Formal Transformations and WSL

Part One

Martin Ward

Reader in Software Engineering

martin@gkc.org.uk

Software Technology Research Lab
De Montfort University

Models and Abstractions

Example of a model: aircraft decoys.

Models and Abstractions

Example of a model: aircraft decoys.

- Chaff — looks like an aircraft on radar

Models and Abstractions

Example of a model: aircraft decoys.

- Chaff — looks like an aircraft on radar
- Plywood model glider — visually looks like an aircraft

Models and Abstractions

Example of a model: aircraft decoys.

- Chaff — looks like an aircraft on radar
- Plywood model glider — visually looks like an aircraft
- . . . add a tape player and amplifier — sounds like an aircraft

Models and Abstractions

Example of a model: aircraft decoys.

- Chaff — looks like an aircraft on radar
- Plywood model glider — visually looks like an aircraft
- . . . add a tape player and amplifier — sounds like an aircraft
- . . . add a heat source — fools infra-red camera also

Models and Abstractions

Example of a model: aircraft decoys.

- Chaff — looks like an aircraft on radar
- Plywood model glider — visually looks like an aircraft
- . . . add a tape player and amplifier — sounds like an aircraft
- . . . add a heat source — fools infra-red camera also
- . . .

Models and Abstractions

Example of a model: aircraft decoys.

- Chaff — looks like an aircraft on radar
- Plywood model glider — visually looks like an aircraft
- . . . add a tape player and amplifier — sounds like an aircraft
- . . . add a heat source — fools infra-red camera also
- . . .

The result is no longer a decoy: it is an actual aircraft!

Models and Abstractions

Example of a model: aircraft decoys.

- Chaff — looks like an aircraft on radar
- Plywood model glider — visually looks like an aircraft
- . . . add a tape player and amplifier — sounds like an aircraft
- . . . add a heat source — fools infra-red camera also
- . . .

The result is no longer a decoy: it is an actual aircraft!

Conclusion:

In order to be useful, a model must be *incomplete.*

Models and Abstractions

- Any mathematical analysis of a computer program must be an *abstraction*: some details are ignored and other features are included in the model.

Models and Abstractions

- Any mathematical analysis of a computer program must be an *abstraction*: some details are ignored and other features are included in the model.
- If *every* feature of reality were included in the model, then it would no longer be a model, but the thing itself!

Models and Abstractions

- Any mathematical analysis of a computer program must be an *abstraction*: some details are ignored and other features are included in the model.
- If *every* feature of reality were included in the model, then it would no longer be a model, but the thing itself!
- So: the right question to ask of any mathematical model or scientific theory is not "Does it account for everything?" but "Is it useful?". In other words, "Does it account for all the features of reality that we are interested in for this purpose?".

Models and Abstractions

- Any mathematical analysis of a computer program must be an *abstraction*: some details are ignored and other features are included in the model.
- If *every* feature of reality were included in the model, then it would no longer be a model, but the thing itself!
- So: the right question to ask of any mathematical model or scientific theory is not "Does it account for everything?" but "Is it useful?". In other words, "Does it account for all the features of reality that we are interested in for this purpose?".
- For example: In analysing a computer program for correctness, we are not interested in how long the program takes to process the input, or what sequence of internal states it goes through on the way to generating the result.

Formal Methods

Formal Methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems.

Classes of formal methods include:

- Algebraic specification (eg OBJ, LARCH)
- Model based (eg Z, VDM, B-Method and B-Toolkit)
- Logic based (eg Hoare Logic, Weakest Preconditions, Tempura)
- Net-based – graphical (eg Petri nets, State Charts)
- Process algebra (eg CSP, Lotos, Timed CSP)

Formal Methods

A simple equation:

$$x^2 - 3x + 2 = 0$$

We can think of an equation as *a description of a set of values*, in this case, the values of x which satisfy the equation.

Formal Methods

A simple equation:

$$x^2 - 3x + 2 = 0$$

We can think of an equation as *a description of a set of values*, in this case, the values of x which satisfy the equation.

What are the values of x which satisfy this equation?

Formal Methods

A simple equation:

$$x^2 - 3x + 2 = 0$$

We can think of an equation as *a description of a set of values*, in this case, the values of x which satisfy the equation.

What are the values of x which satisfy this equation?

$$x = 1 \quad \text{and} \quad x = 2$$

Another Equation

Here is another equation:

$$(x-1)(x-2)=0$$

Another Equation

Here is another equation:

$$(x-1)(x-2)=0$$

What are the values of x which satisfy this equation?

Another Equation

Here is another equation:

$$(x-1)(x-2)=0$$

What are the values of x which satisfy this equation?

$$x=1 \quad \text{and} \quad x=2$$

Proving Equivalence

The two equations:

$$x^2 - 3x + 2 = 0$$

and:

$$(x - 1)(x - 2) = 0$$

define the *same* set.

Proving Equivalence

The two equations:

$$x^2 - 3x + 2 = 0$$

and:

$$(x - 1)(x - 2) = 0$$

define the *same* set.

There are (at least) two ways to prove that the sets are the same:

Proving Equivalence

The two equations:

$$x^2 - 3x + 2 = 0$$

and:

$$(x - 1)(x - 2) = 0$$

define the *same* set.

There are (at least) two ways to prove that the sets are the same:

1. Compute the expressions for all values of x and see for which values the equations are correct

Proving Equivalence

The two equations:

$$x^2 - 3x + 2 = 0$$

and:

$$(x - 1)(x - 2) = 0$$

define the *same* set.

There are (at least) two ways to prove that the sets are the same:

1. Compute the expressions for all values of x and see for which values the equations are correct
2. Start with one equation and use mathematical laws to turn it into the other equation.

Proving Equivalence

$$(x-1)(x-2) = x(x-2) - 1(x-2)$$

By the *Distributive Law*

$$= x^2 - 2x - 1x + 2$$

By the *Distributive* and *Associative Laws*

$$= x^2 - 3x + 2$$

Proving Equivalence

$$(x-1)(x-2) = x(x-2) - 1(x-2)$$

By the *Distributive Law*

$$= x^2 - 2x - 1x + 2$$

By the *Distributive* and *Associative Laws*

$$= x^2 - 3x + 2$$

So:

$$(x-1)(x-2) = 0$$

if and only if:

$$x^2 - 3x + 2 = 0$$

What is a Program?

What is a Program?

- A list of instructions for a machine

What is a Program?

- A list of instructions for a machine
- A piece of text which describes an operation to another person (which may also happen to be executable on a machine)

What is a Program?

- A list of instructions for a machine
- A piece of text which describes an operation to another person (which may also happen to be executable on a machine)
- A description of the desired behaviour of a machine

What is a Program?

- A list of instructions for a machine
- A piece of text which describes an operation to another person (which may also happen to be executable on a machine)
- A description of the desired behaviour of a machine
- A definition of a mathematical function

What is a Program?

- A list of instructions for a machine
- A piece of text which describes an operation to another person (which may also happen to be executable on a machine)
- A description of the desired behaviour of a machine
- A definition of a mathematical function
- A mathematical formula

What is a Program?

- A list of instructions for a machine
- A piece of text which describes an operation to another person (which may also happen to be executable on a machine)
- A description of the desired behaviour of a machine
- A definition of a mathematical function
- A mathematical formula
- A precise definition of the relationship between input and output states

What is a Program?

- A list of instructions for a machine
- A piece of text which describes an operation to another person (which may also happen to be executable on a machine)
- A description of the desired behaviour of a machine
- A definition of a mathematical function
- A mathematical formula
- A precise definition of the relationship between input and output states
- All the above!

What is a Program?

Sometimes a program can look like an equation:

```
y = x*x - 3*x + 2
```

This is not an equation, but an assignment: which is why some programming languages use $:=$ instead of $=$. For example:

$$x = x + 1$$

is an equation which has no solution, but:

$$x := x + 1$$

is a program which increments the value of x.

Formal Methods

Many formal methods treat a program as: *a definition of the relationship between input and output states*.

Some ways to prove that two different programs define the same relationship:

Formal Methods

Many formal methods treat a program as: *a definition of the relationship between input and output states*.

Some ways to prove that two different programs define the same relationship:

- Execute the two programs for all inputs and compare outputs

Formal Methods

Many formal methods treat a program as: *a definition of the relationship between input and output states*.

Some ways to prove that two different programs define the same relationship:

- Execute the two programs for all inputs and compare outputs
- Deduce enough properties of the programs which prove that they must be equivalent

Formal Methods

Many formal methods treat a program as: *a definition of the relationship between input and output states*.

Some ways to prove that two different programs define the same relationship:

- Execute the two programs for all inputs and compare outputs
- Deduce enough properties of the programs which prove that they must be equivalent
- Apply mathematical laws ("Laws of Programming") to one program to turn it into the other program

Formal Methods

Many formal methods treat a program as: *a definition of the relationship between input and output states*.

Some ways to prove that two different programs define the same relationship:

- Execute the two programs for all inputs and compare outputs
- Deduce enough properties of the programs which prove that they must be equivalent
- Apply mathematical laws ("Laws of Programming") to one program to turn it into the other program
- and so on...

Formal Methods

Many formal methods treat a program as: *a definition of the relationship between input and output states*.

Some ways to prove that two different programs define the same relationship:

- Execute the two programs for all inputs and compare outputs
- Deduce enough properties of the programs which prove that they must be equivalent
- Apply mathematical laws ("Laws of Programming") to one program to turn it into the other program
- and so on...

We should consider programs much more as manipulable objects which exist in different forms and which do well-defined things.
—M. Griffiths

Formal Methods

We can sometimes determine the behaviour of a program for an infinite set of inputs by using **mathematical induction**:

Formal Methods

We can sometimes determine the behaviour of a program for an infinite set of inputs by using **mathematical induction**:

1. **Base Step:** Show that the program produces the desired output for an input value of 0;

Formal Methods

We can sometimes determine the behaviour of a program for an infinite set of inputs by using **mathematical induction**:

1. **Base Step:** Show that the program produces the desired output for an input value of 0;
2. **Induction Step:** Prove that: *if* the program works for an input value of k, *then* it also works for $k+1$;

Formal Methods

We can sometimes determine the behaviour of a program for an infinite set of inputs by using **mathematical induction**:

1. **Base Step:** Show that the program produces the desired output for an input value of 0;
2. **Induction Step:** Prove that: *if* the program works for an input value of k, *then* it also works for $k+1$;
3. **Conclusion:** Deduce that the program works for all input values.

Formal Methods

We can sometimes determine the behaviour of a program for an infinite set of inputs by using **mathematical induction**:

1. **Base Step:** Show that the program produces the desired output for an input value of 0;
2. **Induction Step:** Prove that: *if* the program works for an input value of k, *then* it also works for $k+1$;
3. **Conclusion:** Deduce that the program works for all input values.

For example:

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

Prove that this is equivalent to $n := 0$ for all non-negative integer values of n.

Formal Methods

Prove that the following program:

$$
\begin{array}{l}
\langle \mathsf{total} := 0, i := 0 \rangle; \\
\textbf{while } i \neq n \textbf{ do} \\
\quad i := i + 1; \\
\quad \mathsf{total} := \mathsf{total} + A[i] \textbf{ od}
\end{array}
$$

will set total to the value $A[1] + \cdots + A[n]$, i.e.:

$$\sum_{1 \leqslant k \leqslant n} A[k]$$

Formal Methods

One way to prove this is uses two things:

An Invariant Condition: a condition (a formula) which is true just before the loop and is preserved by the body of the loop; and

A Variant Expression: a positive valued expression whose value is reduced on each iteration of the loop.

Formal Methods

One way to prove this is uses two things:

An Invariant Condition: a condition (a formula) which is true just before the loop and is preserved by the body of the loop; and

A Variant Expression: a positive valued expression whose value is reduced on each iteration of the loop.

Invariant Formula: let I be the condition total $= A[1] + \cdots + A[i]$ i.e.:

$$\text{total} = \sum_{1 \leqslant k \leqslant i} A[k]$$

If $i = 0$, then the sum has no elements, so is zero. Setting total $:= 0$ and $i := 0$ will make condition I true (trivially).

Preserving the Invariant

The invariant is true before the loop.

Now consider the body of the loop. If:

$$\mathsf{total} = \sum_{1 \leqslant k \leqslant i} A[k]$$

then:

$$\mathsf{total} + A[i+1] = \sum_{1 \leqslant k \leqslant i+1} A[k]$$

So, if we add 1 to i and then add $A[i]$ to total, then the condition I will still be true. This is what the body of the loop actually does.

If I is true at the start of the loop body, **then** I will still be true at the end of the loop.

The loop body preserves the invariant.

The Variant Expression

The variant expression is:

$$n - i$$

On each iteration of the loop, i is increased, so $n - i$ is reduced. The loop terminates when $i = n$, and n is a non-negative integer, so the expression $n - i$ is always greater than or equal to zero.

The Variant Expression

The variant expression is:

$$n - i$$

On each iteration of the loop, i is increased, so $n - i$ is reduced. The loop terminates when $i = n$, and n is a non-negative integer, so the expression $n - i$ is always greater than or equal to zero.

The purpose of the variant expression is to prove that each iteration of the loop makes some progress. This proves that loop will eventually terminate.

The Variant Expression

The variant expression is:

$$n - i$$

On each iteration of the loop, i is increased, so $n - i$ is reduced. The loop terminates when $i = n$, and n is a non-negative integer, so the expression $n - i$ is always greater than or equal to zero.

The purpose of the variant expression is to prove that each iteration of the loop makes some progress. This proves that loop will eventually terminate.

The variant expression proves that the loop will terminate.

Putting it all Together

$\langle \mathsf{total} := 0, i := 0 \rangle;$
while $i \neq n$ **do**
$\quad i := i + 1;$
$\quad \mathsf{total} := \mathsf{total} + A[i]$ **od**

Putting it all Together

$$
\begin{array}{l}
\langle \mathsf{total} := 0, i := 0 \rangle; \\
\textbf{while } i \neq n \textbf{ do} \\
\quad i := i + 1; \\
\quad \mathsf{total} := \mathsf{total} + A[i] \textbf{ od}
\end{array}
$$

The condition $\mathsf{total} = \sum_{1 \leqslant k \leqslant i} A[k]$ is true just before the loop, and is preserved by every iteration of the loop. So it will still be true when the loop terminates (if it terminates).

Putting it all Together

$$
\begin{array}{l}
\langle \mathsf{total} := 0, i := 0 \rangle; \\
\textbf{while } i \neq n \textbf{ do} \\
\quad i := i + 1; \\
\quad \mathsf{total} := \mathsf{total} + A[i] \textbf{ od}
\end{array}
$$

The condition $\mathsf{total} = \sum_{1 \leqslant k \leqslant i} A[k]$ is true just before the loop, and is preserved by every iteration of the loop. So it will still be true when the loop terminates (if it terminates).

The integer expression $n - i$ is reduced on every iteration of the loop and never goes negative. So the loop actually does terminate.

Putting it all Together

$\langle \mathsf{total} := 0, i := 0 \rangle;$
while $i \neq n$ **do**
  $i := i + 1;$
  $\mathsf{total} := \mathsf{total} + A[i]$ **od**

The condition $\mathsf{total} = \sum_{1 \leqslant k \leqslant i} A[k]$ is true just before the loop, and is preserved by every iteration of the loop. So it will still be true when the loop terminates (if it terminates).

The integer expression $n - i$ is reduced on every iteration of the loop and never goes negative. So the loop actually does terminate.

When the loop terminates, $i \neq n$ is false, so $i = n$ is true.

Putting it all Together

$$\langle \mathsf{total} := 0, i := 0 \rangle;$$
$$\textbf{while } i \neq n \textbf{ do}$$
$$\quad i := i + 1;$$
$$\quad \mathsf{total} := \mathsf{total} + A[i] \textbf{ od}$$

The condition $\mathsf{total} = \sum_{1 \leqslant k \leqslant i} A[k]$ is true just before the loop, and is preserved by every iteration of the loop. So it will still be true when the loop terminates (if it terminates).

The integer expression $n - i$ is reduced on every iteration of the loop and never goes negative. So the loop actually does terminate.

When the loop terminates, $i \neq n$ is false, so $i = n$ is true.

If $i = n$ and $\mathsf{total} = \sum_{1 \leqslant k \leqslant i} A[k]$ is true, then $\mathsf{total} = \sum_{1 \leqslant k \leqslant n} A[k]$ is true. Which is just what we wanted to prove.

Summary

To use the method of invariants to prove the correctness of a program containing a loop we need to find:

1. An invariant condition
2. A variant function

and then we need to prove:

1. The invariant is true just before the loop
2. The invariant is preserved by the loop body
3. The variant function is reduced by the loop body and bounded below
4. The invariant plus the terminating condition implies the required postcondition

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 Invariant

Suppose that the sub-array $A[a\,..\,b]$ has still to be searched: in other words, if x is anywhere in the array, then it must be in $A[a\,..\,b]$. Part of our invariant is therefore:

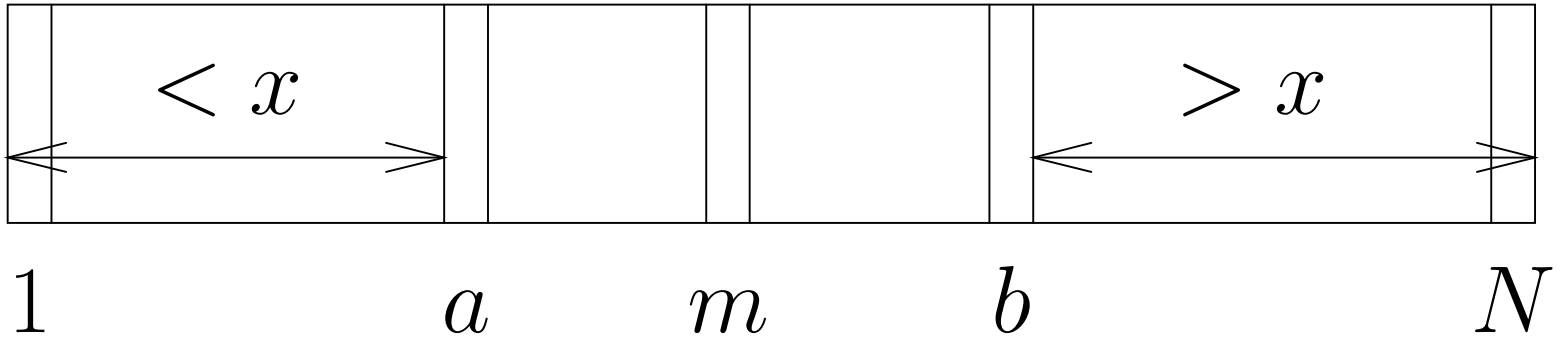
$$A[1\,..\,a-1] < x \;\wedge\; A[b+1\,..\,N] > x$$

We can ensure that I is true initially by assigning $a := 1$ and $b := N$. Then the two sub-arrays in the invariant are empty.

If x has been found, then we will set r to the appropriate index, so the full invariant I is:

$$(r > 0 \;\wedge\; A[r] = x) \;\vee\; (r = 0 \;\wedge\; A[1\,..\,a-1] < x \;\wedge\; A[b+1\,..\,N] > x)$$

Binary Search Invariant



Binary Search Variant Expression

The algorithm makes progress by reducing the size of the sub-array $A[a\,..\,b]$. So the variant expression is the length of this array, i.e.:

$$b - a + 1$$

Binary Search

The invariant I is:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ A[1\,..\,a-1] < x \ \wedge \ A[b+1\,..\,N] > x)$$

If $r \neq 0$ then the required postcondition is satisfied.

Otherwise, if $b < a$ then the sub-array is empty, and since $r = 0$, the postcondition is satisfied.

Otherwise, we need to reduce $b - a + 1$ and ensure that I is still satisfied.

Pick a value m such that $a \leqslant m \leqslant b$. There are three cases:

Binary Search

The invariant I is:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ A[1\,..\,a-1] < x \ \wedge \ A[b+1\,..\,N] > x)$$

If $r \neq 0$ then the required postcondition is satisfied.

Otherwise, if $b < a$ then the sub-array is empty, and since $r = 0$, the postcondition is satisfied.

Otherwise, we need to reduce $b - a + 1$ and ensure that I is still satisfied.

Pick a value m such that $a \leqslant m \leqslant b$. There are three cases:

1. $A[m] = x$ Setting $r := m$ will satisfy I

Binary Search

The invariant I is:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ A[1\,..\,a-1] < x \ \wedge \ A[b+1\,..\,N] > x)$$

If $r \neq 0$ then the required postcondition is satisfied.

Otherwise, if $b < a$ then the sub-array is empty, and since $r = 0$, the postcondition is satisfied.

Otherwise, we need to reduce $b - a + 1$ and ensure that I is still satisfied.

Pick a value m such that $a \leqslant m \leqslant b$. There are three cases:

1. $A[m] = x$ Setting $r := m$ will satisfy I
2. $A[m] < x$ Then $A[1\,..\,m] < x$, so setting $a := m + 1$ preserves I

Binary Search

The invariant I is:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ A[1\,..\,a-1] < x \ \wedge \ A[b+1\,..\,N] > x)$$

If $r \neq 0$ then the required postcondition is satisfied.

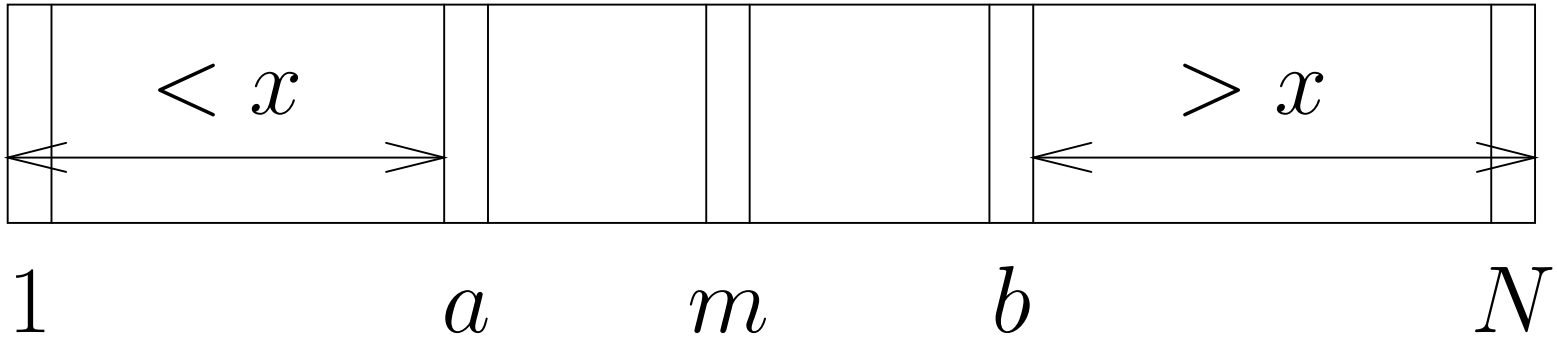
Otherwise, if $b < a$ then the sub-array is empty, and since $r = 0$, the postcondition is satisfied.

Otherwise, we need to reduce $b - a + 1$ and ensure that I is still satisfied.

Pick a value m such that $a \leqslant m \leqslant b$. There are three cases:

1. $A[m] = x$ Setting $r := m$ will satisfy I
2. $A[m] < x$ Then $A[1\,..\,m] < x$, so setting $a := m + 1$ preserves I
3. $A[m] > x$ Then $A[m\,..\,N] > x$, so setting $b := m - 1$ preserves I

Binary Search Invariant



1. If $A[m] = x$ then set $r := m$
2. If $A[m] < x$ then $A[1\,..\,m] < x$, so set $a := m + 1$
3. If $A[m] > x$ then $A[m\,..\,N] > x$, so set $b := m - 1$

Binary Search

Putting these facts together:

$a := 1;\ b := N;\ r := 0;$

while $r = 0 \ \wedge \ b \geqslant a$ **do**

$m :=$ (some value in the range $a \,..\, b$);

if $A[m] = x$ **then** $r := m$

elsif $A[m] < x$ **then** $a := m + 1$

else $b := m - 1$ **fi od**

For efficiency, the assignment to m should pick the middle element, i.e. $\lfloor (a+b)/2 \rfloor$. To avoid numeric overflow this can be calculated as:

$$a + \lfloor (b-a)/2 \rfloor$$

Notice that the correctness of the algorithm does not depend in any way on the choice of m.

Binary Search

$a := 1;\ b := N;\ r := 0;$
while $r = 0 \ \wedge \ b \geqslant a$ **do**
$m := a + \lfloor (b - a)/2 \rfloor;$
if $A[m] = x$ **then** $r := m$
elsif $A[m] < x$ **then** $a := m + 1$
else $b := m - 1$ **fi od**

On termination of the **while** loop we have:

$$I \ \wedge \ \neg(r = 0 \ \wedge \ b \geqslant a)$$

Binary Search

$$((r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ A[1\,..\,a-1] < x \ \wedge \ A[b+1\,..\,N] > x))$$
$$\wedge \ (r \neq 0 \ \vee \ b < a)$$

Binary Search

$$((r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ A[1\,..\,a-1] < x \ \wedge \ A[b+1\,..\,N] > x))$$
$$\wedge \ (r \neq 0 \ \vee \ b < a)$$

which is:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ b < a \ \wedge \ A[1\,..\,a{-}1] < x \ \wedge \ A[b{+}1\,..\,N] > x)$$

Binary Search

$$((r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ A[1\,..\,a-1] < x \ \wedge \ A[b+1\,..\,N] > x))$$
$$\wedge \ (r \neq 0 \ \vee \ b < a)$$

which is:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ b < a \ \wedge \ A[1\,..\,a{-}1] < x \ \wedge \ A[b{+}1\,..\,N] > x)$$

which implies:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ b < a \ \wedge \ A[1\,..\,a{-}1] \neq x \ \wedge \ A[b{+}1\,..\,N] \neq x)$$

Binary Search

$$((r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ A[1\,..\,a-1] < x \ \wedge \ A[b+1\,..\,N] > x))$$
$$\wedge \ (r \neq 0 \ \vee \ b < a)$$

which is:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ b < a \ \wedge \ A[1\,..\,a{-}1] < x \ \wedge \ A[b{+}1\,..\,N] > x)$$

which implies:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ b < a \ \wedge \ A[1\,..\,a{-}1] \neq x \ \wedge \ A[b{+}1\,..\,N] \neq x)$$

which implies:

$$(r \neq 0 \ \wedge \ A[r] = x) \ \vee \ (r = 0 \ \wedge \ A[1\,..\,N] \neq x)$$

as required.

Specification Methods

The next few slides will describe various methods for specifying programs and proving that an implementation of a specification is correct.

Algebraic Specification

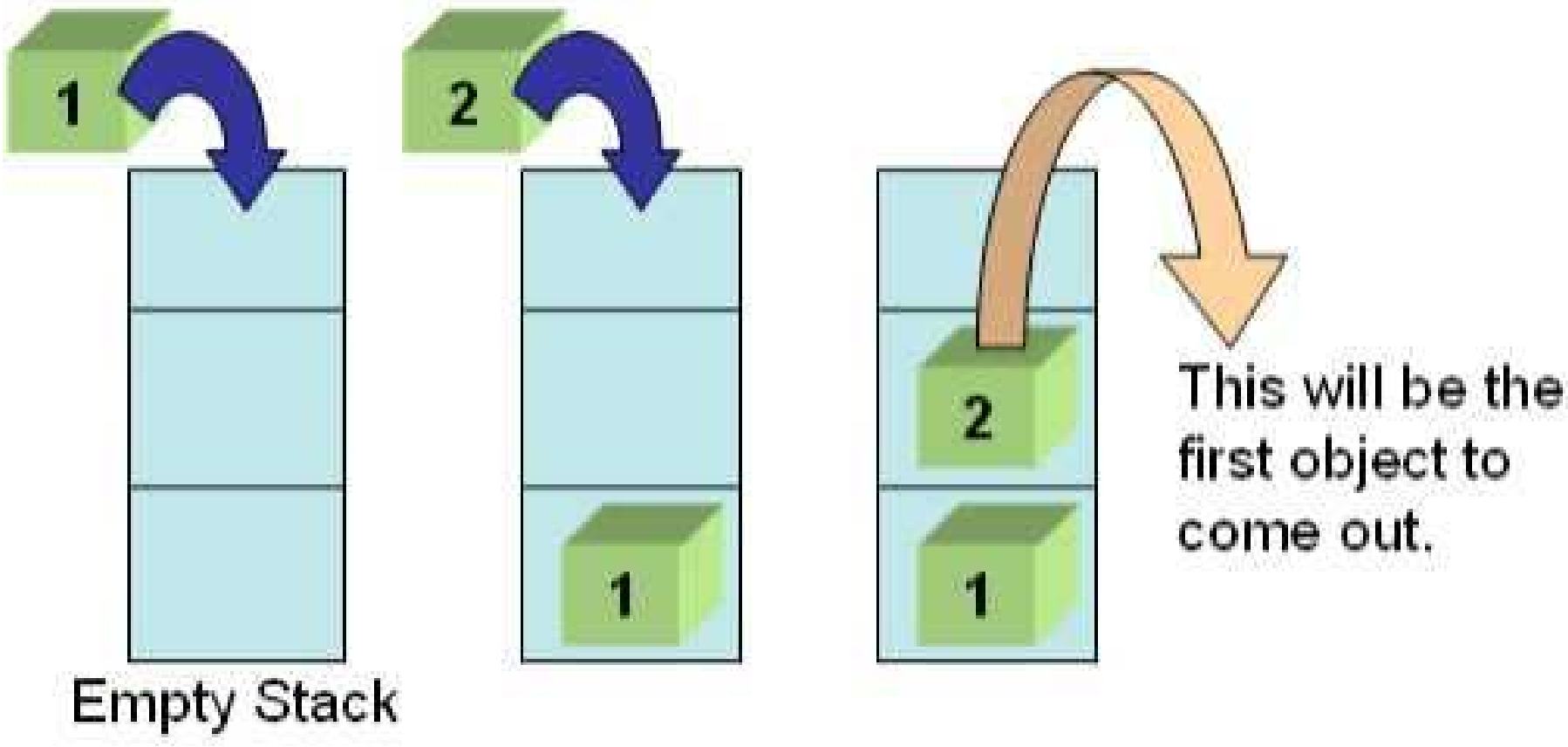
In this approach, an implicit definition of operations is given by relating the behaviour of different operations without defining the meanings of the actual states.

An **algebraic**, or **axiomatic** specification consists of:

1. A name
2. An informal description of the data type
3. A set of abstract data elements
4. A set of operations on the abstract data elements
5. A set of axioms which define the meaning of the operations

Specification of a Stack

Informal Description:



Specification of a Stack

The operations for data type Stack, operating on base set E are:

- init: $\rightarrow$ stack (produces a new empty stack)
- top: stack $\rightarrow E \cup \{\text{error}\}$ (returns the top element)
- push: stack, $E \rightarrow$ stack (adds an element to the stack)
- pop: stack $\rightarrow$ stack $\cup \{\text{error}\}$ (removes the top element)
- is_empty?: stack $\rightarrow \{\textbf{true}, \textbf{false}\}$ (test if the stack is empty)

where error is a special element denoting an error result.

Specification of a Stack

The operations for data type Stack, operating on base set E are:

- init: $\rightarrow$ stack (produces a new empty stack)
- top: stack $\rightarrow E \cup \{\text{error}\}$ (returns the top element)
- push: stack, $E \rightarrow$ stack (adds an element to the stack)
- pop: stack $\rightarrow$ stack $\cup \{\text{error}\}$ (removes the top element)
- is_empty?: stack $\rightarrow \{\textbf{true}, \textbf{false}\}$ (test if the stack is empty)

where error is a special element denoting an error result.

Note that we have only defined the **type** of each operation, and (so far) said nothing about its **behaviour**.

Specification of a Stack

The axioms for Stack are:

1. is_empty?(init) = **true**
2. is_empty?(push(s, x)) = **false**
3. pop(init) = error
4. pop(push(s, x)) = s
5. top(init) = error
6. top(push(s, x)) = x

For example:

$$\text{push}(\text{pop}(\text{push}(\text{push}(\text{init}, 4), 5)), 7)$$

is the same stack as:

$$\text{push}(\text{push}(\text{init}, 4), 7)$$

Algebraic Specifications: Advantages

The *user* of a data type has to show that the data type axioms are sufficient to demonstrate the correctness of each program which makes used of the data type.

The *implementer* of a data type simply has to provide something for which all the axioms are valid.

Algebraic Specifications: Advantages

The *user* of a data type has to show that the data type axioms are sufficient to demonstrate the correctness of each program which makes used of the data type.

The *implementer* of a data type simply has to provide something for which all the axioms are valid.

The user doesn't need to know anything about the implementation.

Algebraic Specifications: Advantages

The *user* of a data type has to show that the data type axioms are sufficient to demonstrate the correctness of each program which makes used of the data type.

The *implementer* of a data type simply has to provide something for which all the axioms are valid.

The user doesn't need to know anything about the implementation.

The implementer doesn't need to know anything about how the data type is actually used.

Stack Implementer

The implementer of a stack needs to do the following:

1. Provide a set of functions: init, top, push, pop and is_empty? which accept parameters of the appropriate types, and produce a result of the appropriate type
2. Prove that all the axioms are universally valid for the implementations of the functions.

Stack Implementer

The implementer of a stack needs to do the following:

1. Provide a set of functions: init, top, push, pop and is_empty? which accept parameters of the appropriate types, and produce a result of the appropriate type
2. Prove that all the axioms are universally valid for the implementations of the functions.

For example, the implementer might decide to implement a stack as a **linked list**.

Stack Implementation

Implementing a stack as a linked list of nodes:

1. A **node** is a struct containing two pointers: `next` and `data`

Stack Implementation

Implementing a stack as a linked list of nodes:

1. A **node** is a struct containing two pointers: `next` and `data`
2. init simply returns a null pointer;

Stack Implementation

Implementing a stack as a linked list of nodes:

1. A **node** is a struct containing two pointers: `next` and `data`
2. init simply returns a null pointer;
3. is_empty? tests if the given pointer is null;

Stack Implementation

Implementing a stack as a linked list of nodes:

1. A **node** is a struct containing two pointers: `next` and `data`
2. init simply returns a null pointer;
3. is_empty? tests if the given pointer is null;
4. The axiom is_empty?(init) is therefore easy to prove!

Stack Implementation

Implementing a stack as a linked list of nodes:

1. A **node** is a struct containing two pointers: `next` and `data`
2. init simply returns a null pointer;
3. is_empty? tests if the given pointer is null;
4. The axiom is_empty?(init) is therefore easy to prove!
5. push(s, x) allocates a new node, sets the `next` pointer to s and the `data` pointer to x and returns a pointer to the new node

Stack Implementation

Implementing a stack as a linked list of nodes:

1. A **node** is a struct containing two pointers: `next` and `data`
2. init simply returns a null pointer;
3. is_empty? tests if the given pointer is null;
4. The axiom is_empty?(init) is therefore easy to prove!
5. push(s, x) allocates a new node, sets the `next` pointer to s and the `data` pointer to x and returns a pointer to the new node
6. Therefore, is_empty?(push(s, x)) is false (unless the memory allocation failed!)

Stack Implementation

Implementing a stack as a linked list of nodes:

1. A **node** is a struct containing two pointers: `next` and `data`
2. init simply returns a null pointer;
3. is_empty? tests if the given pointer is null;
4. The axiom is_empty?(init) is therefore easy to prove!
5. push(s, x) allocates a new node, sets the `next` pointer to s and the `data` pointer to x and returns a pointer to the new node
6. Therefore, is_empty?(push(s, x)) is false (unless the memory allocation failed!)
7. . . . and so on.

Stack User

The stack user might have a harder job than the implementer.

He has to prove that his program is correct for *any* implementation of a stack. In other words: he has to prove that his program is correct using only the axioms of a stack.

Stack User

The stack user might have a harder job than the implementer.

He has to prove that his program is correct for *any* implementation of a stack. In other words: he has to prove that his program is correct using only the axioms of a stack.

For example:

$s := \mathsf{push}(s, x);$
... Do some processing which includes
... pushing and popping items on and off s
$x := \mathsf{top}(s);\ s := \mathsf{pop}(s);$
x *should now contain the value that was pushed*

Stack User

The stack user might have a harder job than the implementer.

He has to prove that his program is correct for *any* implementation of a stack. In other words: he has to prove that his program is correct using only the axioms of a stack.

For example:

$s := \mathsf{push}(s, x)$;
... Do some processing which includes
... pushing and popping items on and off s
$x := \mathsf{top}(s)$; $s := \mathsf{pop}(s)$;
x *should now contain the value that was pushed*

What does the user need to do to prove that this program is correct?

Completeness and Correctness

On first meeting algebraic specifications, an immediate question arises: how do we know that the given axioms provide a precise definition of the behaviour of the operations for the data type? Loosely speaking, completeness is concerned with the problem of whether there are enough independent axioms to adequately describe the behaviour of the operations of the abstract data type. The set of axioms which define the semantics of an abstract data type should be complete in the sense that :

1. operations must be defined which allow the construction of all possible legal instances (all the values we want) of the abstract data type
2. the results for all legal applications and compositions of the operations must be defined.

Consistency

The set of axioms must also be *consistent*: in other words, it must not be possible to derive a contradiction from the set of axioms.

Consistency

The set of axioms must also be *consistent*: in other words, it must not be possible to derive a contradiction from the set of axioms.

For example, suppose we decide to add another axiom to the list:

$$\mathsf{push}(s, x) = s$$

Consistency

The set of axioms must also be *consistent*: in other words, it must not be possible to derive a contradiction from the set of axioms.

For example, suppose we decide to add another axiom to the list:

$$\mathsf{push}(s, x) = s$$

Now, by axiom (1):

$$\mathsf{is\_empty?}(\mathsf{init}) = \textbf{true}$$

Consistency

The set of axioms must also be *consistent*: in other words, it must not be possible to derive a contradiction from the set of axioms.

For example, suppose we decide to add another axiom to the list:

$$\mathsf{push}(s, x) = s$$

Now, by axiom (1):

$$\mathsf{is\_empty?}(\mathsf{init}) = \textbf{true}$$

By the new axiom, $\mathsf{init} = \mathsf{push}(\mathsf{init}, x)$, so:

$$\mathsf{is\_empty?}(\mathsf{push}(\mathsf{init}, x)) = \textbf{true}$$

Consistency

The set of axioms must also be *consistent*: in other words, it must not be possible to derive a contradiction from the set of axioms.

For example, suppose we decide to add another axiom to the list:

$$\mathsf{push}(s, x) = s$$

Now, by axiom (1):

$$\mathsf{is\_empty?}(\mathsf{init}) = \textbf{true}$$

By the new axiom, $\mathsf{init} = \mathsf{push}(\mathsf{init}, x)$, so:

$$\mathsf{is\_empty?}(\mathsf{push}(\mathsf{init}, x)) = \textbf{true}$$

But axiom (2) says, for all s, including $s = \mathsf{init}$:

$$\mathsf{is\_empty?}(\mathsf{push}(s, x)) = \textbf{false}$$

Algebraic Specs: Disadvantages

- It may be necessary to provide several infinite sets of axioms in order to completely specify a simple data type!

Algebraic Specs: Disadvantages

- It may be necessary to provide several infinite sets of axioms in order to completely specify a simple data type!
- It is sometimes difficult to prove that algebraic specifications are mathematically complete and consistent.

Algebraic Specs: Disadvantages

- It may be necessary to provide several infinite sets of axioms in order to completely specify a simple data type!
- It is sometimes difficult to prove that algebraic specifications are mathematically complete and consistent.
- The usual way to prove consistency of a set of axioms is to provide a *model*.

Algebraic Specs: Disadvantages

- It may be necessary to provide several infinite sets of axioms in order to completely specify a simple data type!
- It is sometimes difficult to prove that algebraic specifications are mathematically complete and consistent.
- The usual way to prove consistency of a set of axioms is to provide a *model*.

Why not just use the model itself as the specification?

Model-based Approach

A system is modelled by explicitly defining the states and operations that transform the system from one state to another.

The model is defined in terms of well-known mathematical objects: sets, functions, relations, sequences and so on. Properties of these objects can be used in proofs.

A Model for a Stack

Define a stack as a sequence of elements.

Sequences are well-understood mathematical objects with familiar properties.

Let s be the sequence $\langle s_1, s_2, \ldots, s_n \rangle$.

Define:

- $s[i] = s_i$
- $s[i..j] = \langle s_i, s_{i+1}, \ldots, s_j \rangle$
- $\ell(s) = n$
- $s[j..] = s[j..\ell(s)]$
- $s +\!\!+ t = \langle s_1, \ldots, s_n, t_1, \ldots, t_m \rangle$

A Model for a Stack

Define a Stack in terms of a sequence:

$$
\begin{array}{rcl}
\text{init} & =_{\text{DF}} & \langle\rangle \quad \text{the empty sequence} \\
\text{top}(\langle\rangle) & =_{\text{DF}} & \text{error} \\
\text{top}(s) & =_{\text{DF}} & s[1] \quad \text{if } s \neq \langle\rangle \\
\text{push}(s, x) & =_{\text{DF}} & \langle x \rangle \mathbin{+\!\!+} s \\
\text{pop}(\langle\rangle) & =_{\text{DF}} & \text{error} \\
\text{pop}(s) & =_{\text{DF}} & s[2..] \quad \text{if } s \neq \langle\rangle \\
\text{is\_empty?}(\langle\rangle) & =_{\text{DF}} & \textbf{true} \\
\text{is\_empty?}(s) & =_{\text{DF}} & \textbf{false} \quad \text{if } s \neq \langle\rangle
\end{array}
$$

This is a **model based** specification.

Using the Model

The *user* of a data type replaces calls to operations by the corresponding abstract code. (For example, $\mathsf{push}(s, x)$ is replaced by $\langle x \rangle +\!\!+ s$). Then the user proves that the resulting abstract program is correct.

The *implementer* of a data type has to provide some concrete code which is a valid data refinement of the abstract code. (For example, a linked list implementation of a stack). An *abstraction function* maps from the concrete data elements to the corresponding abstract data elements.

Using the Model

The *user* of a data type replaces calls to operations by the corresponding abstract code. (For example, $\mathsf{push}(s, x)$ is replaced by $\langle x \rangle +\!\!+ s$). Then the user proves that the resulting abstract program is correct.

The *implementer* of a data type has to provide some concrete code which is a valid data refinement of the abstract code. (For example, a linked list implementation of a stack). An *abstraction function* maps from the concrete data elements to the corresponding abstract data elements.

The user doesn't need to know anything about the implementation.

Using the Model

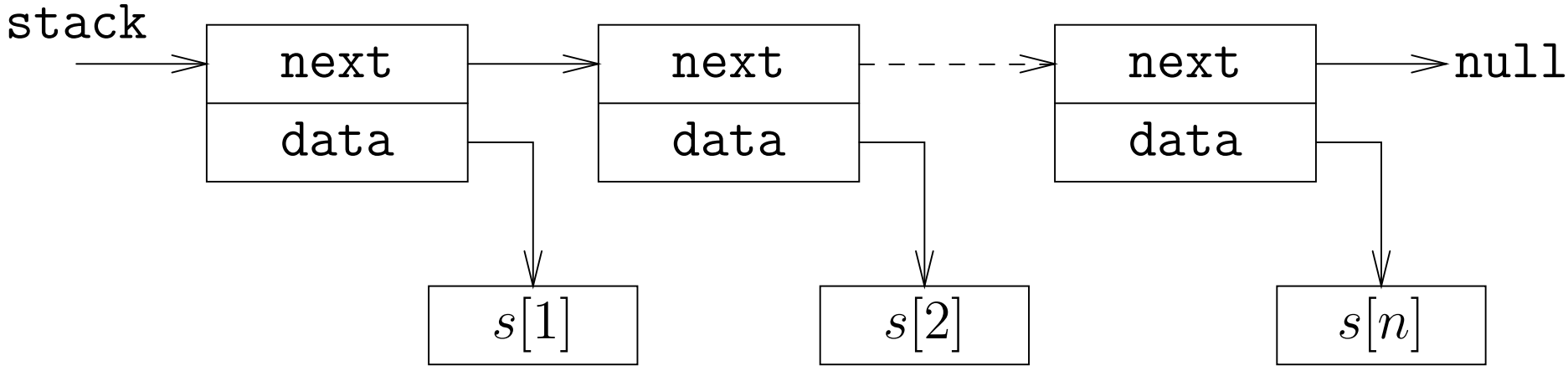
The *user* of a data type replaces calls to operations by the corresponding abstract code. (For example, $\mathsf{push}(s, x)$ is replaced by $\langle x \rangle +\!\!+ s$). Then the user proves that the resulting abstract program is correct.

The *implementer* of a data type has to provide some concrete code which is a valid data refinement of the abstract code. (For example, a linked list implementation of a stack). An *abstraction function* maps from the concrete data elements to the corresponding abstract data elements.

The user doesn't need to know anything about the implementation.

The implementer doesn't need to know anything about how the data type is actually used.

Stack Implementation



Stack Implementer

Our stack implementer with the linked list now has to provide an abstraction function which maps a linked list to a sequence. For example:

$$\mathsf{abs}(\mathtt{s}) =_{\mathsf{DF}} \begin{cases} \langle\rangle & \text{if } \mathtt{s} \text{ is a null pointer} \\ \langle \mathtt{s->data} \rangle \mathbin{+\!\!+} \mathsf{abs}(\mathtt{s->next}) & \text{otherwise} \end{cases}$$

He then has to prove that for each operation, the abstract value of the output of the operation is the correct model operation applied to the abstract values of each parameter.

For example:

$$\mathsf{abs}(\mathsf{push}(\mathtt{s}, \mathtt{x})) = \langle \mathtt{x} \rangle \mathbin{+\!\!+} \mathsf{abs}(\mathtt{s})$$

Stack Implementer

To prove:

$$\mathsf{abs}(\mathsf{push}(\mathtt{s}, \mathtt{x})) = \langle \mathtt{x} \rangle +\!\!+ \mathsf{abs}(\mathtt{s})$$

Note that $\mathsf{push}(\mathtt{s}, \mathtt{x})$ allocates a node, say `s1`, and sets:

`s1->data` $:=$ `x`;

`s1->next` $:=$ `s`

and then returns `s1`.

So, by the definition of abs:

$$\mathsf{abs}(\mathtt{s1}) \quad = \quad \langle \mathtt{s1\text{->}data} \rangle +\!\!+ \mathsf{abs}(\mathtt{s1\text{->}next})$$

since `s1` is not a null pointer

$$= \quad \langle \mathtt{x} \rangle +\!\!+ \mathsf{abs}(\mathtt{s})$$

as required.

Stack User

The stack user's program becomes:

$s := \langle x \rangle +\!\!+ s;$
... Do some processing which includes
... pushing and popping items on and off s
$x := s[1];\ s := s[2\,..];\ ;$
x *should now contain the value that was pushed*

The user can directly use concepts such as the length of the sequence s in his correctness proof. If the "do some processing" section does the following:

1. Preserves the invariant $\ell(s) \geqslant L$ where L is the initial length of the stack; and
2. Ensures that $\ell(s) = L$ at the end of the section

then the correctness proof is simple.

Logic-based Approach

In this approach logic is used to describe the system's desired properties, including the low-level specification, temporal, and probabilistic behaviours. The validity of these properties is achieved using the associated axiom system of the logic. In some cases, a subset of the logic can be executed (e.g., the Tempura system). The executable specification can then be used for simulation and rapid prototyping purposes.

Examples include: Hoare Logic, Dijkstra Weakest Preconditions Calculus, Temporal logic etc.

Logic-based Approach

The logic can be augmented with some concrete programming constructs to obtain what is known as wide-spectrum formalism. The development of systems in this case is achieved by a set of correctness-preserving refinement steps.

Examples include: TAM, the refinement calculus.

FermaT Transformation System

FermaT is both:

- model-based (via denotational semantics) and
- logic-based (via weakest preconditions in infinitary first order logic)

The two foundations for WSL are proved to be equivalent.

This means that both methods (semantics and weakest preconditions) can be applied to prove the correctness of transformations.

Both methods are used in practice.

Net Based Specifications

Graphical notations are popular notations for specifying systems as they are easier to comprehend and, hence, more accessible to non-specialists. This approach uses graphical languages with formal semantics, which brings particular advantages in system development and reengineering.

Petri Net theory was one of the first formalisms to deal with concurrency, nondeterminism and causal connections between events. According to Milner, it was the first unified theory, with levels of abstraction, in which to describe and analyse all aspects of a computer in the context of its environment.

Petri nets provide a graphic representation with formal semantics of system behaviour. A large number of varieties of Petri Net Theory have been proposed. Generally, petri nets can be classified into ordinary (classic) petri nets and timed petri nets.

Net Based Specifications

A Petri net consists of **places**, **transitions**, and **directed arcs**. Arcs run between places and transitions—not between places and places or transitions and transitions. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition.

Places may contain any number of **tokens**. A distribution of tokens over the places of a net is called a marking. Transitions act on input tokens by a process known as firing. A transition is enabled if it can fire, i.e., there are tokens in every input place. When a transition fires, it consumes the tokens from its input places, performs some processing task, and places a specified number of tokens into each of its output places. It does this atomically, i.e., in one non-interruptible step.

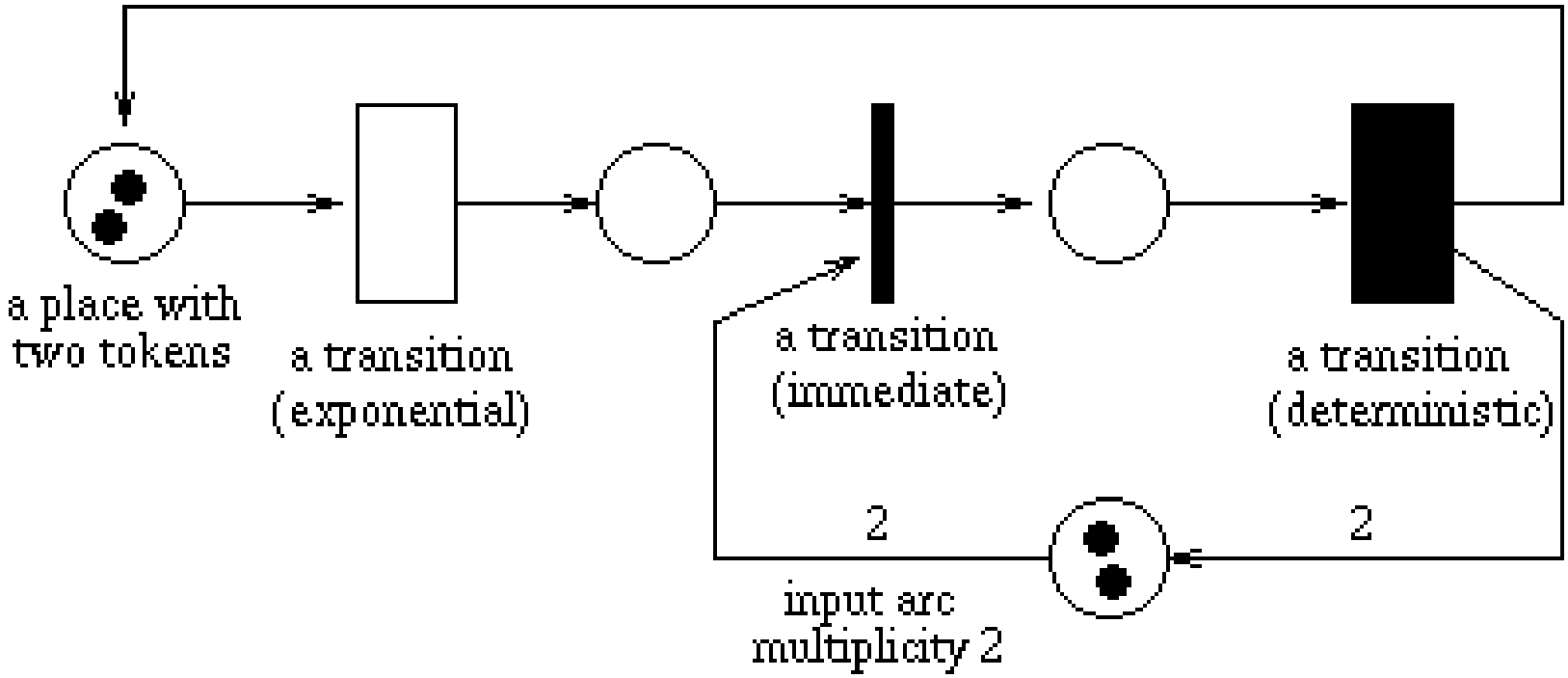
Net Based Specifications

Execution of Petri nets is nondeterministic. This means two things:

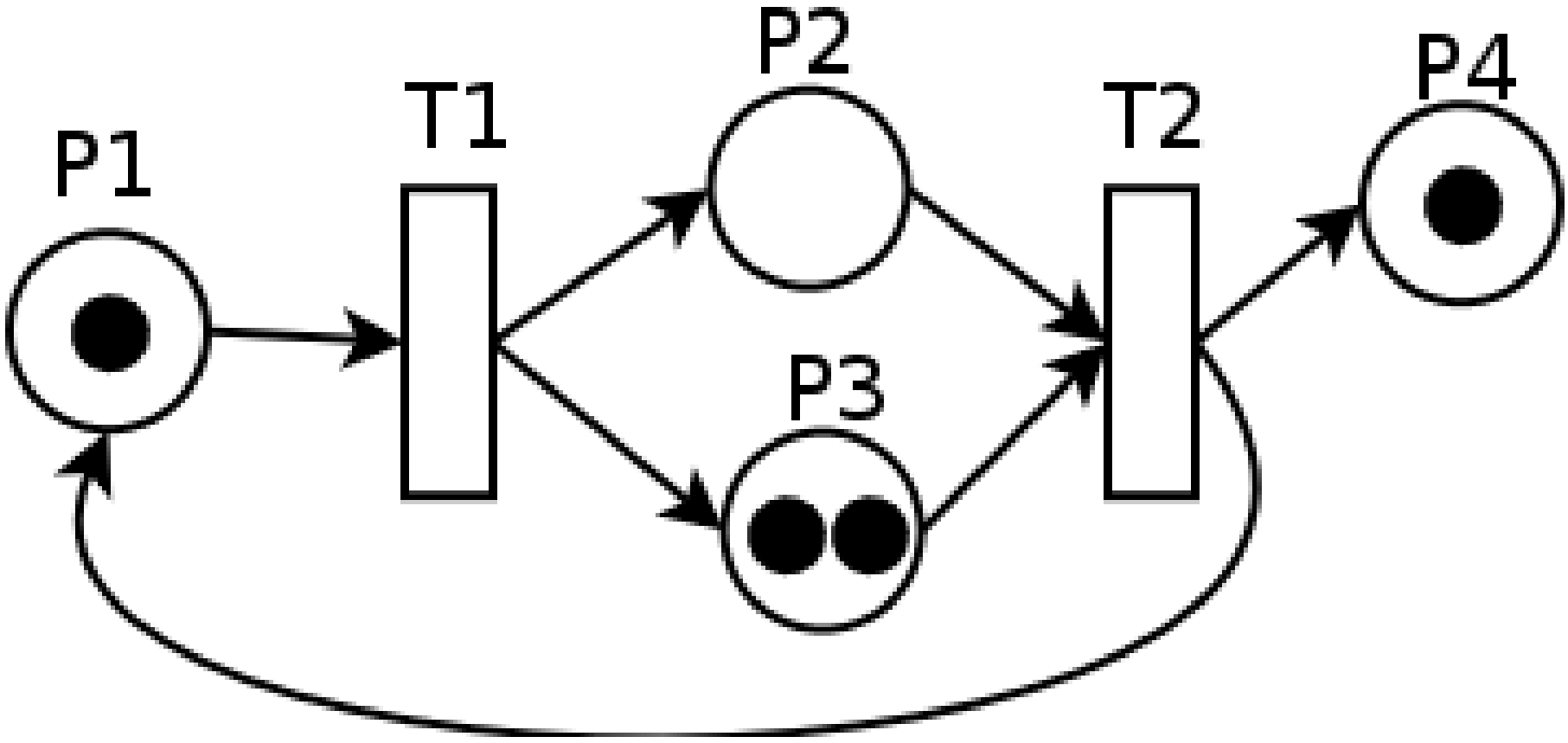
1. Multiple transitions can be enabled at the same time, any one of which can fire
2. None are required to fire — they fire at will, between time 0 and infinity, or not at all (i.e. it is totally possible that nothing fires at all).

Since firing is nondeterministic, Petri nets are well suited for modelling the concurrent behaviour of distributed systems.

Petri Nets: Example



Petri Nets: Example



Statecharts

Statecharts provides an abstraction mechanism based on finite state machine. It represents an improved version of the structured methods. A graphic tool called "Statemate" exists to implement the formalism.

In statecharts, conventional finite state machines are extended by AND/OR decomposition of states, interlevel transitions, and an implicit intercomposition broadcast communication. Statecharts denote the composition of state machines into super-machines which may execute concurrently. The state machines contain transitions which are marked by enabling and output events. It is assumed that events are instantaneous, and a global discrete clock is used to trigger sets of concurrent events. Statecharts are hierarchical, and may be composed into complex charts.

Statecharts

Statecharts support the typical structural top-down system development methods. They do not fit in with the procedures of reverse engineering, which involve abstraction of specifications from source code. Real time is incorporated in Statecharts by having an implicit clock, allowing transitions to be triggered by timeouts relative to this clock, and by requiring that if a transition can be taken, then it must be taken immediately.

Process Algebra

The term "process algebra" was coined in 1982 by Bergstra & Klop. A process algebra is a structure in the sense of universal algebra that satisfied a particular set of axioms. Since 1984 the phrase "process algebra" has also been used to denote an area of science: the algebraic approach to the study of concurrent processes.

In this approach, an explicit representation of concurrent processes is allowed. System behaviour is represented by constraints on all allowable observable communications between processes.

The main algebraic approaches to concurrency are

- CCS, Milner's Calculus of Communicating Systems
- CSP, Hoare's Communicating Sequential Processes
- ACP, Bergstra & Klop's Algebra of Communicating Processes

Process Algebra

A process algebra starts with a set of names (or channels) whose purpose is to provide means of communication, together with a means to form new processes from old. The basic operators, always present in some form or other include:

- parallel composition of processes
- specifying which channels to use for sending and receiving data
- sequentializing interactions
- hiding interaction channels
- recursion or process replication

System behaviour is represented by constraints on all allowable observed communications between processes.

Syntax and Semantics of WSL

WSL is a Wide Spectrum Language which forms the basis for the WSL theory of program transformations and the FermaT program transformation system.

Program States

A program starts executing in some *state*. A state is a collection of variables each of which has a value. The collection of variables is called the *state space*. The state space may change during the execution of the program, with variables being added or removed.

Program States

A program starts executing in some *state*. A state is a collection of variables each of which has a value. The collection of variables is called the *state space*. The state space may change during the execution of the program, with variables being added or removed.

The state space is a finite non-empty set of variables. Each variable in the state space has a value, take from some set $\mathcal{H}$ of values.

Program States

A program starts executing in some *state*. A state is a collection of variables each of which has a value. The collection of variables is called the *state space*. The state space may change during the execution of the program, with variables being added or removed.

The state space is a finite non-empty set of variables. Each variable in the state space has a value, take from some set $\mathcal{H}$ of values.

So a state can be modelled as a function from the state space to the set of values. This function returns the value of each variable in the state space.

Program States

A program starts executing in some *state*. A state is a collection of variables each of which has a value. The collection of variables is called the *state space*. The state space may change during the execution of the program, with variables being added or removed.

The state space is a finite non-empty set of variables. Each variable in the state space has a value, take from some set $\mathcal{H}$ of values.

So a state can be modelled as a function from the state space to the set of values. This function returns the value of each variable in the state space.

For example, let s be the state in which x has the value 1 and y has the value 2:

$$s = \{x \mapsto 1, y \mapsto 2\}$$

State Transformation

Start a program executing in some state: it will either run forever, or (eventually) terminate in some state.

State Transformation

Start a program executing in some state: it will either run forever, or (eventually) terminate in some state.

"Running forever" is represented by the special state $\perp$.
A *proper state* is any state other than $\perp$.

State Transformation

Start a program executing in some state: it will either run forever, or (eventually) terminate in some state.

"Running forever" is represented by the special state $\perp$.
A *proper state* is any state other than $\perp$.

Programs may be *non-deterministic*: for the same initial state, there may be two or more possible final states.

State Transformation

Start a program executing in some state: it will either run forever, or (eventually) terminate in some state.

"Running forever" is represented by the special state $\perp$.
A *proper state* is any state other than $\perp$.

Programs may be *non-deterministic*: for the same initial state, there may be two or more possible final states.

So we will represent the behaviour of a program by a function which maps from each initial state to the set of possible final states.

State Transformation

Start a program executing in some state: it will either run forever, or (eventually) terminate in some state.

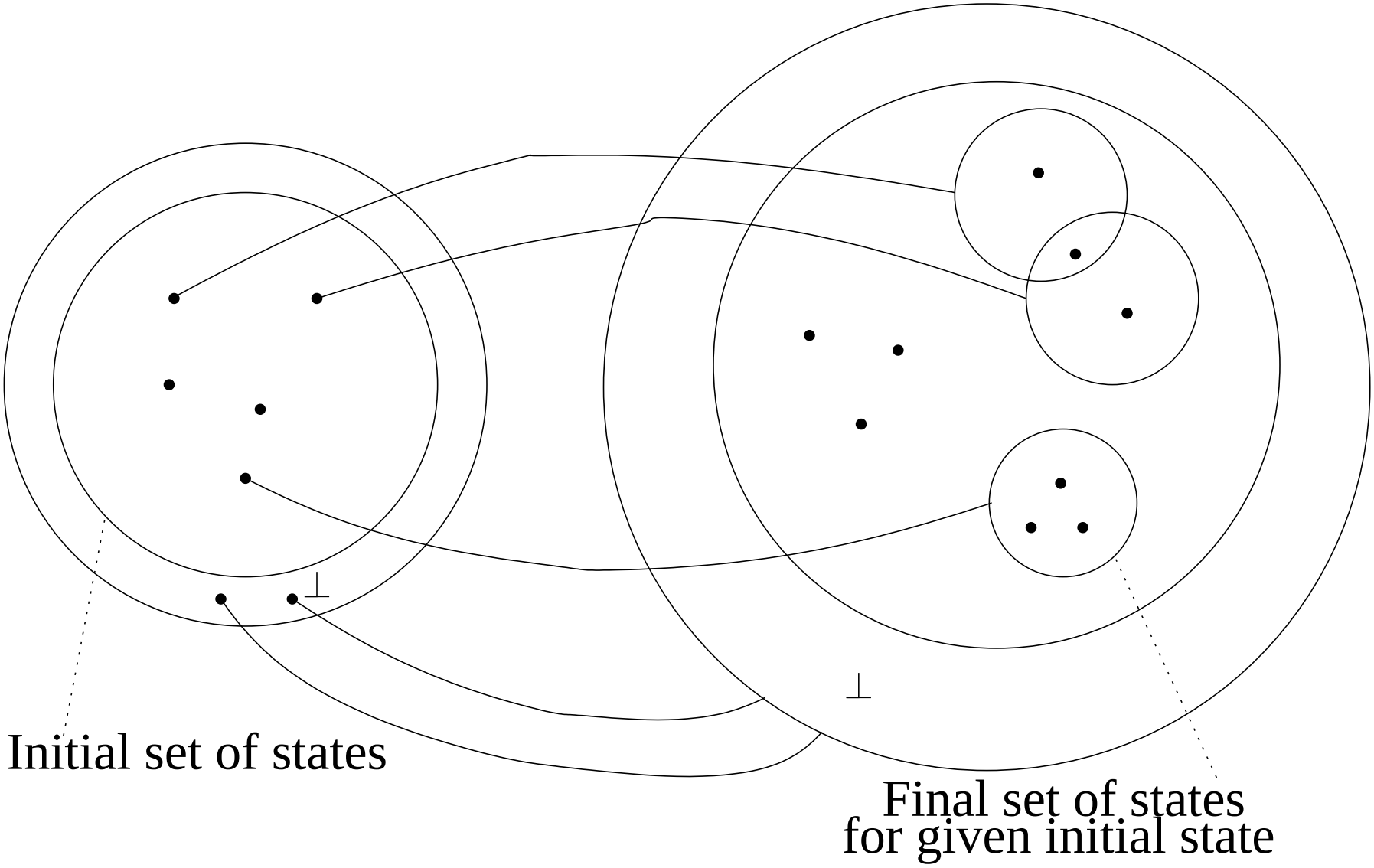
"Running forever" is represented by the special state $\perp$.
A *proper state* is any state other than $\perp$.

Programs may be *non-deterministic*: for the same initial state, there may be two or more possible final states.

So we will represent the behaviour of a program by a function which maps from each initial state to the set of possible final states.

This function is called a *state transformation*.

The Semantics of WSL



State Transformation

Starting a program in the state "running forever" state $\perp$ means that some previous program is running forever: in other words, this program never even gets started!

So, if the initial state is $\perp$ then the final state can only be $\perp$.

A program cannot terminate if it never gets started!

If the set of final states includes $\perp$ then by definition it includes all other states.

State Transformation

Two simple programs

State Transformation

Two simple programs

skip

terminates immediately in the state in which it was started.

For each proper initial state s, the set of final states is $\{s\}$.

State Transformation

Two simple programs

skip

terminates immediately in the state in which it was started.

For each proper initial state s, the set of final states is $\{s\}$.

abort

Does not terminate on any initial state. The set of final states always includes $\perp$ plus every other state.

An Example

Suppose that the set $\mathcal{H}$ contains only two values: 0 and 1, and the state space contains only two variables x and y.

An Example

Suppose that the set $\mathcal{H}$ contains only two values: 0 and 1, and the state space contains only two variables x and y.

The five possible states are:

1. $s_{00} = \{x \mapsto 0, y \mapsto 0\}$
2. $s_{01} = \{x \mapsto 0, y \mapsto 1\}$
3. $s_{10} = \{x \mapsto 1, y \mapsto 0\}$
4. $s_{11} = \{x \mapsto 1, y \mapsto 1\}$
5. $\bot$

An Example

The semantic function for the program $y := x$ maps:

An Example

The semantic function for the program $y := x$ maps:

$$
\begin{aligned}
s_{00} &\mapsto \{s_{00}\} \\
s_{01} &\mapsto \{s_{00}\} \\
s_{10} &\mapsto \{s_{11}\} \\
s_{11} &\mapsto \{s_{11}\} \\
\bot &\mapsto \{\bot, s_{00}, s_{01}, s_{10}, s_{11}\}
\end{aligned}
$$

An Example

The semantic function for the program **add**(y) maps:

An Example

The semantic function for the program **add**(y) maps:

$$
\begin{aligned}
s_{00} &\mapsto \{s_{00}, s_{01}\} \\
s_{01} &\mapsto \{s_{00}, s_{01}\} \\
s_{10} &\mapsto \{s_{10}, s_{11}\} \\
s_{11} &\mapsto \{s_{10}, s_{11}\} \\
\bot &\mapsto \{\bot, s_{00}, s_{01}, s_{10}, s_{11}\}
\end{aligned}
$$

The WSL Kernel Language

"The quarks of programming"

The primitive kernel statements are constructed from formulae and lists of variables.

The WSL Kernel Language

"The quarks of programming"

The primitive kernel statements are constructed from formulae and lists of variables.

Let **P** and **Q** be any formulae and **x** and **y** be any lists of variables:

- **Assertion:** $\{\mathbf{P}\}$ Does nothing if **P** is true, aborts if **P** is false;

The WSL Kernel Language

"The quarks of programming"

The primitive kernel statements are constructed from formulae and lists of variables.

Let **P** and **Q** be any formulae and **x** and **y** be any lists of variables:

- **Assertion:** $\{\mathbf{P}\}$ Does nothing if **P** is true, aborts if **P** is false;
- **Guard:** $[\mathbf{Q}]$ Ensures that **Q** is true by restricting previous nondeterminism;

The WSL Kernel Language

"The quarks of programming"

The primitive kernel statements are constructed from formulae and lists of variables.

Let **P** and **Q** be any formulae and **x** and **y** be any lists of variables:

- **Assertion:** $\{\mathbf{P}\}$ Does nothing if **P** is true, aborts if **P** is false;
- **Guard:** $[\mathbf{Q}]$ Ensures that **Q** is true by restricting previous nondeterminism;
- **Add variables:** $\mathbf{add}(\mathbf{x})$ adds the variables in **x** to the state space and assigns arbitrary values to them;

The WSL Kernel Language

"The quarks of programming"

The primitive kernel statements are constructed from formulae and lists of variables.

Let **P** and **Q** be any formulae and **x** and **y** be any lists of variables:

- **Assertion:** $\{\mathbf{P}\}$ Does nothing if **P** is true, aborts if **P** is false;
- **Guard:** $[\mathbf{Q}]$ Ensures that **Q** is true by restricting previous nondeterminism;
- **Add variables:** **add**$(\mathbf{x})$ adds the variables in **x** to the state space and assigns arbitrary values to them;
- **Remove variables:** **remove**$(\mathbf{y})$ removes the variables in **y** from the state space.

The WSL Kernel Language

"The quarks of programming"

The WSL Kernel Language

"The quarks of programming"

The compound statements are as follows; for any kernel language statements $\mathbf{S}_1$ and $\mathbf{S}_2$, the following are also kernel language statements:

- **Sequence:** $(\mathbf{S}_1;\ \mathbf{S}_2)$ executes $\mathbf{S}_1$ followed by $\mathbf{S}_2$;

The WSL Kernel Language

"The quarks of programming"

The compound statements are as follows; for any kernel language statements $\mathbf{S}_1$ and $\mathbf{S}_2$, the following are also kernel language statements:

- **Sequence:** $(\mathbf{S}_1;\ \mathbf{S}_2)$ executes $\mathbf{S}_1$ followed by $\mathbf{S}_2$;
- **Nondeterministic choice:** $(\mathbf{S}_1 \sqcap \mathbf{S}_2)$ chooses one of $\mathbf{S}_1$ or $\mathbf{S}_2$ for execution;

The WSL Kernel Language

"The quarks of programming"

The compound statements are as follows; for any kernel language statements $\mathbf{S}_1$ and $\mathbf{S}_2$, the following are also kernel language statements:

- **Sequence:** $(\mathbf{S}_1;\ \mathbf{S}_2)$ executes $\mathbf{S}_1$ followed by $\mathbf{S}_2$;
- **Nondeterministic choice:** $(\mathbf{S}_1\ \sqcap\ \mathbf{S}_2)$ chooses one of $\mathbf{S}_1$ or $\mathbf{S}_2$ for execution;
- **Recursion:** $(\mu X.\mathbf{S}_1)$ where X appearing in the body $\mathbf{S}_1$ represents a recursive procedure call.

Refinement in WSL

Refinement is defined in terms of the semantics: a statement $\mathbf{S}_2$ refines $\mathbf{S}_1$ if for each initial state, the set of final states for $\mathbf{S}_2$ is a subset of the set of final states for $\mathbf{S}_1$.

If $\mathbf{S}_1$ aborts for an initial state, then $\mathbf{S}_2$ can do anything for that initial state. (Recall that if the final set of states includes $\perp$ then it includes every other state as well).

So *anything* is a valid refinement of **abort**.

Also, $y := x$ is a refinement of **add**(y).

Refinement in WSL

Refinement is defined in the context of a set Δ of "applicability conditions". If $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ under the conditions Δ then we write:

$$\Delta \vdash \mathbf{S}_1 \ \leq \ \mathbf{S}_2$$

Refinement in WSL

Refinement is defined in the context of a set Δ of "applicability conditions". If $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ under the conditions Δ then we write:

$$\Delta \vdash \mathbf{S}_1 \ \leq \ \mathbf{S}_2$$

For example:

$$\Delta \vdash \textbf{abort} \ \leq \ \mathbf{S}$$

for any statement **S** and any set Δ

Refinement in WSL

Refinement is defined in the context of a set Δ of "applicability conditions". If $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ under the conditions Δ then we write:

$$\Delta \vdash \mathbf{S}_1 \ \leq \ \mathbf{S}_2$$

For example:

$$\Delta \vdash \textbf{abort} \ \leq \ \mathbf{S}$$

for any statement $\mathbf{S}$ and any set Δ

Also:

$$\Delta \vdash \textbf{add}(y) \ \leq \ y := e$$

for any variable y and any expression e.

Refinement in WSL

If $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$ and $\Delta \vdash \mathbf{S}_2 \leq \mathbf{S}_1$ then we say that $\mathbf{S}_1$ and $\mathbf{S}_2$ are **equivalent** and write:

$$\Delta \vdash \mathbf{S}_1 \approx \mathbf{S}_2$$

In this case, the semantic functions for $\mathbf{S}_1$ and $\mathbf{S}_2$ are identical.

Refinement in WSL

If $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$ and $\Delta \vdash \mathbf{S}_2 \leq \mathbf{S}_1$ then we say that $\mathbf{S}_1$ and $\mathbf{S}_2$ are **equivalent** and write:

$$\Delta \vdash \mathbf{S}_1 \approx \mathbf{S}_2$$

In this case, the semantic functions for $\mathbf{S}_1$ and $\mathbf{S}_2$ are identical.

For example:

$$\Delta \vdash \textbf{if B then } \mathbf{S}_1 \textbf{ else } \mathbf{S}_2 \textbf{ fi} \approx \textbf{if } \neg\textbf{B then } \mathbf{S}_2 \textbf{ else } \mathbf{S}_1 \textbf{ fi}$$

Refinement in WSL

If we give a programmer a specification, then we should be happy with *any* refinement of the specification as the implementation, because:

1. It terminates whenever it is required to; and
2. Whenever it terminates, the final state is one of the states allowed by the specification.

Language Extensions

The simple and easily-transformed WSL is extended into a powerful programming language by defining new constructs in terms of existing ones.

Assignments can be defined using **add** and guard statements:

$$x := 1$$

is defined as:

Language Extensions

The simple and easily-transformed WSL is extended into a powerful programming language by defining new constructs in terms of existing ones.

Assignments can be defined using **add** and guard statements:

$$x := 1$$

is defined as:

$$\textbf{add}(\langle x \rangle);\ [x = 1]$$

Language Extensions

The simple and easily-transformed WSL is extended into a powerful programming language by defining new constructs in terms of existing ones.

Assignments can be defined using **add** and guard statements:

$$x := 1$$

is defined as:

$$\textbf{add}(\langle x \rangle);\ [x = 1]$$

while:

$$x := x + 1$$

Language Extensions

The simple and easily-transformed WSL is extended into a powerful programming language by defining new constructs in terms of existing ones.

Assignments can be defined using **add** and guard statements:

$$x := 1$$

is defined as:

$$\textbf{add}(\langle x \rangle);\ [x = 1]$$

while:

$$x := x + 1$$

is defined as:

$$\textbf{add}(\langle x' \rangle);\ [x' = x + 1];\ \textbf{add}(\langle x \rangle);\ [x = x'];\ \textbf{remove}(\langle x' \rangle)$$

Language Extensions

The **if** statement

if B then S_1 else S_2 fi

Language Extensions

The **if** statement

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

can be implemented by a nondeterministic choice with guarded arms:

$$(([\mathbf{B}];\ \mathbf{S}_1)\ \sqcap\ ([\neg\mathbf{B}];\ \mathbf{S}_2))$$

Language Extensions

Loops are defined using recursion, for example the **while** loop:

while B do S od

Language Extensions

Loops are defined using recursion, for example the **while** loop:

while B do S od

is defined:

$$(\mu X.((([\mathbf{B}];\ \mathbf{S});\ X)\ \sqcap\ [\neg\mathbf{B}]))$$

Example WSL Programs

A deterministic program:

$$x := 1$$

For each initial state, s this program has a single final state in which the value of x is 1, and all other variables have their original values.

A non-deterministic program:

$$(x := 1 \ \sqcap \ x := 2)$$

For each initial state, s this program has a two final states, in one of which x has the value 1. In the other, x has the value 2. All other variables have their original values.

The Specification Statement

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

"Assign a new value $\mathbf{x}'$ to $\mathbf{x}$ such that $\mathbf{Q}$ is true, otherwise abort"

The formula $\mathbf{Q}$ defines the relationship between the new value $\langle x'_1, x'_2, \ldots, x'_n \rangle$ and the old value $\langle x_1, x_2, \ldots, x_n \rangle$

The Specification Statement

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

"Assign a new value $\mathbf{x}'$ to $\mathbf{x}$ such that $\mathbf{Q}$ is true, otherwise abort"

The formula $\mathbf{Q}$ defines the relationship between the new value $\langle x'_1, x'_2, \ldots, x'_n \rangle$ and the old value $\langle x_1, x_2, \ldots, x_n \rangle$

For example, add 1 to x:

$$\langle x \rangle := \langle x' \rangle.(x' = x + 1)$$

The Specification Statement

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

"Assign a new value $\mathbf{x}'$ to $\mathbf{x}$ such that $\mathbf{Q}$ is true, otherwise abort"

The formula $\mathbf{Q}$ defines the relationship between the new value $\langle x'_1, x'_2, \ldots, x'_n \rangle$ and the old value $\langle x_1, x_2, \ldots, x_n \rangle$

For example, add 1 to x:

$$\langle x \rangle := \langle x' \rangle.(x' = x + 1)$$

Swap the values of x and y:

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

The Specification Statement

Some more examples:

$$\langle x \rangle := \langle x' \rangle.(x^2 - 3x + 2 = x')$$

This will set x to the value $x^2 - 3x + 2$

$$\langle x \rangle := \langle x' \rangle.(x'^2 - 3x' + 2 = 0)$$

This will set x to either 1 or 2, the choice is made nondeterministically.

$$\langle x \rangle := \langle x' \rangle.((x' - 1)(x' - 2) = 0)$$

$$\langle x \rangle := \langle x' \rangle.(x' = 1 \ \vee \ x' = 2)$$

These are both equivalent to the second example above.

The Specification Statement

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

The formal definition is:

$$\{\exists \mathbf{x}'.\, \mathbf{Q}\};\ \mathbf{add}(\mathbf{x}');\ [\mathbf{Q}];\ \mathbf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'];\ \mathbf{remove}(\mathbf{x}')$$

The Specification Statement

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

The formal definition is:

$$\{\exists \mathbf{x}'.\, \mathbf{Q}\};\ \mathbf{add}(\mathbf{x}');\ [\mathbf{Q}];\ \mathbf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'];\ \mathbf{remove}(\mathbf{x}')$$

Informally, this means:

The Specification Statement

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

The formal definition is:

$$\{\exists \mathbf{x}'.\, \mathbf{Q}\};\ \mathbf{add}(\mathbf{x}');\ [\mathbf{Q}];\ \mathbf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'];\ \mathbf{remove}(\mathbf{x}')$$

Informally, this means:

1. If there is no value for $\mathbf{x}'$ which satisfies $\mathbf{Q}$, then **abort**;

The Specification Statement

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

The formal definition is:

$$\{\exists \mathbf{x}'.\, \mathbf{Q}\};\ \mathbf{add}(\mathbf{x}');\ [\mathbf{Q}];\ \mathbf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'];\ \mathbf{remove}(\mathbf{x}')$$

Informally, this means:

1. If there is no value for $\mathbf{x}'$ which satisfies $\mathbf{Q}$, then **abort**;
2. Add new variables $\mathbf{x}'$ to the state;

The Specification Statement

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

The formal definition is:

$$\{\exists \mathbf{x}'.\, \mathbf{Q}\};\ \mathbf{add}(\mathbf{x}');\ [\mathbf{Q}];\ \mathbf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'];\ \mathbf{remove}(\mathbf{x}')$$

Informally, this means:

1. If there is no value for $\mathbf{x}'$ which satisfies $\mathbf{Q}$, then **abort**;
2. Add new variables $\mathbf{x}'$ to the state;
3. Restrict the nondeterminacy of the **add** so that $\mathbf{Q}$ is true.

The Specification Statement

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

The formal definition is:

$$\{\exists \mathbf{x}'.\, \mathbf{Q}\};\ \mathbf{add}(\mathbf{x}');\ [\mathbf{Q}];\ \mathbf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'];\ \mathbf{remove}(\mathbf{x}')$$

Informally, this means:

1. If there is no value for $\mathbf{x}'$ which satisfies $\mathbf{Q}$, then **abort**;
2. Add new variables $\mathbf{x}'$ to the state;
3. Restrict the nondeterminacy of the **add** so that $\mathbf{Q}$ is true.
4. Copy $\mathbf{x}'$ into $\mathbf{x}$

The Specification Statement

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

The formal definition is:

$$\{\exists \mathbf{x}'.\, \mathbf{Q}\};\ \mathbf{add}(\mathbf{x}');\ [\mathbf{Q}];\ \mathbf{add}(\mathbf{x});\ [\mathbf{x} = \mathbf{x}'];\ \mathbf{remove}(\mathbf{x}')$$

Informally, this means:

1. If there is no value for $\mathbf{x}'$ which satisfies $\mathbf{Q}$, then **abort**;
2. Add new variables $\mathbf{x}'$ to the state;
3. Restrict the nondeterminacy of the **add** so that $\mathbf{Q}$ is true.
4. Copy $\mathbf{x}'$ into $\mathbf{x}$
5. Remove $\mathbf{x}'$ from the state.

The Specification Statement

Specification of a sorting program:

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

The output must be sorted and a permutation of the input.

It precisely describes *what* we want our sorting program to do without saying *how* it is to be achieved

Other Specification Statements

The specification statement

$$\mathbf{x} : [\mathrm{Pre}, \mathrm{Post}]$$

of Morgan et al may be defined:

$$\{\mathrm{Pre}\};\ \mathrm{add}(\mathbf{x});\ [\mathrm{Post}]$$

Back's atomic description:

$$\mathbf{x}/\mathbf{y}.\mathbf{Q}$$

may be defined as:

$$\{\exists \mathbf{x}.\, \mathbf{Q}\};\ \mathrm{add}(\mathbf{x});\ [\mathbf{Q}];\ \mathrm{remove}(\mathbf{y})$$

More Language Extensions

- **for** loops
- Dijkstra's Guarded Command Language
- Loops with multiple **exit**s
- Mutually recursive procedures (labels and **goto**s)
- Local variables
- Procedures and functions with parameters
- Expressions with side-effects
- Assembler language

More Language Extensions

Unbounded loops, or "Floops":

do $\ldots$ **exit**$(1)\ldots$ **od**

do do $\ldots$ **exit**$(1)\ldots$ **exit**$(2)\ldots$ **od od**

Within unbounded loops, the statement **exit**(n) terminates the enclosing n nested loops. Any **exit**(1) will terminate the enclosing loop. **exit**(2) will terminate a double loop, and so on.

A **do** $\ldots$ **od** loop can only be terminated by an **exit** statement.

A **simple terminal statement** is any simple statement (something other than an **if** statement or **do** $\ldots$ **od** loop) which can terminate the program.

More Language Extensions

Example:

```
do do last := item[i];
        i := i + 1;
        if i = n + 1 then write(line); exit(2) fi;
        if item[i] ≠ last
           then write(line); exit(1);
                   if i = j then exit(2) fi
            else line :=line ++ ", " ++ number[i] fi od;
    if i = j then exit(1) fi
    line := item[i] ++ " " ++ number[i] od
```

Which are the simple terminal statements?

More Language Extensions

Example:

```
do do last := item[i];
        i := i + 1;
        if i = n + 1 then write(line); exit(2) fi;
        if item[i] ≠ last
           then write(line); exit(1);
                   if i = j then exit(2) fi
            else line :=line ++ ", " ++ number[i] fi od;
    if i = j then exit(1) fi
    line := item[i] ++ " " ++ number[i] od
```

Which are the simple terminal statements?

More Language Extensions

```
do do last := item[i];
        i := i + 1;
        if i = n + 1 then write(line); exit(2) fi;
        if item[i] ≠ last
           then write(line); exit(1);
                   if i = j then exit(2) fi
            else line :=line ++ ", " ++ number[i] fi od;
    if i = j then exit(2) fi
    line := item[i] ++ " " ++ number[i] od;
skip
```

Now, which are the simple terminal statements?

More Language Extensions

```
do do last := item[i];
        i := i + 1;
        if i = n + 1 then write(line); exit(2) fi;
        if item[i] ≠ last
           then write(line); exit(1);
                   if i = j then exit(2) fi
            else line :=line ++ ", " ++ number[i] fi od;
     if i = j then exit(2) fi
     line := item[i] ++ " " ++ number[i] od;
skip
```

Now, which are the simple terminal statements?

Action Systems

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

- A collection of mutually recursive parameterless procedures
- **call** A_i is a call to action A_i
- A special statement **call** Z causes immediate termination of the whole action system
- An action system is a single statement which can appear as a component of another statement (including another action system)

Regular Action Systems

actions A_1 :

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

...

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

- If execution of each action body **S**$_i$ always leads to an action call (or **call** Z) then we have a **Regular Action System**.
- In this case, action calls are like **goto**s: no action ever returns.
- The system can only terminate via **call** Z

Regular Action Systems

```
var ⟨m := 0, p := 0, last := “”⟩ :
    actions prog :
    prog  ≡ line := “”; m := 0; i := 1;
            call inhere end
    loop  ≡ i := i + 1;
            if i = n + 1 then call alldone fi;
            m := 1;
            if item[i] ≠ last
              then write(line var os);
                   line := “ ”; m := 0;
                   call inhere fi;
            call more end
    inhere ≡ p := number[i];
             line := item[i]; line := line ++ “ ” ++ p;
             call more end
    more  ≡ if m = 1
              then p := number[i];
                   line := line ++ “, ” ++ p fi;
            last := item[i];
            call loop end
    alldone ≡ write(line var os); call Z end endactions end
```

Transformed Version

```
var ⟨m := 0, p := 0, last := “”⟩ :
    line := “”;
    m := 0;
    i := 1;
    do p := number[i];
        line := item[i];
        line := line ++ “ ” ++ p;
        do if m = 1
              then p := number[i]; line := line ++ “, ” ++ p fi;
           last := item[i];
           i := i + 1;
           if i = n + 1
              then write(line var os); exit(2) fi;
           m := 1;
           if item[i] ≠ last
              then write(line var os); line := “”; m := 0; exit(1) fi od od end
```

The FermaT Transformation System

The FermaT Transformation System

- The result of over 25 years research and development in transformation theory

The FermaT Transformation System

- The result of over 25 years research and development in transformation theory
- Uses a Wide Spectrum Language, called WSL, which was developed in parallel with the development of the transformation theory

The FermaT Transformation System

- The result of over 25 years research and development in transformation theory
- Uses a Wide Spectrum Language, called WSL, which was developed in parallel with the development of the transformation theory
- FermaT implements over 100 transformations together with their applicability conditions

The FermaT Transformation System

- The result of over 25 years research and development in transformation theory
- Uses a Wide Spectrum Language, called WSL, which was developed in parallel with the development of the transformation theory
- FermaT implements over 100 transformations together with their applicability conditions
- Transformations are implemented in an extension of WSL, called $\mathcal{METAWSL}$

The FermaT Transformation System

- FermaT is implemented almost entirely in $\mathcal{META}$WSL
- Therefore, FermaT can transform its own source code!
- This is used on a regular bases as part of the build process

The FermaT Transformation System

- FermaT is use in commercial applications:
  - Assembler to C migration
  - Assembler to COBOL migration
  - Program Slicing
  - Program Comprehension
  - System Reengineering

Program Transformation

A program transformation is an operation which can be applied to any program and returns a semantically equivalent program.

In FermaT, transformations are applied to programs written in WSL.

Program Transformation

A program transformation is an operation which can be applied to any program and returns a semantically equivalent program.

In FermaT, transformations are applied to programs written in WSL.

For example:

if $x = 0$ **then** $y := 1$ **else** $y := 2$ **fi**

is semantically equivalent to:

if $x \neq 0$ **then** $y := 2$ **else** $y := 1$ **fi**

The WSL Language used in FermaT

- Assignment: $x := e$

The WSL Language used in FermaT

- Assignment: $x := e$
- Assertion: $\{\mathbf{Q}\}$

The WSL Language used in FermaT

- Assignment: $x := e$
- Assertion: $\{\mathbf{Q}\}$
- Specification Statement: $\mathbf{x} := \mathbf{x}'.(\mathbf{Q})$

The WSL Language used in FermaT

- Assignment: $x := e$
- Assertion: $\{\mathbf{Q}\}$
- Specification Statement: $\mathbf{x} := \mathbf{x}'.(\mathbf{Q})$
- Skip statement: **skip**

The WSL Language used in FermaT

- Assignment: $x := e$
- Assertion: $\{\mathbf{Q}\}$
- Specification Statement: $\mathbf{x} := \mathbf{x}'.(\mathbf{Q})$
- Skip statement: **skip**
- Abort statement: **abort**

The WSL Language used in FermaT

- Assignment: $x := e$
- Assertion: $\{\mathbf{Q}\}$
- Specification Statement: $\mathbf{x} := \mathbf{x}'.(\mathbf{Q})$
- Skip statement: **skip**
- Abort statement: **abort**
- If statement: **if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**

The WSL Language used in FermaT

- Assignment: $x := e$
- Assertion: $\{\mathbf{Q}\}$
- Specification Statement: $\mathbf{x} := \mathbf{x}'.(\mathbf{Q})$
- Skip statement: **skip**
- Abort statement: **abort**
- If statement: **if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**
- While loop: **while B do** $\mathbf{S}_1$ **od**

The WSL Language used in FermaT

- Assignment: $x := e$
- Assertion: $\{\mathbf{Q}\}$
- Specification Statement: $\mathbf{x} := \mathbf{x}'.(\mathbf{Q})$
- Skip statement: **skip**
- Abort statement: **abort**
- If statement: **if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**
- While loop: **while B do** $\mathbf{S}_1$ **od**
- For loop: **for** $i := b$ **to** e **step** s **do S od**

The WSL Language used in FermaT

- Assignment: $x := e$
- Assertion: $\{\mathbf{Q}\}$
- Specification Statement: $\mathbf{x} := \mathbf{x}'.(\mathbf{Q})$
- Skip statement: **skip**
- Abort statement: **abort**
- If statement: **if B then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**
- While loop: **while B do** $\mathbf{S}_1$ **od**
- For loop: **for** $i := b$ **to** e **step** s **do S od**
- Floop: **do** $\ldots$ **exit**$(n)\ldots$ **exit**$(m)\ldots$ **od**

The WSL Language used in FermaT

- Procedure: **proc** $F(x)$ $\equiv$ **S end**

The WSL Language used in FermaT

- Procedure: **proc** $F(x)$ $\equiv$ **S end**
- Function: **funct** $f(x)$ $\equiv$ **S**; (e) **end**

The WSL Language used in FermaT

- Procedure: **proc** $F(x)$ $\equiv$ **S end**
- Function: **funct** $f(x)$ $\equiv$ **S**; (e) **end**
- Boolean Function: **bfunct** $B(x)$ $\equiv$ **S**; $(\mathbf{B})$ **end**

The WSL Language used in FermaT

- Procedure: **proc** $F(x)$ $\equiv$ **S end**
- Function: **funct** $f(x)$ $\equiv$ **S**; (e) **end**
- Boolean Function: **bfunct** $B(x)$ $\equiv$ **S**; $(\mathbf{B})$ **end**
- Where clause: **begin S where** definitions **end**

The WSL Language used in FermaT

- Procedure: **proc** $F(x)$ $\equiv$ **S end**
- Function: **funct** $f(x)$ $\equiv$ **S**; (e) **end**
- Boolean Function: **bfunct** $B(x)$ $\equiv$ **S**; $(\mathbf{B})$ **end**
- Where clause: **begin S where** definitions **end**
- Action System:

The WSL Language used in FermaT

- Procedure: **proc** $F(x)$ $\equiv$ **S end**
- Function: **funct** $f(x)$ $\equiv$ **S**; (e) **end**
- Boolean Function: **bfunct** $B(x)$ $\equiv$ **S**; $(\mathbf{B})$ **end**
- Where clause: **begin S where** definitions **end**
- Action System:

  actions A_1 :
  A_1 $\equiv$ $\mathbf{S}_1$ **end**
  A_2 $\equiv$ $\mathbf{S}_2$ **end**
  $\ldots$
  A_n $\equiv$ $\mathbf{S}_n$ **end endactions**

The WSL Language used in FermaT

Each of these WSL Language constructs is defined in terms of the kernel language.

Transformations in FermaT map directly from WSL to WSL without translating to the kernel language level.

Transformation Proof Methods

Transformation proof methods include:

Transformation Proof Methods

Transformation proof methods include:

- Translate to the kernel language and prove the transformation via:
  - Denotational semantics (comparing the semantic functions); or
  - Weakest preconditions

Transformation Proof Methods

Transformation proof methods include:

- Translate to the kernel language and prove the transformation via:
  - Denotational semantics (comparing the semantic functions); or
  - Weakest preconditions
- Prove via weakest preconditions without using the kernel language

Transformation Proof Methods

Transformation proof methods include:

- Translate to the kernel language and prove the transformation via:
  - Denotational semantics (comparing the semantic functions); or
  - Weakest preconditions
- Prove via weakest preconditions without using the kernel language
- Prove by applying a sequence of existing transformations and proof rules

Program Transformation

WSL includes both *abstract specifications* and *executable programs* within the same language. This means that:

- Refinement of a specification into an executable program, and
- Reverse engineering from a program to a specification

are both examples of program transformations.