Provably Correct Transformation of Specifications into Programs

Martin Ward

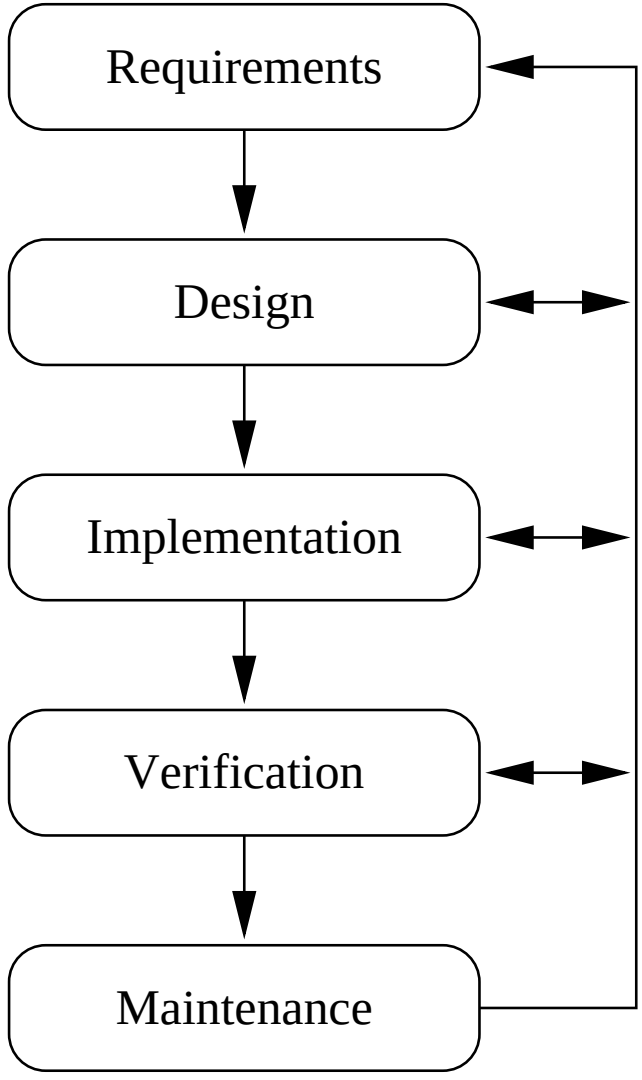
martin@gkc.org.uk

Software Technology Research Lab
De Montfort University
Gateway House,
Leicester LE1 9BH, UK

The Waterfall Model

1. **Requirements Elicitation**: analyse the problem domain and determine from the users 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 debug as needed;
5. **Maintenance**: any modifications required after delivery to correct faults, improve performance, or adapt the product to a modified environment.

The Waterfall Model



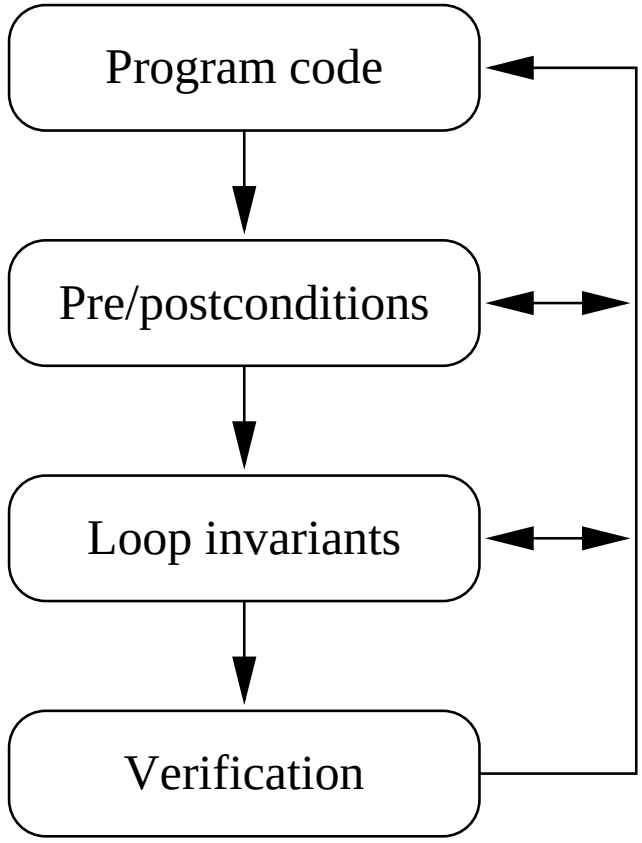
Proving the Correctness of a Program

Testing shows the presence, not the absence of bugs. — E.W.Dijkstra

Proving the correctness of a program has three requirements:

1. A precise mathematical specification which defines what the program is supposed to do
2. A definition of the semantics of the programming language
3. A mathematical proof that the program satisfies the specification

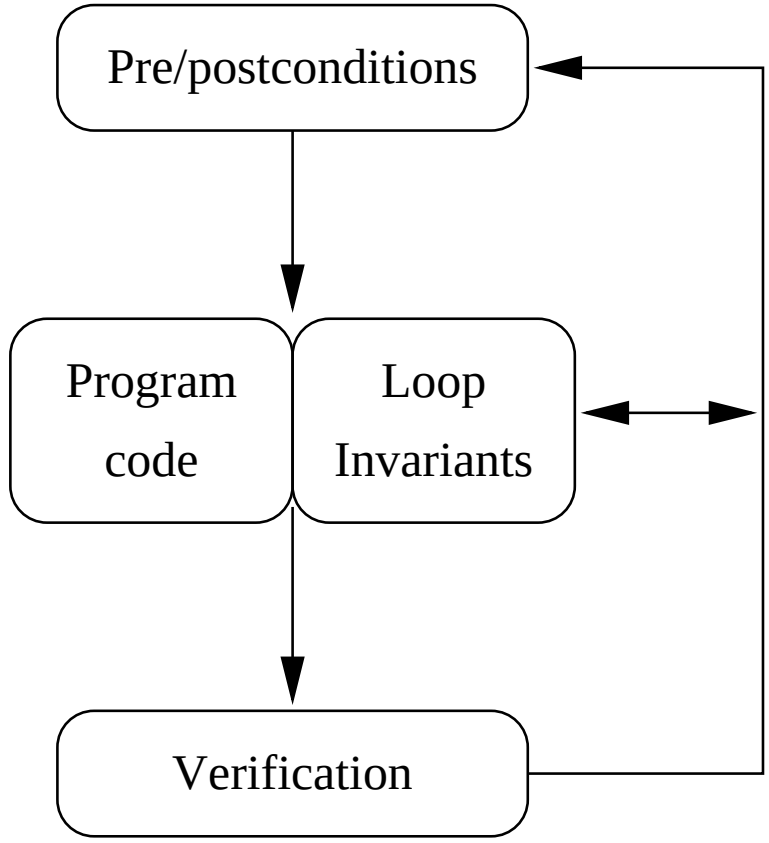
Program Verification



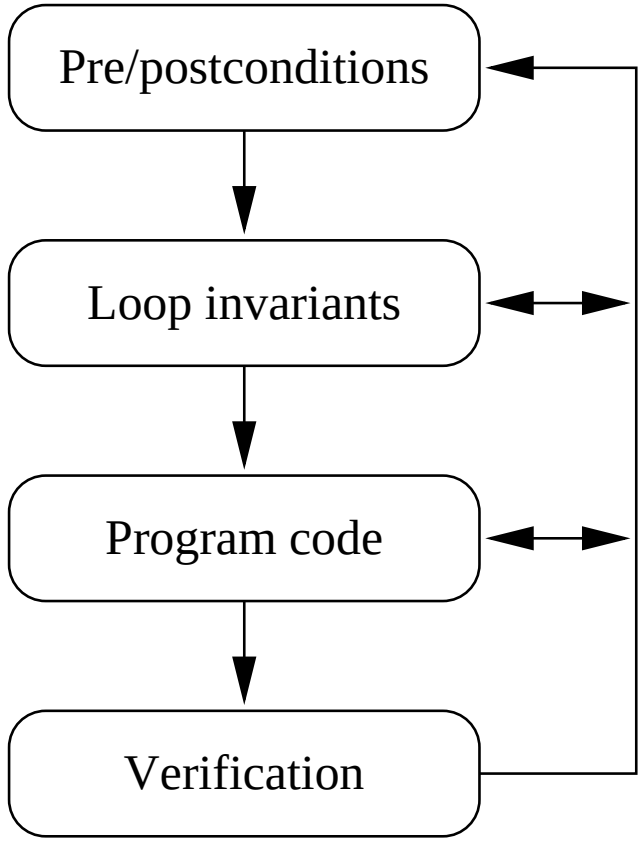
Loop Invariants

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



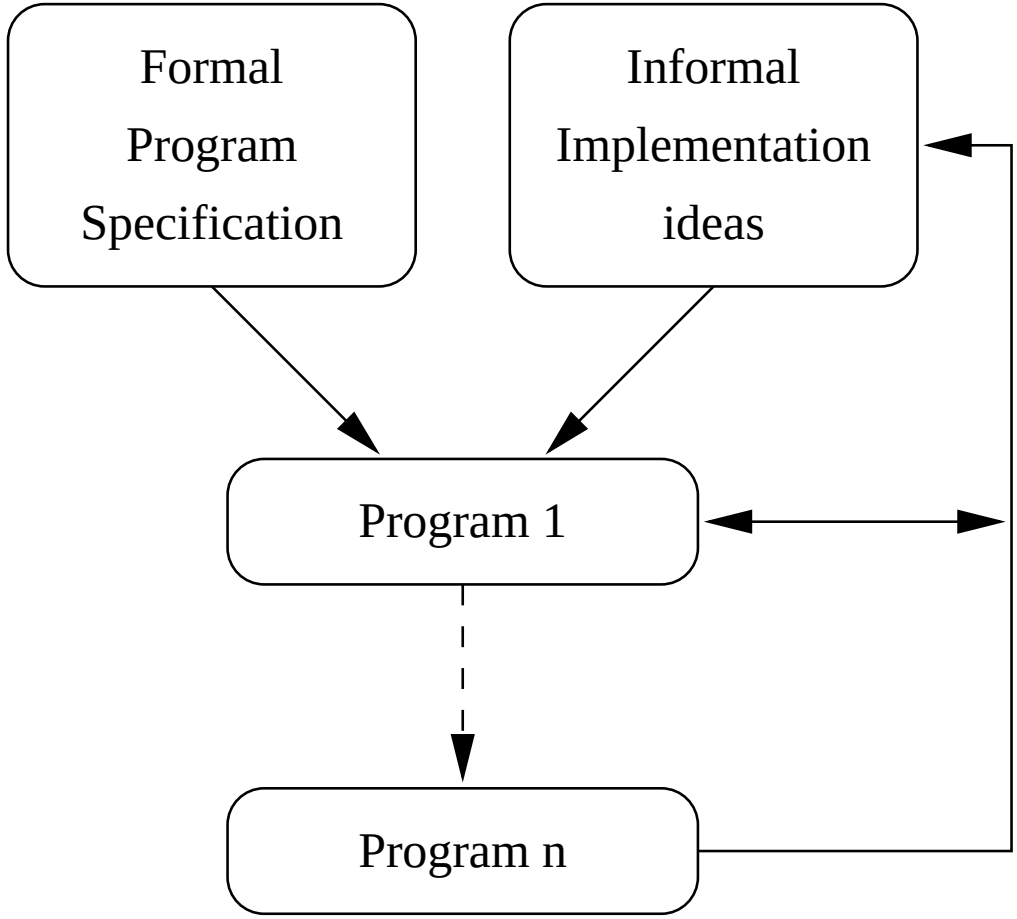
Invariant Based Development



Common Features

- Verification is the final step
- Up to this point, the program may be incomplete or incorrect
- Introducing a loop requires developing a loop invariant and variant expression

Algorithm Derivation



Algorithm Derivation

1. **Formal Specification:** Develop a WSL specification statement

Algorithm Derivation

1. **Formal Specification:** Develop a WSL specification statement
2. **Elaboration:** Take out simple case by transforming the specification and refining

Algorithm Derivation

1. **Formal Specification:** Develop a WSL specification statement
2. **Elaboration:** Take out simple case by transforming the specification and refining
3. **Divide and Conquer:** Use the informal ideas to divide up the general case with further transformations

Algorithm Derivation

1. **Formal Specification:** Develop a WSL specification statement
2. **Elaboration:** Take out simple case by transforming the specification and refining
3. **Divide and Conquer:** Use the informal ideas to divide up the general case with further transformations
4. **Recursion Introduction:** Apply the Recursive Implementation Theorem to produce a recursive program with no remaining copies of the specification.

Algorithm Derivation

1. **Formal Specification:** Develop a WSL specification statement
2. **Elaboration:** Take out simple case by transforming the specification and refining
3. **Divide and Conquer:** Use the informal ideas to divide up the general case with further transformations
4. **Recursion Introduction:** Apply the Recursive Implementation Theorem to produce a recursive program with no remaining copies of the specification.
5. **Recursion Removal:** If an iterative implementation is required, apply the Generic Recursion Removal Theorem.

Algorithm Derivation

1. **Formal Specification:** Develop a WSL specification statement
2. **Elaboration:** Take out simple case by transforming the specification and refining
3. **Divide and Conquer:** Use the informal ideas to divide up the general case with further transformations
4. **Recursion Introduction:** Apply the Recursive Implementation Theorem to produce a recursive program with no remaining copies of the specification.
5. **Recursion Removal:** If an iterative implementation is required, apply the Generic Recursion Removal Theorem.
6. **Optimisation:** Apply further optimising transformations as required.

Specification Statement

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 specification statement takes the form:

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q}$$

where:

- $\mathbf{x}$ is a list of the variables whose values may be altered;
- $\mathbf{x}'$ is the corresponding list of primed variables representing the new values; and
- $\mathbf{Q}$ is the condition which $\mathbf{x}$ and $\mathbf{x}'$ must satisfy.

If there is no assignment of values to $\mathbf{x}'$ which will satisfy $\mathbf{Q}$, then the statement aborts.

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.

Specification Statement

This statement:

$$\langle x, y \rangle := \langle x', y' \rangle . (x' = y \ \wedge \ y' = x)$$

swaps the values of variables x and y.

$$\langle x \rangle := \langle x' \rangle . (x' = x^2 - 2x + 2)$$

assigns x the value $x^2 - 2x + 2$.

$$\langle x \rangle := \langle x' \rangle . (x' = x'^2 - 2x' + 2)$$

will assign x the value 1 or 2.

$$\langle x \rangle := \langle x' \rangle . (x = x'^2 - 2x' + 2)$$

assigns x one of the two values: $1 \pm \sqrt{x^2 + 1}$

Specification Statement

A specification for sorting the array segment $A[a\,..\,b]$:

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

This defines precisely *what* a sorting program is required to achieve, without specifying *how* the result is to be accomplished.

It is not biased towards any particular sorting algorithm.

The WSL Language

- **Skip:** The statement **skip** terminates immediately
- **Abort:** The statement **abort** never terminates
- **Specification Statement**
- **Simple Assignment:** $v := e$ is defined as $\langle v \rangle := \langle v' \rangle.(v' = e)$
- **Deterministic Choice:** **if** $\mathbf{B}_1$ **then** $\mathbf{S}_1$ **elsif** $\mathbf{B}_2$ **then** $\mathbf{S}_2 \ldots$ **else** $\mathbf{S}_n$ **fi**
- **Nondeterministic Choice:** **if** $\mathbf{B}_1 \rightarrow \mathbf{S}_1 \Box \ldots \Box \mathbf{B}_n \rightarrow \mathbf{S}_n$ **fi**
- **While Loop:** **while B do S od**
- **Nondeterministic loop:** **do** $\mathbf{B}_1 \rightarrow \mathbf{S}_1 \Box \ldots \Box \mathbf{B}_n \rightarrow \mathbf{S}_n$ **od**
- **Floop:** **do S od**
- **Recursive Statement:** $(\mu X.\mathbf{S})$
- **Recursive procedures:**

Floop

do S od

The floop is an "unbounded" loop which is terminated by the execution of a statement of the form **exit**(n) where n is a positive integer (not a variable or expression). **exit**(n) causes immediate termination of the n enclosing levels of nested floops.

Recursive Procedures

A **where** statement contains a main body plus a collection of (possibly mutually recursive) procedures:

begin

S

where

proc $F_1(\mathbf{x}_1) \equiv \mathbf{S}_1$.

...

proc $F_n(\mathbf{x}_n) \equiv \mathbf{S}_n$.

end

Action System

An *Action System* is a set of parameterless mutually recursive procedures together with the name of the first action to be called.

actions A_1 :
$A_1 \equiv$ **S**$_1$**.**
$A_2 \equiv$ **S**$_2$**.**
...
$A_n \equiv$ **S**$_n$**. endactions**

where, in this case, A_1 is the starting action: so **S**$_1$ is the first statement to be executed. A statement of the form **call** A_i is a call to action A_i.

The statement **call** Z causes immediate termination of the whole action system.

Splitting a Tautology

If $\mathbf{B}_1 \vee \mathbf{B}_2$ is true then:

$$\mathbf{S} \approx \textbf{if } \mathbf{B}_1 \to \mathbf{S} \,\square\, \mathbf{B}_2 \to \mathbf{S} \textbf{ fi}$$

For any formula **B** we have:

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

Introduce Assertions

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

$$\textbf{if } B_1 \rightarrow S_1 \square B_2 \rightarrow S_2 \textbf{ fi} \approx \textbf{if } B_1 \rightarrow \{B_1\};\ S_1 \square B_2 \rightarrow \{B_2\};\ S_2 \textbf{ fi}$$

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

Assignment Merging

For any variable x and expressions e_1 and e_2:

$$x := e_1;\ x := e_2 \ \approx \ x := e_2[e_1/x]$$

Fold/Unfold

If **S** is any statement in an action system, one of whose actions is $A_i \equiv \mathbf{S}_i\mathbf{.}$, then:

$$\mathbf{S} \approx \mathbf{S}[\mathbf{S}_i/\mathbf{call}\ A_i]$$

Recursion Introduction

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 some (or all) copies of the original specification

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

Recursion Introduction

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 $\{\mathbf{P}\};\ \mathbf{S} \ \leq$ **proc** $X \ \equiv\ \mathbf{S}'$ **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

If an iterative implementation is required, then apply the Generic Recursion Removal Theorem to produce an iterative program.

Again: this can be carried out in stages, or only partially: for example, tail recursion can be converted to iteration, but inner recursive calls left in place.

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 $\mathbf{S}_{j0}$, $\mathbf{S}_{j1}$, $\ldots$,$\mathbf{S}_{jn_j-1}$ 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:

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

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

$\ldots B_j \equiv \mathbf{S}_{j0};$

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

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

$x := g_{j1};$

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

$\square \ldots \square$ $m = \langle j, k\rangle \rightarrow \mathbf{S}_{jk}[\textbf{call } \hat{F}/\textbf{call } Z]$; **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:

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

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

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

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

then call Z

else $L := L - 1;$

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

Linear Recursion Removal

For example:

proc $F \equiv$

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

is equivalent to the iterative program:

proc $F' \equiv$

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

actions A_1:

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

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

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

then call Z

else $L := L - 1$;

$\mathbf{S}_3$; **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.

Binary Search

Given a sorted array $A[1\,..\,n]$ and a value x, set r to a value $1 \leqslant r \leqslant n$ such that $A[r] = x$. Set $r = 0$ if x does not appear in A.

Binary Search

Given a sorted array $A[1\,..\,n]$ and a value x, set r to a value $1 \leqslant r \leqslant n$ such that $A[r] = x$. Set $r = 0$ if x does not appear in A.

Binary search is a fundamental algorithm which is very simple, but nearly everyone gets it wrong. There are subtle details in the implementation which are easy to overlook in an informal development.

Binary Search

Given a sorted array $A[1\,..\,n]$ and a value x, set r to a value $1 \leqslant r \leqslant n$ such that $A[r] = x$. Set $r = 0$ if x does not appear in A.

Binary search is a fundamental algorithm which is very simple, but nearly everyone gets it wrong. There are subtle details in the implementation which are easy to overlook in an informal development.

The informal idea is to pick an element in the middle of the array and examine its value. If this is equal to x, then we have finished, otherwise we can use the fact that the array is sorted to narrow down the area to be searched.

Binary Search Specification

Define the specification statement SEARCH(a, b) as:

$$\langle r \rangle := \langle r' \rangle.(a \leqslant r' \leqslant b \ \wedge \ A[r'] = x \ \vee \ r = 0 \ \wedge \ x \notin A[a \,..\, b])$$

where $1 \leqslant a, b \leqslant n$.

Binary Search Elaboration

If $a > b$ then the segment is empty, so x cannot be present:

$$\{a > b\};\ \text{SEARCH}(a, b) \ \approx\ \{a > b\};\ r := 0$$

If $a = b$ then the segment contains a single element, which can be tested against x:

$$\{a = b\};\ \text{SEARCH}(a, b) \ \approx\ \{a = b\};\ \textbf{if}\ A[a] = x\ \textbf{then}\ r := a\ \textbf{else}\ r := 0\ \textbf{fi}$$

So the elaborated specification is:

if $a > b$ **then** $r := 0$
elsif $a = b$ **then if** $A[a] = x$ **then** $r := a$ **else** $r := 0$ **fi**
else $\{a < b\};\ \text{SEARCH}(a, b)$ **fi**

Binary Search Divide and Conquer

Informal idea: Pick an element from the middle of the array and compare against x.

Since the array is sorted, if $x \leqslant A[m]$ then if there is an index r such that $A[r] = x$, then there will be such an index in the range $a\,..\,m$.

Conversely, if $x > A[m]$ then x cannot appear in the range $a\,..\,m$, so must be in the range $m+1\,..\,b$ if it appears at all.

Therefore:

$$\{x \leqslant A[m]\};\ \mathsf{SEARCH}(a, b) \ \approx\ \{x \leqslant A[m]\};\ \mathsf{SEARCH}(a, m)$$

and

$$\{x > A[m]\};\ \mathsf{SEARCH}(a, b) \ \approx\ \{x > A[m]\};\ \mathsf{SEARCH}(m+1, b)$$

Binary Search Divide and Conquer

Putting these results together SEARCH(a, b) is equivalent to:

if $a > b$ **then** $r := 0$
elsif $a = b$ **then if** $x = A[a]$ **then** $r := a$ **else** $r := 0$ **fi**
else pick any m in the range $a \,..\, b-1$
if $x \leqslant A[m]$ **then** SEARCH(a, m)
else SEARCH$(m+1, b)$ **fi**

Now:

$$m - a < b - a$$

and:

$$b - (m+1) < b - a$$

So we can apply the Recursive Implementation Theorem.

Recursive Implementation

proc search(a, b) $\equiv$
  if $a > b$ **then** $r := 0$
  elsif $a = b$ **then if** $x = A[a]$ **then** $r := a$ **else** $r := 0$ **fi**
    else pick any m in the range $a \,..\, b-1$
      if $x \leqslant A[m]$ **then** search(a, m)
        else search$(m+1, b)$ **fi end**

We can minimise the number of recursive calls by picking the middle element:

var $\langle m := (a+b)/2 \rangle$:
  if $x \leqslant A[m]$ **then** search(a, m)
    else search$(m+1, b)$ **fi end**

Binary Search Recursion Removal

Convert the tail-recursion to a loop:

proc search(a, b) $\equiv$

do if $a > b$ **then** $r := 0$; **exit**

elsif $a = b$ **then if** $x = A[a]$ **then** $r := a$ **else** $r := 0$ **fi**; **exit**

else var $\langle m := (a + b)/2 \rangle$:

if $x \leqslant A[m]$ **then** $b := m$

else $a := m + 1$ **fi end fi od end**

Binary Search Optimisation

Take the termination code out of the loop, and convert to a **while** loop:

proc search(a, b) $\equiv$
  while $a < b$ **do**
    var $\langle m := (a + b)/2 \rangle$:
      if $x \leqslant A[m]$ **then** $b := m$
        else $a := m + 1$ **fi end od**;
  if $a > b$ **then** $r := 0$
  elsif $x = A[a]$ **then** $r := A[a]$
    else $r := 0$ **fi end**

Advantages

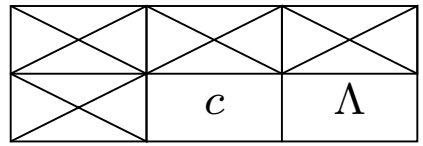
- At every stage we are working with a correct program
- No need to derive invariants
- No need for complex induction proofs
- For much of the process, the program has no loops or recursion
- No "verification" step: the final program is correct by construction
- Treat correctness and efficiency separately
- Method scales up to large and complex programs

Case Study: Polynomial Addition

Fields

UP	EXP	RIGHT
LEFT	CV	DOWN

Poly $= c$ (constant)



Poly $= g_0 + g_1 x^{e_1} + ... + g_n x^{e_n}$

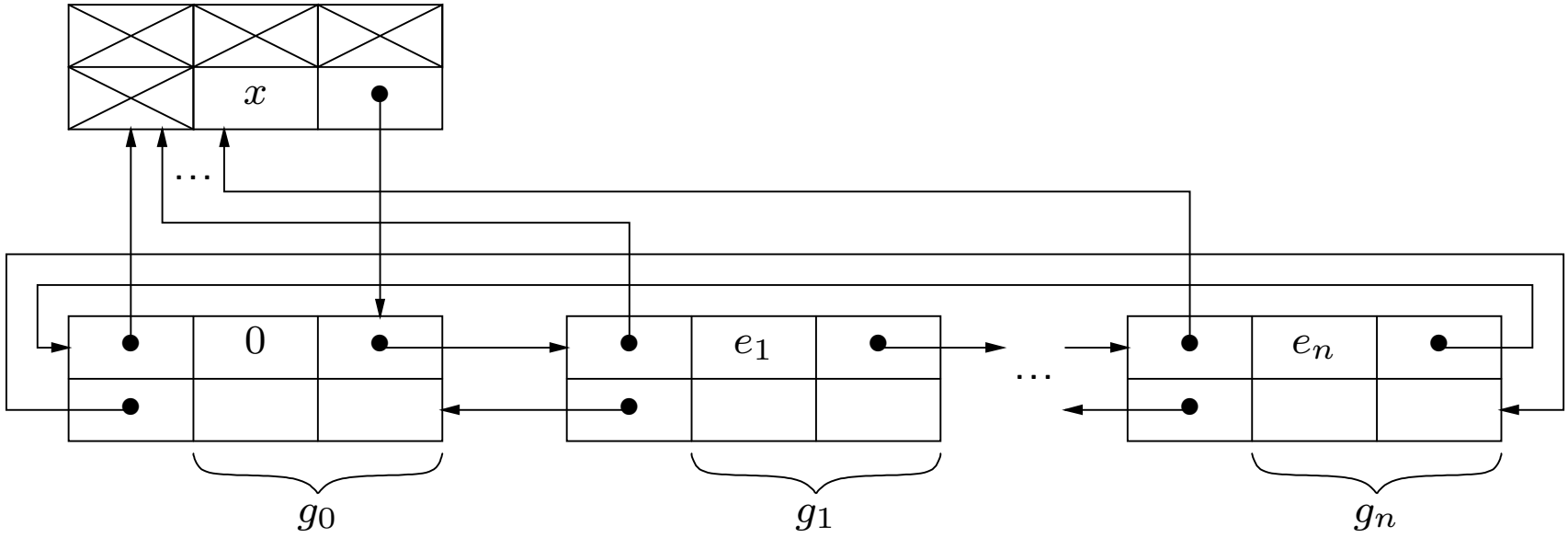


Figure 1: Representation of polynomials using four-directional links

Knuth's Algorithm (1 of 3)

proc Knuth_Add(P, Q) $\equiv$
  actions ADD :
  ADD $\equiv$ [Test type of polynomial.]
    if $D[P] = \Lambda$
      then while $D[Q] \neq \Lambda$ **do** $Q := D[Q]$ **od**; **call** A3
      else if $D[Q] = \Lambda \vee$ String_Less?$(C[Q], C[P])$
        then call A2
        elsif $C[Q] = C[P]$
          then $\langle P := D[P], Q := D[Q] \rangle$; **call** ADD
          else $Q := D[Q]$; **call** ADD **fi fi.**
  A2 $\equiv$ [Downward insertion.]
    $R_1 \xleftarrow{\text{pop}}$ Avail;
    $S := D[Q]$;
    if $S \neq \Lambda$
      then do $U[S] := R_1$; $S := R[S]$;
        if $E[S] = 0$ **then exit**(1) **fi od fi**;
    $\langle U[R_1] := Q, D[R_1] := D[Q], L[R_1] := R_1, R[R_1] := R_1, C[R_1] := C[Q], E[R_1] := 0 \rangle$;
    $\langle C[Q] := C[P], D[Q] := R_1 \rangle$;
    call ADD.
  A3 $\equiv$ [Match found.]
    $C[Q] := C[Q] + C[P]$;
    if $C[Q] = 0 \wedge E[Q] \neq 0$
      then call A8 **fi**;
    if $E[Q] = 0$ **then call** A7 **fi**;
    call A4.

Knuth's Algorithm (2 of 3)

A4 $\equiv$ [Advance to left.]
$P := L[P]$;
if $E[P] = 0$ **then call** A6
else do $Q := L[Q]$;
if $E[Q] \leqslant E[P]$
then exit(1) **fi od**;
if $E[P] = E[Q]$ **then call** ADD **fi fi**;
call A5.

A5 $\equiv$ [Insert to right.]
$R_1 \xleftarrow{\text{pop}}$ Avail;
$\langle U[R_1] := U[Q], D[R_1] := \Lambda, L[R_1] := Q, R[R_1] := R[Q] \rangle$;
$L[R[R_1]] := R_1;\ R[Q] := R_1;$
$\langle E[R_1] := E[P], C[R_1] := 0 \rangle;\ Q := R_1;$
call ADD.

A6 $\equiv$ [Return upward.]
$P := U[P]$; **call** A7.

A7 $\equiv$ [Move Q up to right level.]
if $U[P] = \Lambda$
then call A11
else while $C[U[Q]] \neq C[U[P]]$ **do** $Q := U[Q]$ **od**;
call A4 **fi.**

A8 $\equiv$ [Delete zero term.]
$R_1 := Q;\ Q := R[R_1];\ S := L[R_1];\ R[S] := Q;\ L[Q] := S;$
Avail $\xleftarrow{\text{push}} R_1$;
if $E[L[P]] = 0 \ \wedge\ Q = S$
then call A9 **else call** A4 **fi.**

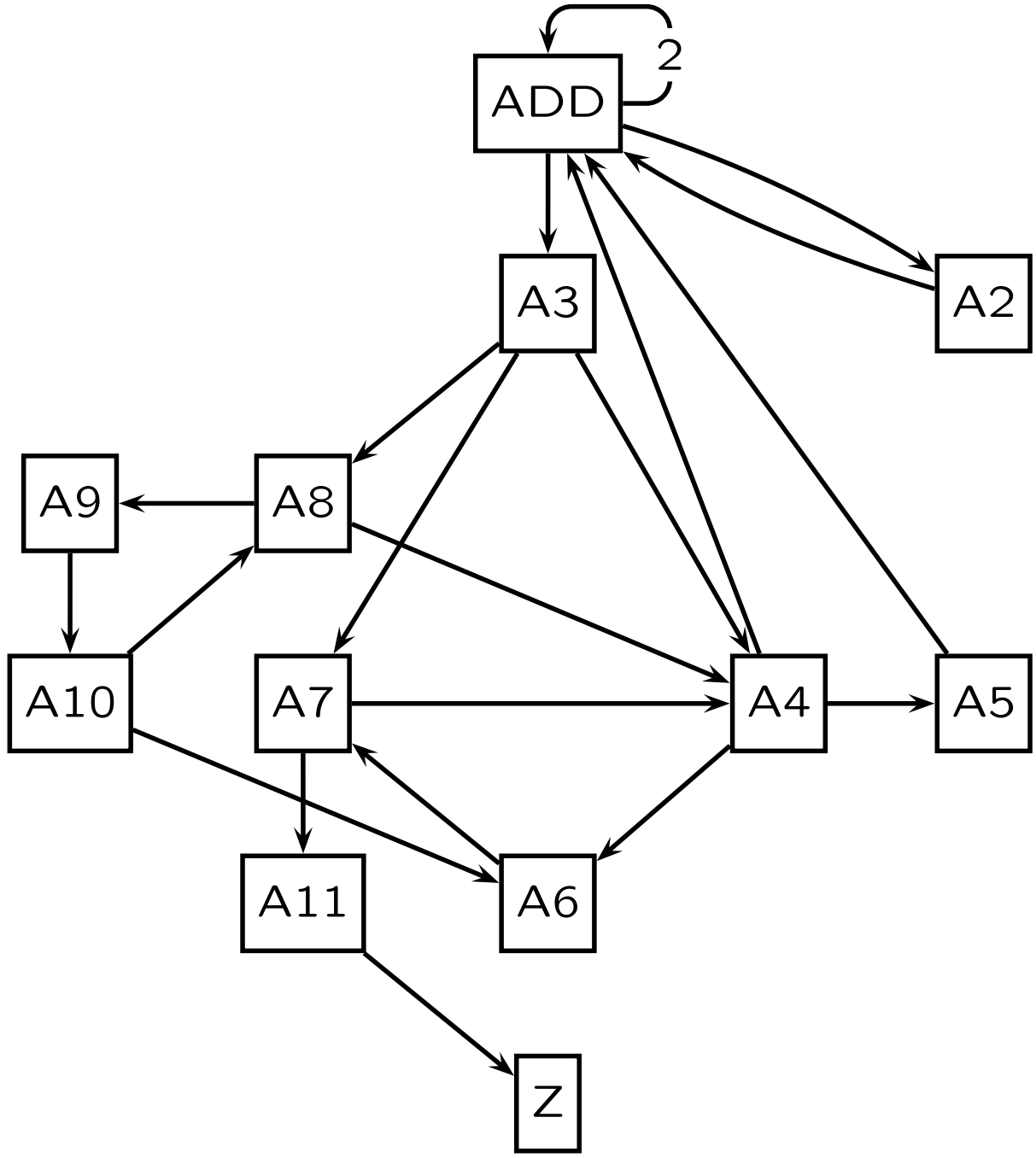
Knuth's Algorithm (3 of 3)

A9 $\equiv$ [Delete constant polynomial.]
$R_1 := Q;$
$Q := U[Q];$
$\langle D[Q] := D[R_1], C[Q] := C[R_1] \rangle;$
Avail $\overset{\text{push}}{\longleftarrow} R_1;$
$S := D[Q];$
if $S \neq \Lambda$ **then do** $U[S] := Q;\ S := R[S];$
if $E[S] = 0$ **then exit**(1) **fi od fi**;
call A10.

A10 $\equiv$ [Zero detected?]
if $D[Q] = \Lambda \ \wedge \ C[Q] = 0 \ \wedge \ E[Q] \neq 0$
then $P := U[P];$ **call** A8 **else call** A6 **fi.**

A11 $\equiv$ [Terminate.]
while $U[Q] \neq \Lambda$ **do** $Q := U[Q]$ **od**; **call** Z**. endactions.**

Call Graph for Knuth's Algorithm



Formal Specification

A constant polynomial is represented as the singleton list $\langle c \rangle$ where c is an integer. Otherwise, a polynomial is represented as the list of three or more elements,

$$p = \langle x, \langle g_0, e_0 \rangle, \langle g_1, e_1 \rangle, \ldots, \langle g_n, e_n \rangle \rangle$$

where x is a variable, $0 = e_0 < e_1 < \cdots < e_n$ are the integer exponents and g_i are polynomials.

For example $\langle x, \langle \langle 0 \rangle, 0 \rangle, \langle \langle 4 \rangle, 1 \rangle, \langle \langle 1 \rangle, 2 \rangle \rangle$ represents the polynomial:

$$0.x^0 + 4.x^1 + 1.x^2$$

which is:

$$x^2 + 4.x$$

Formal Specification

With these definitions, the formal specification for a program which assigns r to the value of a list which represents a polynomial equal to $\mathsf{abs}(p) + \mathsf{abs}(q)$ is simply:

$$\mathsf{add}(r, p, q) \ =_{\mathsf{DF}} \ r := r'.(I(r') \ \wedge \ \mathsf{abs}(r') = \mathsf{abs}(p) + \mathsf{abs}(q))$$

Formal Specification

The abstraction function $\mathsf{Abs}(P)$ maps the concrete polynomial P (an index into the arrays) to the corresponding abstract polynomial and is defined as follows:

$$\mathsf{Abs}(P) =_{\mathsf{DF}} \begin{cases} \langle C[P] \rangle & \text{if } D[P] = \Lambda \\ \langle C[P] \rangle +\!\!+ \mathsf{Abs\_Terms}(D[P]) & \text{otherwise} \end{cases}$$

where Abs_Terms returns the list of abstract polynomial terms:

$$\mathsf{Abs\_Terms}(P) =_{\mathsf{DF}} \begin{cases} \langle \langle C[P], E[P] \rangle \rangle & \text{if } E[R[P]] = 0 \\ \langle \langle C[P], E[P] \rangle \rangle +\!\!+ \mathsf{Abs\_Terms}(R[P]) & \text{otherwise} \end{cases}$$

Derived Algorithm

```
proc add( var ) ≡
  do do if D[P] = Λ then add_const; exit(1) fi;
        do if D[Q] = Λ
              then copy_and_add; exit(2)
           elsif C[P] < C[Q]
                 then Q := D[Q]
                  else exit(1) fi od;
        insert_below_Q;
        P := D[P];
        Q := D[Q] od;
     do if U[P] = Λ then exit(2) fi;
        while C[U[Q]] ≠ C[U[P]] do Q := U[Q] od;
        check_for_zero_term;
        do P := R[P];
           if E[P] = 0 then exit(1) fi;
           Q := R[Q];
           while E[Q] > 0 ∧ E[Q] < E[P] do Q := R[Q] od;
           if E[Q] = E[P] then exit(2) fi;
           insert_copy od;
        P := U[P];
        check_for_const_poly od od end
proc add_const(c, Q) ≡
  while D[Q] ≠ Λ do Q := D[Q] od;
  C[Q] := C[Q] + c.
```

Derived Algorithm

proc insert_below_Q $\equiv$
  if $C[Q] < C[P]$
    then C : Insert a node below Q and convert Q to a constant poly;
      var $\langle R_1 := 0, S := D[Q] \rangle$:
      $R_1 \overset{\text{pop}}{\longleftarrow}$ Avail; ;
      do $U[S] := R_1;\ S := R[S];$
        if $E[S] = 0$ **then exit**(1) **fi od**;
      $U[R_1] := Q;\ D[R_1] := S;\ L[R_1] := R_1;\ R[R_1] := R_1;$
      $C[R_1] := C[Q];\ E[R_1] := 0;\ C[Q] := C[P];\ D[Q] := R_1$ **end fi.**

proc check_for_zero_term $\equiv$
  C : Check for a zero term with non-zero exponent;
  if $E[Q] \neq 0 \ \wedge\ D[Q] = \Lambda \ \wedge\ C[Q] = 0$
    then var $\langle R_1 := Q, S := R[Q] \rangle$:
      $Q := L[Q];\ L[S] := Q;\ R[Q] := S;$
      Avail $\overset{\text{push}}{\longleftarrow} R_1$ **end fi.**

proc insert_copy $\equiv$
  C : Insert a copy of P to the left of Q;
  var $\langle R_1 := 0 \rangle$:
  $R_1 \overset{\text{pop}}{\longleftarrow}$ Avail;
  $L[R_1] := L[Q];\ R[R_1] := Q;\ U[R_1] := U[Q];\ E[R_1] := E[P];$
  $R[L[Q]] := R_1;\ L[Q] := R_1;\ Q := R_1$ **end**;
  copy(P, Q)**.**

Derived Algorithm

proc check_for_const_poly $\equiv$

C : Check for a constant poly;

if $R[Q] = Q$

then var $\langle R_1 := Q, S := 0 \rangle$:

$Q := U[Q];\ D[Q] := D[R_1];\ C[Q] := C[R_1];$

Avail $\overset{\text{push}}{\longleftarrow} R_1;$

$S := D[Q];$

if $S \neq \Lambda$

then do $U[S] := Q;\ S := R[S];$

if $E[S] = 0$ **then exit fi od fi end fi.**

Execution Timings

Vars	Terms	Count	Derived	Knuth	Ratio
7	4	1,000,000	10.77s	22.64s	2.10
7	4	10,000,000	109.14s	229.57s	2.10
2	4	100,000,000	34.46s	130.76s	3.79
2	20	10,000,000	14.38s	31.66s	2.20
7	10	10,00,000	158.68s	332.80s	2.10
7	20	400,000	1,271.43s	3,129.33s	2.46

Table 1: Execution times for the algorithms

Conclusion

Transformational Programming:

- Is a practical method for deriving complex algorithms
- Scales up to large programs via divide and conquer
- Generated code "works first time": correct by construction
- No need to invent loop invariants
- No verification stage
- Produces efficient implementations

Publications

The paper "Provably Correct Derivation of Algorithms Using FermaT" is available from:

http://www.gkc.org.uk/martin/papers/trans-prog-t.pdf

My other papers are also on my web site:

http://www.gkc.org.uk/martin/papers/