P** be **true**.

The variant function is **t**. If the value of **t** is initially no larger than t_0, the before each copy of the specification we know that **t** is strictly less than t_0.

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 :

    $A_1 \;\equiv$ $\mathbf{S}_1$**.**

    $\ldots A_i \;\equiv$ $\mathbf{S}_i$**.**

    $\ldots B_j \;\equiv \mathbf{S}_{j0};\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));$

            $\ldots;\ F(g_{jn_j}(x));\ \mathbf{S}_{jn_j}$**.**

    $\ldots$ **endactions.**

where $\mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j}$ preserve the value of x and no **S** contains a call to F and the statements $\mathbf{S}_{j0}$, $\mathbf{S}_{j1}$, . . . ,$\mathbf{S}_{jn_j-1}$ contain no action calls.

Note: Any action system can be converted into this form using the destructuring and restructuring transformations.

Recursion Removal

Stack L records "postponed" operations

A postponed call $F(e)$ is recorded by pushing $\langle 0, e \rangle$ onto L

A postponed execution of $\mathbf{S}_{jk}$ is recorded by pushing the value $\langle\langle j, k\rangle, x\rangle$ onto L.

When then procedure body would normally terminate (via **call** Z) we call a new action $\hat{F}$ which pops the top item off L and carries out the postponed operation.

If we call $\hat{F}$ with the stack empty then all postponed operations have been completed and the procedure terminates by calling Z.

Recursion Removal

proc $F'(x) \;\equiv$

var $\langle L := \langle\rangle, m := 0\rangle :$

actions $A_1 :$

$A_1 \;\equiv$ S_1[**call** $\hat{F}$/**call** Z]**.**

$\ldots A_i \;\equiv$ S_i[**call** $\hat{F}$/**call** Z]**.**

$\ldots B_j \;\equiv$ S_{j0};

$L := \langle\langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle,$

$\ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L;$

$x := g_{j1};$

call A_j**.**

$\ldots \hat{F} \;\equiv$ **if** $L = \langle\rangle$

then call Z

else $\langle m, x\rangle \xleftarrow{\text{pop}} L;$

if $m = 0 \rightarrow$ **call** A_1

$\Box \ldots \Box\; m = \langle j, k\rangle \rightarrow$ S_{jk}[**call** $\hat{F}$/**call** Z]; **call** $\hat{F}$

$\ldots$ **fi fi. endactions end.**

Linear Recursion Removal

Consider the special case of a parameterless, linear recursion:

proc $F \;\equiv$

actions A_1 :

$A_1 \;\equiv$ $\mathbf{S}_1$**.**

$\ldots A_i \;\equiv$ $\mathbf{S}_i$**.** $\ldots$

$B_1 \;\equiv$ $\mathbf{S}_0;\; F;\;$ $\mathbf{S}_{11}$**. endactions.**

After applying the general recursion removal theorem, the *only* value pushed into the stack is $\langle\langle 1, 1\rangle\rangle$. So the stack can be replaced by an integer which records how many values are on the stack,

Linear Recursion Removal

The iterative program is:

proc $F' \equiv$

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

    actions A_1 :

    $A_1 \equiv$ $\mathbf{S}_1$[**call** $\hat{F}$/**call** Z]**.**

    $\ldots A_i \equiv$ $\mathbf{S}_i$[**call** $\hat{F}$/**call** Z]**.** $\ldots$

    $B_1 \equiv$ $\mathbf{S}_{j0}$; $L := L + 1$; **call** A_1**.**

    $\hat{F} \equiv$ **if** $L = 0$

        then call Z

        else $L := L - 1$;

          $\mathbf{S}_{11}$[**call** $\hat{F}$/**call** Z]; **call** $\hat{F}$ **fi. endactions end.**

Linear Recursion Removal

For example:

proc $F \;\equiv$

  if B then S$_1$ **else S**$_2$; F; **S**$_3$ **fi.**

is equivalent to the iterative program:

proc $F' \;\equiv$

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

    actions A_1 :

    $A_1 \;\equiv$ **if B then S**$_1$; **call** $\hat{F}$ **else call** B_1 **fi.**

    $B_1 \;\equiv$ **S**$_2$; $L := L + 1$; **call** A_1**.**

    $\hat{F} \;\equiv$ **if** $L = 0$

        then call Z

        else $L := L - 1$;

          S$_3$; **call** $\hat{F}$ **fi. endactions end.**

Linear Recursion Removal

Remove the recursion in $\hat{F}$, unfold into A_1, unfold B_1 into A_1 and remove the recursion to give:

proc $F' \equiv$

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

    while $\neg$**B** **do** **S**$_2$; $L := L + 1$ **od**;

    S$_1$;

    while $L \neq 0$ **do** $L := L - 1$; **S**$_3$ **od.**

This restructuring is carried out automatically by FermaT's Collapse_Action_System transformation.

Recursion Removal Example

For the factorial program we derived this recursive procedure:

proc fact $\equiv$

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

else $n := n - 1;$ fact; $n := n + 1;$ $r := n.r$ **fi**

This transforms to the equivalent iterative procedure:

proc F' $\equiv$

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

while $n \neq 0$ **do** $n := n - 1;$ $L := L + 1$ **od**;

$r := 1;$

while $L \neq 0$ **do** $L := L - 1;$ $n := n + 1;$ $r := n.r$ **od.**

The first loop just copies n to L and sets n to zero.

The second loop iterates n from 1 to L (which was the initial value of n).

Recursion Removal Example

proc $F' \equiv$

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

    $n := 0;$

    $r := 1;$

    while $L \neq 0$ **do** $L := L - 1;\ n := n + 1;\ r := n.r$ **od.**

If we add a new variable n_0 to record the initial value of n then L is not needed since the test $L \neq 0$ can be replaced by the equivalent test $n \neq n_0$:

proc $F' \equiv$

  var $\langle n_0 := n \rangle :$

    $n := 0;$

    $r := 1;$

    while $n \neq n_0$ **do** $n := n + 1;\ r := n.r$ **od.**

Recursion Removal Example

The result can be written as a **for** loop:

proc $F' \ \equiv$

$r := 1;$

for $i := 1$ **to** n **do** $r := i.r$ **od.**

Selection Sort

Define the predicate $\mathsf{Sorted}(A, i, j)$ to be true precisely when the array segment $A[i\,..\,j]$ is sorted:

$$\mathsf{Sorted}(A, i, j) \;=_{\text{DF}}\; \forall k.\, i \leqslant k < j \Rightarrow A[k] \leqslant A[k+1]$$

Define the predicate $\mathsf{Perm}(A, A')$ to mean that the elements in array A form a permutation of the elements in array A'.

The formal specification for a sorting program can now be written as follows:

$$\mathsf{SORT}(A, i, j) \;=_{\text{DF}}$$
$$A[i\,..\,j] := A'[i\,..\,j].(\mathsf{Sorted}(A', i, j) \,\wedge\, \mathsf{Perm}(A[i\,..\,j], A'[i\,..\,j]))$$

Selection Sort: Elaboration

If $i \geqslant j$ then the array has at most one element and is therefore already sorted. So in this case:

$$\mathsf{SORT}(A, i, j) \;\approx\; \textbf{skip}$$

So we can elaborate the specification to:

if $i < j$ **then** $\mathsf{SORT}(A, i, j)$ **fi**

Selection Sort: Informal Idea

The informal idea behind selection sorting is: "find the smallest element in the array, and move it to the front".

Inserting any permutation of $A[i\,..\,j]$ before a copy of SORT(A, i, j) has no effect, so SORT(A, i, j) is equivalent to:

if $i < j$

then var $\langle s := 0 \rangle$:

$s := s'.(i \leqslant s' \leqslant j \ \wedge\ \forall k.\, i \leqslant k \leqslant j \Rightarrow A[s'] \leqslant A[k])$;

$\langle A[i], A[s] \rangle := \langle A[s], A[i] \rangle$ **end**;

SORT(A, i, j) **fi**

With this addition to the program, we have the assertion: $\forall k.\, i \leqslant k \leqslant j \Rightarrow A[i] \leqslant A[k])$ just before the copy of SORT.

So SORT(A, i, j) can be refined as SORT$(A, i + 1, j)$

Selection Sort: Recursion Introduction

The expression $j - i$ is positive, and is reduced before the copy of SORT, so we can apply Recursion_Introduction to get this recursive program:

proc sort $\equiv$
    if $i < j$
        then var $\langle s := 0 \rangle$:
            $s := s'.(i \leqslant s' \leqslant j \ \wedge \ \forall k.\, i \leqslant k \leqslant j \Rightarrow A[s'] \leqslant A[k]);$
            $\langle A[i], A[s] \rangle := \langle A[s], A[i] \rangle$ **end**;
        $i := i + 1;$
        sort **fi.**

Selection Sort: Recursion Introduction

This is equivalent to the **while** loop:

proc sort $\equiv$

while $i < j$ **do**

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

$s := s'.(i \leqslant s' \leqslant j \ \wedge \ \forall k.\, i \leqslant k \leqslant j \Rightarrow A[s'] \leqslant A[k])$;

$\langle A[i], A[s] \rangle := \langle A[s], A[i] \rangle$ **end**;

$i := i + 1$ **od.**

Selection Sort: Refinement

To implement the inner specification statement, first take out a trivial case:

if $i = j$ then the specification can be implemented as the assignment $s := i$:

if $i = j$
  then $s := i$
  else $s := s'.(i \leqslant s' \leqslant j \ \wedge \ \forall k.\, i \leqslant k \leqslant j \Rightarrow A[s'] \leqslant A[k])$ **fi**

Our informal idea for implementing the specification is to first set s to the index of the smallest element in $A[i\,..\,j-1]$ and then compare $A[s]$ against $A[j]$.

Selection Sort: Refinement

This produces the elaborated specification:

if $i = j$
  then $s := i$
  else $j := j - 1;$
    $s := s'.(i \leqslant s' \leqslant j \ \wedge \ \forall k.\, i \leqslant k \leqslant j \Rightarrow A[s'] \leqslant A[k]);$
    $j := j + 1;$
    if $A[j] < A[s]$ **then** $s := j$ **fi fi**

The variable j is our variant function, so we can apply Recursion_Introduction on the copy of the specification:

proc search $\equiv$
  if $i = j$
    then $s := i$
    else $j := j - 1;$ search; $j := j + 1;$
      if $A[j] < A[s]$ **then** $s := j$ **fi fi.**

Selection Sort: Refinement

Apply Recursion_Removal to get:

proc search $\equiv$

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

    while $i \neq j$ **do**

      $j := j - 1;\ L := L + 1$ **od**;

    $s := i;$

    while $L \neq 0$ **do**

      $L := L - 1;\ j := j + 1;$

      if $A[j] < A[s]$ **then** $s := j$ **fi od end.**

Selection Sort: Refinement

As above, the variable L is incremented whenever j is decremented, and vice-versa. So if j_0 is the original value of j then $L = j_0 - j$. The first loop assigns $j := i$:

proc search $\equiv$

  var $\langle j_0 := j \rangle$:

    $j := i$;

    $s := i$;

    while $j < j_0$ **do**

      $j := j + 1$;

      if $A[j] < A[s]$ **then** $s := j$ **fi od end.**

Selection Sort: Refinement

Putting this into the sorting program, instead of the specification we get the completed program:

proc sort $\equiv$

  while $i < j$ **do**

    var $\langle s := i, j_0 := j \rangle$:

      $j := i$;

      while $j < j_0$ **do**

        $j := j + 1$;

        if $A[j] < A[s]$ **then** $s := j$ **fi** **od**;

      $\langle A[i], A[s] \rangle := \langle A[s], A[i] \rangle$ **end**;

    $i := i + 1$ **od.**

String Comparison

Given two character strings a and b, it required to determine whether they are equal "apart from blanks" (the space character being regarded as non-significant).

We represent the strings as arrays of characters, with the special symbol end denoting the end of the string.

String Comparison

Given two character strings a and b, it required to determine whether they are equal "apart from blanks" (the space character being regarded as non-significant).

We represent the strings as arrays of characters, with the special symbol end denoting the end of the string.

Define the function $\mathsf{strip}(s, i)$ to return the sequence of all non-space characters in s from the ith character to the end of the string:

$$\mathsf{strip}(s,i) \quad = \quad \begin{cases} \langle\rangle & \text{if } s[i] = \mathsf{end} \\ \mathsf{strip}(s, i+1) & \text{if } s[i] = \mathsf{space} \\ \langle s[i]\rangle \mathbin{+\!\!+} \mathsf{strip}(s, i+1) & \text{otherwise} \end{cases}$$

Formal Specification

With this definition of strip our formal specification is:

$$\text{COMP} =_{\text{DF}} \textbf{if}\ \mathsf{strip}(a,1) = \mathsf{strip}(b,1)\ \textbf{then}\ R := 1\ \textbf{else}\ R := 0\ \textbf{fi}$$

Informal Ideas

Our informal idea is to step through both arrays a character at a time until we reach the end, or find a significant difference. This suggests generalising the specification to compare the strings from a given index onwards:

$$\mathsf{COMP}(i, j) \;=_{\text{DF}}\; \textbf{if}\ \mathsf{strip}(a, i) = \mathsf{strip}(b, j)\ \textbf{then}\ R := 1\ \textbf{else}\ R := 0\ \textbf{fi}$$

Elaborated Specification

The obvious special cases to consider are the values of $a[i]$ and $b[j]$.

Elaborated Specification

The obvious special cases to consider are the values of $a[i]$ and $b[j]$.

First we consider the case where $a[i] = \text{space}$:

if $a[i] = \text{space}$ **then** COMP(i, j)
else COMP(i, j) **fi**

Elaborated Specification

The obvious special cases to consider are the values of $a[i]$ and $b[j]$.

First we consider the case where $a[i] = \text{space}$:

if $a[i] = \text{space}$ **then** $\text{COMP}(i, j)$
else $\text{COMP}(i, j)$ **fi**

By definition, if $a[i] = \text{space}$ then $\text{strip}(a, i) = \text{strip}(a, i + 1)$ so $\text{COMP}(i, j) \approx \text{COMP}(i + 1, j)$.

We have:

if $a[i] = \text{space}$ **then** $\text{COMP}(i + 1, j)$
else $\text{COMP}(i, j)$ **fi**

Recursive Implementation

Let i' be the first index such that $a[i] = \text{end}$.

Then the variant function $i' - i$ is reduced before the first copy of the specification, but (obviously) not before the second copy.

We can still apply Recursive_Implementation, provided we only apply it to the *first* copy of the specification:

proc comp $\equiv$
  if $a[i] = \text{space}$ **then** $i := i + 1;$ comp
    else COMP(i, j) **fi**

Recursion Removal

This simple tail-recursion is transformed to a **while** loop:

while $a[i] = \text{space}$ **do** $i := i + 1$ **od**;
COMP(i, j)

Recursion Removal

This simple tail-recursion is transformed to a **while** loop:

while $a[i] = \text{space}$ **do** $i := i + 1$ **od**;
COMP(i, j)

A similar argument for $b[j]$ produces:

while $a[i] = \text{space}$ **do** $i := i + 1$ **od**;
while $b[j] = \text{space}$ **do** $j := j + 1$ **od**;
COMP(i, j)

Further Refinement

Consideration of the cases where $a[i] = \text{end}$ and/or $b[j] = \text{end}$ gives:

while $a[i] = \text{space}$ **do** $i := i + 1$ **od**;
while $b[j] = \text{space}$ **do** $j := j + 1$ **od**;
if $a[i] = \text{end} \wedge b[j] = \text{end}$ **then** $R := 1$
elsif $a[i] \neq a[j]$ **then** $R := 0$
else $i := i + 1;\ j := j + 1;$ COMP(i, j) **fi**

Final Program

Apply Recursive_Implementation and Recursion_Removal to get the final iterative program:

do while $a[i] =$ space **do** $i := i + 1$ **od**;
  while $b[j] =$ space **do** $j := j + 1$ **od**;
  if $a[i] =$ end $\wedge$ $b[j] =$ end **then** $R := 1$; **exit**(1)
  elsif $a[i] \neq a[j]$ **then** $R := 0$; **exit**(1) **fi**;
  $i := i + 1$; $j := j + 1$ **od**

Greatest Common Divisor

The Greatest Common Divisor (GCD) of two numbers is the largest number which divides both of the numbers with no remainder.

A specification for a program which computes the GCD is the following:

$$r := \mathsf{GCD}(x, y)$$

where:

$$\mathsf{GCD}(x, y) = \mathsf{max}\,\{\, n \in \mathbb{N} \mid x \bmod n = 0 \ \wedge \ y \bmod n = 0 \,\}$$

(Note that this is undefined when both x and y are zero).

Implementation Ideas

It is easy to prove the following facts about GCD:

Implementation Ideas

It is easy to prove the following facts about GCD:

1. $\mathsf{GCD}(0, y) = y$

Implementation Ideas

It is easy to prove the following facts about GCD:

1. $\text{GCD}(0, y) = y$

2. $\text{GCD}(x, 0) = x$

Implementation Ideas

It is easy to prove the following facts about GCD:

1. $\text{GCD}(0, y) = y$
2. $\text{GCD}(x, 0) = x$
3. $\text{GCD}(x, y) = \text{GCD}(y, x)$

Implementation Ideas

It is easy to prove the following facts about GCD:

1. $\text{GCD}(0, y) = y$
2. $\text{GCD}(x, 0) = x$
3. $\text{GCD}(x, y) = \text{GCD}(y, x)$
4. $\text{GCD}(x, y) = \text{GCD}(-x, y) = \text{GCD}(x, -y)$

Implementation Ideas

It is easy to prove the following facts about GCD:

1. GCD$(0, y) = y$
2. GCD$(x, 0) = x$
3. GCD$(x, y) =$ GCD(y, x)
4. GCD$(x, y) =$ GCD$(-x, y) =$ GCD$(x, -y)$
5. GCD$(x, y) =$ GCD$(x - y, y) =$ GCD$(x, y - x)$

Specification Elaboration

Split on the conditions $x = 0$ and $y = 0$, using Fact (1) and Fact (2) respectively to show $r := \mathsf{GCD}(x, y)$ is refined by:

if $x = 0$ **then** $r := y$

elsif $y = 0$ **then** $r := x$

else $r := \mathsf{GCD}(x, y)$ **fi**

Specification Elaboration

Split on the conditions $x = 0$ and $y = 0$, using Fact (1) and Fact (2) respectively to show $r := \text{GCD}(x, y)$ is refined by:

if $x = 0$ **then** $r := y$
elsif $y = 0$ **then** $r := x$
else $r := \text{GCD}(x, y)$ **fi**

Fact (3) does not appear to make much progress.

Specification Elaboration

Split on the conditions $x = 0$ and $y = 0$, using Fact (1) and Fact (2) respectively to show $r := \text{GCD}(x, y)$ is refined by:

if $x = 0$ **then** $r := y$
elsif $y = 0$ **then** $r := x$
else $r := \text{GCD}(x, y)$ **fi**

Fact (3) does not appear to make much progress.

If we restrict attention to non-negative integers, then Fact (4) does not apply. So we are left with Fact (5).

Specification Elaboration

Split on the conditions $x = 0$ and $y = 0$, using Fact (1) and Fact (2) respectively to show $r := \mathsf{GCD}(x, y)$ is refined by:

if $x = 0$ **then** $r := y$
elsif $y = 0$ **then** $r := x$
else $r := \mathsf{GCD}(x, y)$ **fi**

Fact (3) does not appear to make much progress.

If we restrict attention to non-negative integers, then Fact (4) does not apply. So we are left with Fact (5).

We can only transform $r := \mathsf{GCD}(x, y)$ to $r := \mathsf{GCD}(x - y, y)$ under the condition $x \geqslant y$.

Specification Elaboration

Split on the conditions $x = 0$ and $y = 0$, using Fact (1) and Fact (2) respectively to show $r := \mathsf{GCD}(x, y)$ is refined by:

if $x = 0$ **then** $r := y$
elsif $y = 0$ **then** $r := x$
else $r := \mathsf{GCD}(x, y)$ **fi**

Fact (3) does not appear to make much progress.

If we restrict attention to non-negative integers, then Fact (4) does not apply. So we are left with Fact (5).

We can only transform $r := \mathsf{GCD}(x, y)$ to $r := \mathsf{GCD}(x - y, y)$ under the condition $x \geqslant y$.

Similarly, we can only transform $r := \mathsf{GCD}(x, y)$ to $r := \mathsf{GCD}(x, y - x)$ under the condition $y \geqslant x$.

Elaborated Specification

We have the following elaboration of the specification:

if $x = 0$
  then $r := y$
elsif $y = 0$
    then $r := x$
elsif $x \geqslant y$ **then** $r := \mathsf{GCD}(x - y, y)$
         else $r := \mathsf{GCD}(x, y - x)$ **fi**

The variant function $x + y$ is reduced before each copy of the specification.

Recursion Introduction

Applying the recursion introduction gives:

proc gcd$(x, y)\ \equiv$
  if $x = 0$
    then $r := y$
  elsif $y = 0$
      then $r := x$
  elsif $x \geqslant y$ **then** $r :=$ gcd$(x - y, y)$
            else $r :=$ gcd$(x, y - x)$ **fi** **end**

Recursion Removal gives:

proc gcd$(x, y)\ \equiv$
  while $x \neq 0 \ \wedge\ y \neq 0$ **do**
    if $x \geqslant y$ **then** $x := x - y$
            else $y := y - x$ **fi** **od**;
  if $x = 0$ **then** $r := y$ **else** $r := x$ **fi** **end**

Optimisation

This algorithm, although correct, is very inefficient when x and y are very different in size. For example, if $x = 1$ and $y = 2^{31}$ then the program will take $2^{31} - 1$ steps.

One solution is to look for other properties of GCD to use: this involves throwing away all our work so far.

Unfortunately, this is the *only* option offered by the "Invariant Based Programming" approach.

Optimisation

With the transformational programming approach, we have another option: transform the program in order to improve its efficiency.

Apply Entire Loop Unrolling to the program at the point just after the assignment $x := x - y$ with the condition $x \geqslant y$:

proc gcd(x, y) $\equiv$
    while $x \neq 0 \wedge y \neq 0$ **do**
        if $x \geqslant y$ **then** $x := x - y$;
                        while $x \geqslant y$ **do**
                            if $x \geqslant y$ **then** $x := x - y$ **fi od**
                  else $y := y - x$ **fi od**;
    $r := x$ **end**

Optimisation

This simplifies to:

proc gcd$(x, y) \;\equiv$

  while $x \neq 0 \wedge y \neq 0$ **do**

    if $x \geqslant y$ **then** **while** $x \geqslant y$ **do** $x := x - y$ **od**

          else $y := y - x$ **fi** **od**;

  if $x = 0$ **then** $r := y$ **else** $r := x$ **fi** **end**

Consider the inner **while** loop. This repeatedly subtracts y from x.

If the loop executes q times, then $x = x_0 - q.y$.

In other words:

while $x \geqslant y$ **do** $x := x - y$ **od** $\;\approx\; x := x \bmod y$

Optimisation

Similarly, entire loop unrolling can be applied after the assignment $y := y - x$ and the same optimisation applied to give:

proc gcd$(x, y) \;\equiv$

    while $x \neq 0 \;\wedge\; y \neq 0$ **do**

        if $x \geqslant y$ **then** $x := x \bmod y$

                    else $y := y \bmod x$ **fi od**;

    if $x = 0$ **then** $r := y$ **else** $r := x$ **fi end**

Alternate Program Derivation

With different informal ideas, the same derivation process can derive a different algorithm.

For example, suppose the target machine does not have an efficient integer division instruction, but does have a binary shift.

We make use of the following additional facts about GCD:

1. $\text{GCD}(x, y) = 2.\text{GCD}(x/2, y/2)$ when x and y are both even;
2. $\text{GCD}(x, y) = \text{GCD}(x/2, y)$ when x is even and y is odd;
3. $\text{GCD}(x, y) = \text{GCD}(x, y/2)$ when x is odd and y is even;
4. $\text{GCD}(x, y) = \text{GCD}((x - y)/2, y)$ when x and y are odd and $x \geqslant y$;
5. $\text{GCD}(x, y) = \text{GCD}(x, (y - x)/2)$ when x and y are odd and $y \geqslant x$.

Elaborated Specification

Applying Fact (1) above produces:

if $x = 0$ **then** $r := y$

elsif $y = 0$ **then** $r := x$

elsif even?(x) $\wedge$ even?(y)

then $r := 2.$GCD$(x/2, y/2)$

else $r :=$ GCD(x, y) **fi**

Recursive Implementation

Applying the recursive implementation theorem plus recursion removal to the first occurrence only of GCD produces:

if $x = 0$
  then $r := y$
elsif $y = 0$
    then $r := x$
    else var $\langle L := 0 \rangle$:
        while even?(x) $\wedge$ even?(y) **do**
          $L := L + 1;$
          $x := x/2;\ y := y/2$ **od**;
        $r :=$ GCD(x, y);
        $r := 2^L.r$ **end fi**

Recursive Implementation

Applying Fact (1) above, followed by recursion introduction and recursion removal produces the following result:

if $x = 0$ **then** $r := y$
elsif $y = 0$ **then** $r := x$
else var $\langle L := 0 \rangle$:
  while even?(x) $\wedge$ even?(y) **do**
    $L := L + 1$;
    $x := x/2$; $y := y/2$ **od**;
  while even?(x) **do** $x := x/2$ **od**;
  $\{x \neq 0 \ \wedge \ y \neq 0 \ \wedge \ \neg$even?$(x)\}$;
  $r :=$ GCD(x, y);
  $r := 2^L.r$ **end fi**

Recursive Implementation

Define:

$$\mathsf{GCDx}(x, y) \;=_{\mathsf{DF}}\; \{y \neq 0 \;\wedge\; \neg\mathsf{even?}(x)\};\; r := \mathsf{GCD}(x, y)$$

By Fact (3) we show that $\mathsf{GCDx}(x, y)$ is equivalent to:

while even?(y) **do** $y := y/2$ **od**;

$\mathsf{GCDx}(x, y)$

Recursive Implementation

Now apply Fact (4), and also Fact (3) from the first set of facts, to ensure that x is odd in every occurrence of GCDx:

while even?(y) **do** $y := y/2$ **od**;
if $x = y$ **then** $r := x$
elsif $x > y$ **then** GCDx$(y, (x - y)/2)$
else GCDx$(x, (y - x)/2)$ **fi**

Recursive Implementation

Apply recursion introduction and recursion removal to derive this implementation of GCDx(x, y):

do while even?(y) **do** $y := y/2$ **od**;

  if $x = y$ **then** $r := x$; **exit fi**;

  if $x > y$

   then $\langle x, y \rangle := \langle y, x - y \rangle$

   else $y := y - x$ **fi**;

  $y := y/2$ **od**

Recursive Implementation

The final program is therefore:

if $x = 0$ **then** $r := y$
elsif $y = 0$ **then** $r := x$
else var $\langle L := 0 \rangle$:
while even?(x) $\wedge$ even?(y) **do**
$L := L + 1;$
$x := x/2;\ y := y/2$ **od**;
while even?(x) **do** $x := x/2$ **od**;
do while even?(y) **do** $y := y/2$ **od**;
if $x = y$ **then** $r := x$; **exit fi**;
if $x > y$
then $\langle x, y \rangle := \langle y, x - y \rangle$
else $y := y - x$ **fi**;
$y := y/2$ **od**
$r := 2^L.r$ **end fi**