

Formal Methods (6)

Sequential Program Development

Jean-Raymond Abrial (ETHZ)

May 2005

Purpose of this Lecture

- To present a **formal approach** for developing **sequential programs**
- To present a large number of examples:
 - **array** programs
 - **pointer** program
 - **numerical** programs

Introduction

- A typical **sequential program** is made of :
 - a number of **MULTIPLE ASSIGNMENTS** (**:=**)
 - **scheduled** by means of some :
 - **CONDITIONAL** operators (**if**)
 - **ITERATIVE** operators (**while**)
 - **SEQUENTIAL** operators (**;**)

An Example

```
while  $j \neq m$  do  
  if  $g(j + 1) > x$  then  
     $j := j + 1$   
  elsif  $k = j$  then  
     $k, j := k + 1, j + 1$   
  else  
     $k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1)$   
  end  
end ;  
 $p := k$ 
```

Statements for a Pidgin Programming Language

while *condition* **do** *statement* **end**

if *condition* **then** *statement* **else** *statement* **end**

if *condition* **then** *statement* **elsif** ... **else** *statement* **end**

statement ; *statement*

variable_list := *expression_list*

An Event Design Approach (1)

- **Separating** completely in the design:
 - the individual **assignments**
 - from their **scheduling**

- This approach favors:
 - the **distribution** of computation
 - over its **centralization**

An Event Design Approach (2)

- Each individual assignment is formalized by a **guarded event** made of:
 - A **firing condition**: the guard,
 - An **action**: the multiple assignment.
- These events are scheduled **implicitly**.

```
while  $j \neq m$  do  
  if  $g(j + 1) > x$  then  
     $j := j + 1$   
  elsif  $k = j$  then  
     $k, j := k + 1, j + 1$   
  else  
     $k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1)$   
  end  
end ;  
 $p := k$ 
```

```
when  
   $j \neq m$   
   $g(j + 1) > x$   
then  
   $j := j + 1$   
end
```

```
while  $j \neq m$  do  
  if  $g(j + 1) > x$  then  
     $j := j + 1$   
  elsif  $k = j$  then  
     $k, j := k + 1, j + 1$   
  else  
     $k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1)$   
  end  
end ;  
 $p := k$ 
```

```
when  
   $j \neq m$   
   $g(j + 1) \leq x$   
   $k = j$   
then  
   $k, j := k + 1, j + 1$   
end
```

```
while  $j \neq m$  do  
  if  $g(j + 1) > x$  then  
     $j := j + 1$   
  elsif  $k = j$  then  
     $k, j := k + 1, j + 1$   
  else  
     $k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1)$   
  end  
end ;  
 $p := k$ 
```

```
when  
   $j \neq m$   
   $g(j + 1) \leq x$   
   $k \neq j$   
then  
   $k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1)$   
end
```

```
while  $j \neq m$  do  
  if  $g(j + 1) > x$  then  
     $j := j + 1$   
  elsif  $k = j$  then  
     $k, j := k + 1, j + 1$   
  else  
     $k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1)$   
  end  
end ;  
 $p := k$ 
```

```
when  
   $j = m$   
then  
   $p := k$   
end
```

The Various Events of our Program

when

$j \neq m$

$g(j + 1) > x$

then

$j := j + 1$

end

when

$j \neq m$

$g(j + 1) \leq x$

$k = j$

then

$k, j := k + 1, j + 1$

end

when

$j \neq m$

$g(j + 1) \leq x$

$k \neq j$

then

$k, j, g := \dots$

end

when

$j = m$

then

$p := k$

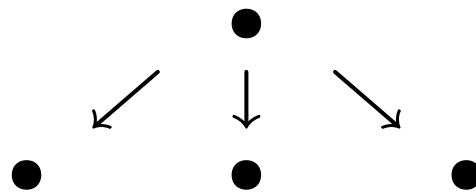
end

Composing a Program from Events

- We have just **decomposed** a program into separate events
- Our approach will consists in doing the **reverse operation**
- We shall **construct the events** first
- And then **compose our program** from these events

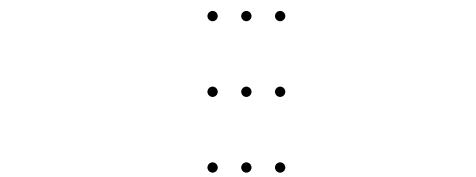
Principles of the Event Approach

Specification Phase



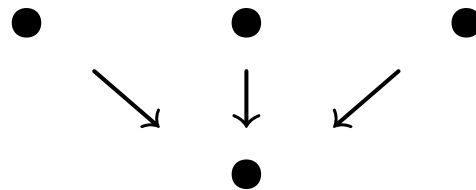
initial event: **Specification**

Design Phase



new events: **Refinements**

Merging Phase



final event: **Program**

Using Event Systems for Developing Sequential Programs

- **Sequential Programs** are usually specified by means of:
 - A **pre-condition**
 - and a **post-condition**
- It is represented with a **Hoare-triple**

$$\{Pre\} \quad P \quad \{Post\}$$

Example 1: The search Program

Example 1: The search Program

- We are given (Pre-condition)

Example 1: The search Program

- We are given (Pre-condition)
 - a natural number n : $n \in \mathbb{N}$

Example 1: The search Program

- We are given (Pre-condition)
 - a natural number n : $n \in \mathbb{N}$
 - an array f of n elements built on a set S : $f \in 1 .. n \rightarrow S$

Example 1: The search Program

- We are given (Pre-condition)
 - a natural number n : $n \in \mathbb{N}$
 - an array f of n elements built on a set S : $f \in 1 .. n \rightarrow S$
 - a value v known to be in the array: $v \in \text{ran}(f)$

Example 1: The search Program

- We are given (**Pre-condition**)
 - a natural number n : $n \in \mathbb{N}$
 - an array f of n elements built on a set S : $f \in 1 .. n \rightarrow S$
 - a value v known to be in the array: $v \in \text{ran}(f)$

- We are looking for (**Post-condition**)

Example 1: The search Program

- We are given (**Pre-condition**)
 - a natural number n : $n \in \mathbb{N}$
 - an array f of n elements built on a set S : $f \in 1 .. n \rightarrow S$
 - a value v known to be in the array: $v \in \text{ran}(f)$
- We are looking for (**Post-condition**)
 - an index r in the domain of the array: $r \in \text{dom}(f)$

Example 1: The search Program

- We are given (**Pre-condition**)
 - a natural number n : $n \in \mathbb{N}$
 - an array f of n elements built on a set S : $f \in 1 .. n \rightarrow S$
 - a value v known to be in the array: $v \in \text{ran}(f)$

- We are looking for (**Post-condition**)
 - an index r in the domain of the array: $r \in \text{dom}(f)$
 - such that $f(r) = v$

Example 1: The search Program

- We are given (**Pre-condition**)
 - a natural number n : $n \in \mathbb{N}$
 - an array f of n elements built on a set S : $f \in 1 .. n \rightarrow S$
 - a value v known to be in the array: $v \in \text{ran}(f)$
- We are looking for (**Post-condition**)
 - an index r in the domain of the array: $r \in \text{dom}(f)$
 - such that $f(r) = v$

$$\left\{ \begin{array}{l} n \in \mathbb{N} \\ f \in 1 .. n \rightarrow S \\ v \in \text{ran}(f) \end{array} \right\} \quad \text{search} \quad \{ r \in f^{-1}[\{v\}] \}$$

Encoding a Hoare-triple in an Event System

- The input parameters are **constants**
- The **pre-condition** corresponds to **properties** of these constants
- The output parameters are **variables**
- The **post-condition** is in the **event** (most of the time unique)

Encoding a Hoare-triple in an Event System

$$\left\{ \begin{array}{l} n \in \mathbb{N} \\ f \in 1..n \rightarrow S \\ v \in \text{ran}(f) \end{array} \right\}$$

search

$$\{ r \in f^{-1}[\{v\}] \}$$

Encoding a Hoare-triple in an Event System

$$\left\{ \begin{array}{l} n \in \mathbb{N} \\ f \in 1..n \rightarrow S \\ v \in \text{ran}(f) \end{array} \right\} \quad \text{search} \quad \{ r \in f^{-1}[\{v\}] \}$$

carrier sets: S

constants: n, f, v

variables: r

prp0_1: $n \in \mathbb{N}$

prp0_2: $f \in 1..n \rightarrow S$

prp0_3: $v \in \text{ran}(f)$

inv0_1: $r \in \mathbb{N}$

Encoding a Hoare-triple in an Event System

$$\left\{ \begin{array}{l} n \in \mathbb{N} \\ f \in 1..n \rightarrow S \\ v \in \text{ran}(f) \end{array} \right\} \quad \text{search} \quad \{ r \in f^{-1}[\{v\}] \}$$

carrier sets: S

constants: n, f, v

variables: r

prp0_1: $n \in \mathbb{N}$

prp0_2: $f \in 1..n \rightarrow S$

prp0_3: $v \in \text{ran}(f)$

inv0_1: $r \in \mathbb{N}$

init

$r : \in \mathbb{N}$

search

$r : \in f^{-1}[\{v\}]$

Applying the Feasibility Rule on our Example

init
 $r \in \mathbb{N}$

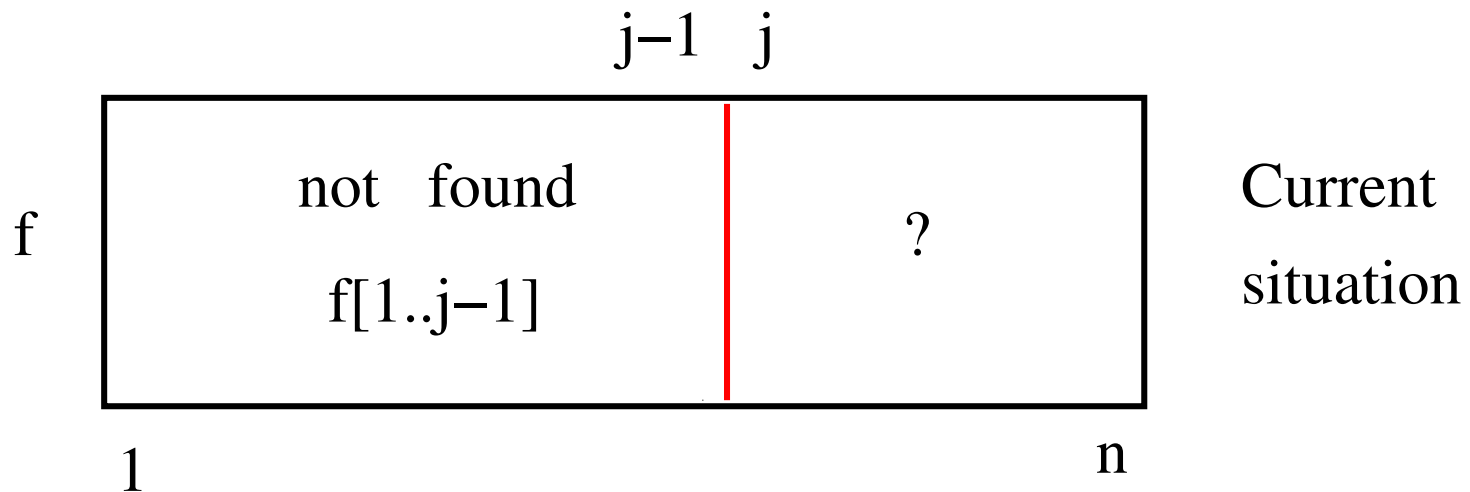
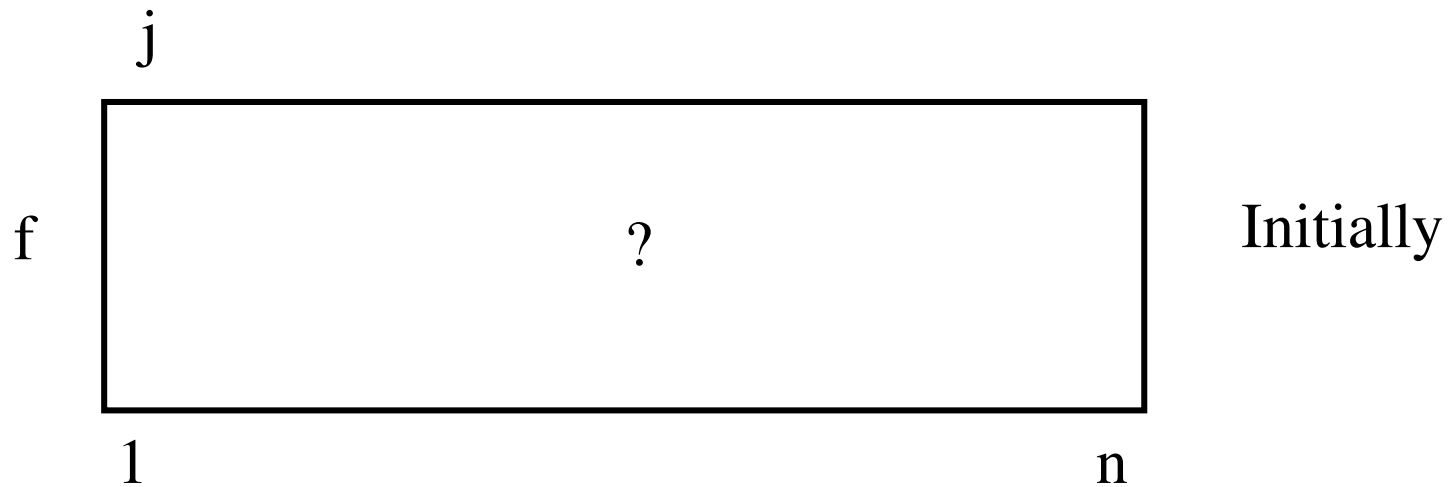
search
 $r \in f^{-1}[\{v\}]$

$n \in \mathbb{N}$
 $f \in 1..n \rightarrow S$
 $v \in \text{ran}(f)$
 \Rightarrow
 $\mathbb{N} \neq \emptyset$

$n \in \mathbb{N}$
 $f \in 1..n \rightarrow S$
 $v \in \text{ran}(f)$
 $r \in \mathbb{N}$
 \Rightarrow
 $f^{-1}[\{v\}] \neq \emptyset$

Ideas for a Refinement

We introduce a **new variables j** , initially set to 1



Development of the **search** Program: Refinement

carrier sets: S

constants: n, f, v

variables: r, j

inv1_1: $j \in 1 .. n$

inv1_2: $v \notin f[1 .. j - 1]$

init

$r := \text{N}$

$j := 1$

progress

when

$f(j) \neq v$

then

$j := j + 1$

end

search

when

$f(j) = v$

then

$r := j$

end

To be Proved (as usual)

- Events **search** and **init** refine their abstractions
- New Event **progress** refines skip
- The exhibited **variant** is a natural number
- New Event **progress** decreases the variant
- The system is **deadlock free**

Proving Refinement of event **search**

(abstract_)search
 $r \in f^{-1}[\{v\}]$

(concrete_)search
when
 $f(j) = v$
then
 $r := j$
end

- Before-after predicates

$$r' \in f^{-1}[\{v\}]$$

$$\begin{aligned} r' &= j \\ j' &= j \end{aligned}$$

- We have a case of **superposition** here

Proving Refinement of search

- Applying rule **IMP_REF**

Properties of the constants

$$n \in \mathbb{N}$$

$$f \in 1..n \rightarrow S$$

$$v \in \text{ran}(f)$$

Abstract Invariant

$$r \in \mathbb{N}$$

Concrete Invariants

$$j \in 1..n$$

$$v \notin f[1..j-1]$$

Concrete guard

$$f(j) = v$$

Concrete before-after predicate

$$r' = j$$

$$j' = j$$

\Rightarrow

Abstract before-after predicate

\Rightarrow

$$r' \in f^{-1}[\{v\}]$$

This reduces to:

$$j \in f^{-1}[f[\{j}]]$$

Proving Refinement of new event **progress**

```
(implicit_abstract_)progress  
skip
```

```
(concrete_)progress  
when  
   $f(j) \neq v$   
then  
   $j := j + 1$   
end
```

- Before-After predicates

$$r' = r$$

$$r' = r$$
$$j' = j + 1$$

- Applying rule EQL_REF is trivial (common actions are identical)
- We have just to apply rule INV_REF

Proof Obligation for refinement of progress (1)

- Applying rule INV_REF

Properties of the constants

$$n \in \mathbb{N}$$

$$f \in 1..n \rightarrow S$$

$$v \in \text{ran}(f)$$

Abstract Invariant

$$r \in \mathbb{N}$$

Concrete Invariants

$$j \in \text{dom}(f)$$

$$v \notin f[1..j-1]$$

Concrete guard

$$f(j) \neq v$$

\Rightarrow

\Rightarrow

Modified invariant **inv1_1**

$$j+1 \in \text{dom}(f)$$

We are Left to Prove:

$$\begin{aligned}v &\in f[1 .. n] \\j &\in 1 .. n \\v &\notin f[1 .. j - 1] \\f(j) &\neq v \\ \Rightarrow \\j + 1 &\in 1 .. n\end{aligned}$$

We can prove the following lemma (**by contradiction**)

$$\begin{aligned}v &\in f[1 .. n] \\j &\in 1 .. n \\v &\notin f[1 .. j - 1] \\f(j) &\neq v \\ \Rightarrow \\j &\neq n\end{aligned} \quad \begin{array}{l} \text{then it remains to prove} \\ \text{easy} \end{array}$$

$$\begin{aligned}v &\in f[1 .. n] \\j &\in 1 .. n \\v &\notin f[1 .. j - 1] \\f(j) &\neq v \\j &\neq n \\ \Rightarrow \\j + 1 &\in 1 .. n\end{aligned}$$

Proof Obligation for refinement of progress (2)

- Applying rule INV_REF

Properties of the constants

$$n \in \mathbb{N}$$

$$f \in 1..n \rightarrow S$$

$$v \in \text{ran}(f)$$

Abstract Invariant

$$r \in \mathbb{N}$$

Concrete Invariants

$$j \in \text{dom}(f)$$

$$v \notin f[1..j-1]$$

Concrete guard

$$f(j) \neq v$$

\Rightarrow

\Rightarrow

Modified invariant **inv1_2**

$$v \notin f[1..j+1-1]$$

We are Left to Prove:

$$\begin{aligned}j &\in 1 \dots n \\v &\notin f[1 \dots j - 1] \\f(j) &\neq v \\ \Rightarrow \\v &\notin f[1 \dots j]\end{aligned}$$

We can use the following:

$$f[1 \dots j] = f[1 \dots j - 1] \cup \{f(j)\}$$

New Event **progress** Must Decrease a Variant

inv1_1: $j \in 1 .. n$

inv1_2: $v \notin f[1 .. j - 1]$

```
progress
  when
     $f(j) \neq v$ 
  then
     $j := j + 1$ 
  end
```

- We propose the variant $n - j$
- We have to prove two things:
 - The variant is a **natural number** (trivial according to **inv1_1**)
 - The variant is **decreased** by the new event **progress**

Deadlock Freeness

Deadlock freeness is obvious

progress

when

$f(j) \neq v$

then

$j := j + 1$

end

search

when

$f(j) = v$

then

$r := j$

end

Constructing the Final Program

We are using some **Merging Rules** to build the final program

init

$r \in \mathbb{N}$

$j := 1$

progress

when

$f(j) \neq v$

then

$j := j + 1$

end

search

when

$f(j) = v$

then

$r := j$

end

Merging Rule (1)

when
P
Q
then
S
end

when
P
 $\neg Q$
then
T
end

\rightsquigarrow

when
P
then
while *Q* **do**
S
end;
T
end

M_WHILE1

- Side Conditions:

- *P* must be invariant under *S*
- The **first event** must have been introduced at **one** refinement step below the second one.

Merging Rule (2)

| | | | | |
|--------------------------------------|--|--------------------|---|--------------|
| <pre>when P Q then S end</pre> | <pre>when P ¬ Q then T end</pre> | \rightsquigarrow | <pre>when P then if Q then S else T end end</pre> | M_IF1 |
|--------------------------------------|--|--------------------|---|--------------|

- Side Conditions: The **disjunctive negation** of the previous side conditions

Merging Rule (3)

| | | | | |
|--|---|--------------------|--|-----------------|
| <pre>when P then Q end end</pre> | <pre>when P then ¬ Q end if R then T else U end end</pre> | \rightsquigarrow | <pre>when P then if Q then S elsif R then T else U end end end</pre> | M_ELSIF1 |
|--|---|--------------------|--|-----------------|

Merging Rule (4)

when
 Q
then
 S
end

when
 $\neg Q$
then
 T
end

\rightsquigarrow

while Q **do**
 S
end;
 T

M_WHILE2

- Side Conditions:

- The **first event** must have been introduced at **one refinement step below the second one.**

Merging Rule (5)

| | | | | |
|--|--|--------------------|---|--------------|
| when <i>Q</i> then <i>S</i> end | when $\neg Q$ then <i>T</i> end | \rightsquigarrow | if <i>Q</i> then <i>S</i> else <i>T</i> end | M_IF2 |
|--|--|--------------------|---|--------------|

- Side Conditions: The **negation** of the previous side condition

Merging Rule (6)

| | | | | |
|-------------|---------------------------|--------------------|------------------------------|--|
| when | when | | if Q then | |
| Q | $\neg Q$ | | S | |
| then | then | | elsif R then | |
| S | if R then | | T | |
| end | T | \rightsquigarrow | else | |
| | else | | U | |
| | U | | end | |
| | end | | | |
| | end | | | |

M_ELSIF2

Merging Rule (7)

when
P
Q
then
S
end

when
P
 $\neg Q$
then
skip
end

\rightsquigarrow

when
P
then
while *Q* do
S
end
end

M_WHILE3

Merging Rule (8)

when
 Q
then
 S
end

when
 $\neg Q$
then
 skip
end

\rightsquigarrow

while Q **do**
 S
end

M_WHILE4

Applying Rule **M_WHILE2**

```
progress
  when
     $f(j) \neq v$ 
  then
     $j := j + 1$ 
  end
```

```
search
  when
     $f(j) = v$ 
  then
     $r := j$ 
  end
```

```
progress_search
  while  $f(j) \neq v$  do
     $j := j + 1$ 
  end;
   $r := j$ 
```

Merging Rule **M_WHILE2**

when
 Q
then
 S
end

when
 $\neg Q$
then
 T
end

\rightsquigarrow

while Q **do**
 S
end;
 T

M_WHILE2

- Side Conditions:

- The **first event** must have been introduced at **one refinement step below the second one.**

Final Rule M_INIT

- Once we have obtained an “event” **without guard**
- We add to it the event **init** by **sequential composition**
- We then obtain the final “program”

Applying Rule M_INIT

init

$r := \in \mathbb{N}$

$j := 1$

progress_search

while $f(j) \neq v$ **do**

$j := j + 1$

end;

$r := j$

$\left\{ \begin{array}{l} n \in \mathbb{N} \\ f \in 1..n \rightarrow S \\ v \in \text{ran}(f) \end{array} \right\}$

search_program

$j := 1;$

while $f(j) \neq v$ **do**

$j := j + 1$

end;

$r := j$

$\{ r \in f^{-1}[\{v\}] \}$

Example 2: The Very Classical Binary Search

- Almost the **same specification** as in Example 1
- It will show the usage of **more merging rules**

Binary Search

- We are given (Pre-condition)

Binary Search

- We are given (Pre-condition)
 - a natural number n : $n \in \mathbb{N}$

Binary Search

- We are given (Pre-condition)
 - a natural number n : $n \in \mathbb{N}$
 - a sorted array f of n elements built on a set \mathbb{N} : $f \in 1..n \rightarrow \mathbb{N}$

Binary Search

- We are given (Pre-condition)
 - a natural number n : $n \in \mathbb{N}$
 - a **sorted** array f of n elements built on a set \mathbb{N} : $f \in 1..n \rightarrow \mathbb{N}$
 - a value v known to be in the array: $v \in \text{ran}(f)$

Binary Search

- **We are given** (Pre-condition)
 - a natural number n : $n \in \mathbb{N}$
 - a **sorted** array f of n elements built on a set \mathbb{N} : $f \in 1..n \rightarrow \mathbb{N}$
 - a value v known to be in the array: $v \in \text{ran}(f)$

- **We are looking for** (Post-condition)

Binary Search

- **We are given** (Pre-condition)
 - a natural number n : $n \in \mathbb{N}$
 - a **sorted** array f of n elements built on a set \mathbb{N} : $f \in 1..n \rightarrow \mathbb{N}$
 - a value v known to be in the array: $v \in \text{ran}(f)$

- **We are looking for** (Post-condition)
 - an index r in the domain of the array: $r \in \text{dom}(f)$

Binary Search

- **We are given** (Pre-condition)
 - a natural number n : $n \in \mathbb{N}$
 - a **sorted** array f of n elements built on a set \mathbb{N} : $f \in 1..n \rightarrow \mathbb{N}$
 - a value v known to be in the array: $v \in \text{ran}(f)$

- **We are looking for** (Post-condition)
 - an index r in the domain of the array: $r \in \text{dom}(f)$
 - such that $f(r) = v$

Binary Search: the State

constants: n, f, v

variables: r

inv0_1: $r \in \mathbb{N}$

prp0_1: $n \in \mathbb{N}$

prp0_2: $f \in 1..n \rightarrow \mathbb{N}$

prp0_3: $\forall i, j \cdot \left(\begin{array}{l} i \in 1..n \\ j \in 1..n \\ i \leq j \\ \Rightarrow \\ f(i) \leq f(j) \end{array} \right)$

prp0_4: $v \in \text{ran}(f)$

Binary Search: the Events

init

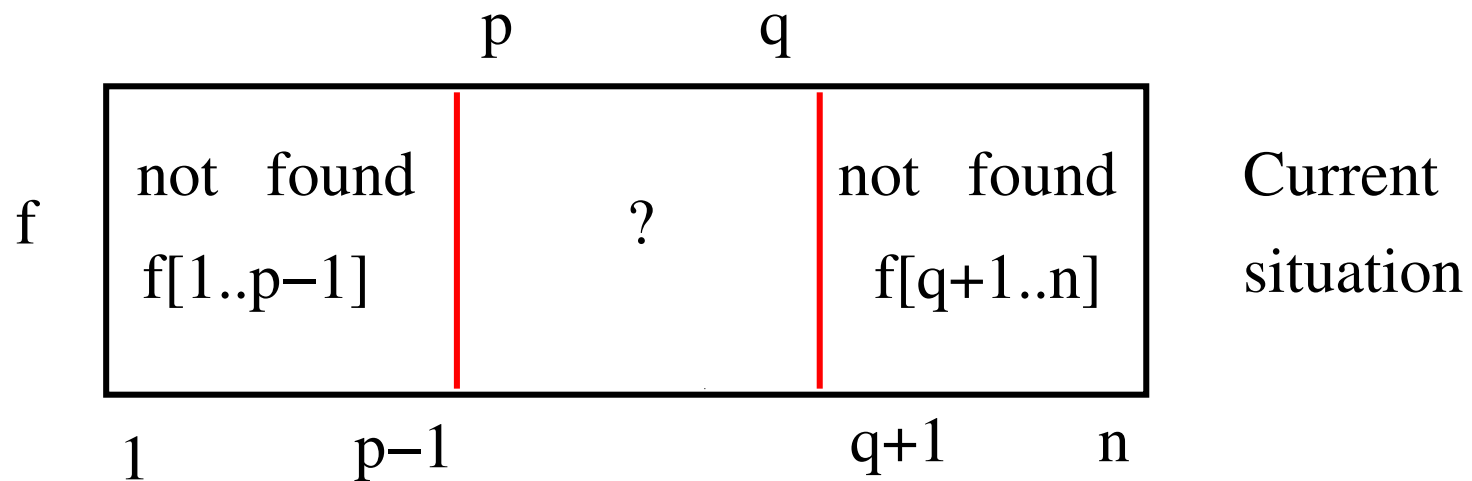
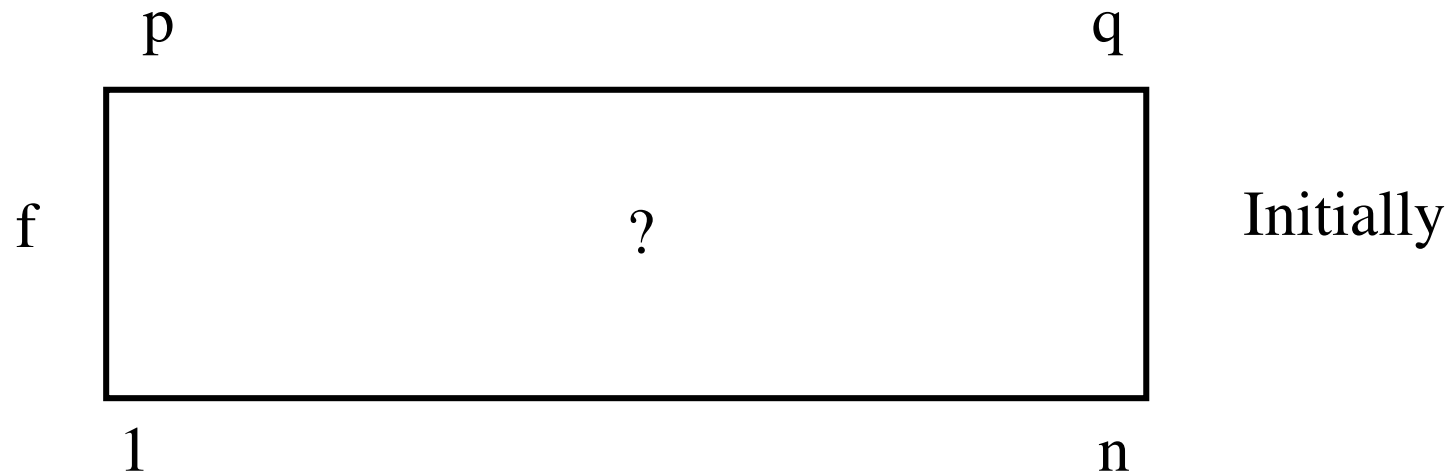
$r \in \mathbb{N}$

bin_search

$r \in f^{-1}[\{v\}]$

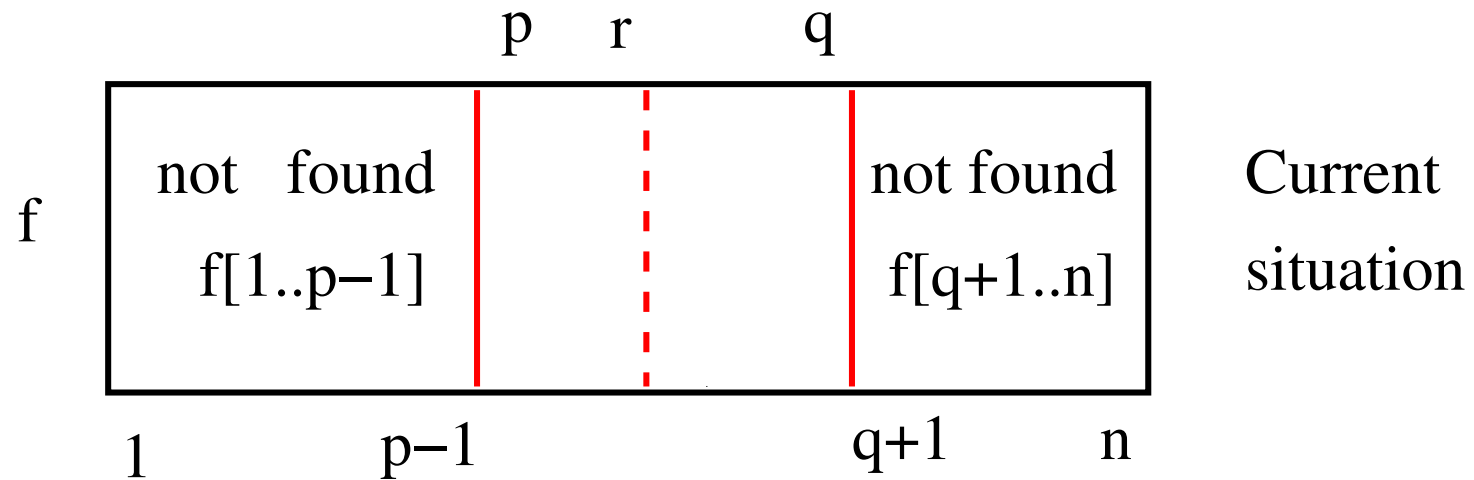
Ideas for a Refinement (1)

- We introduce two new variables p and q

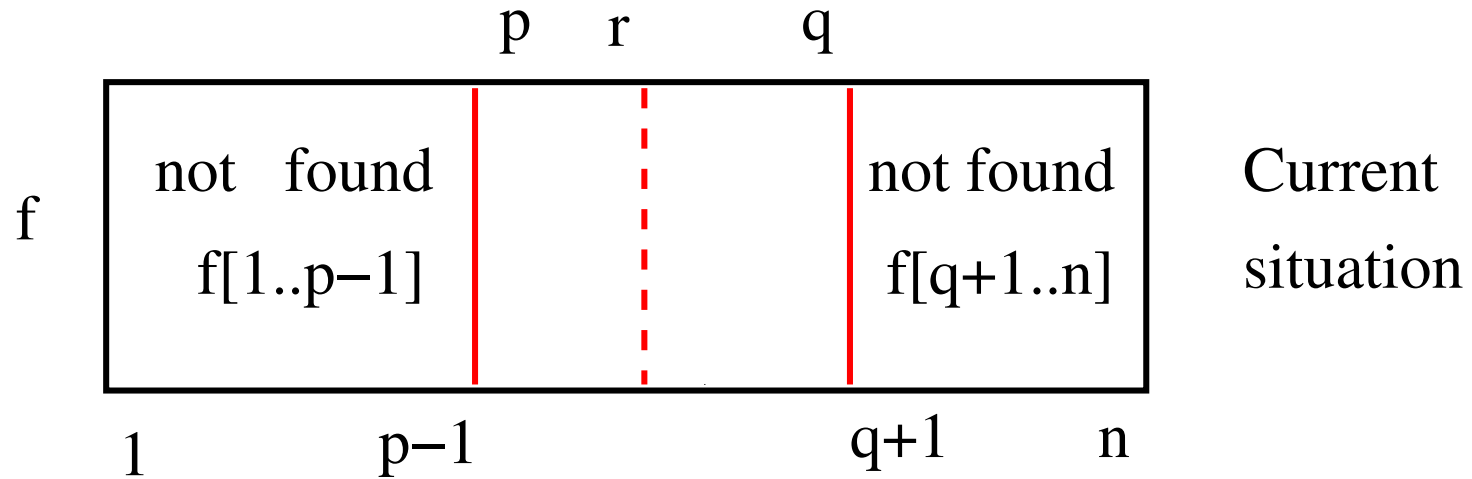


Ideas for a Refinement (2)

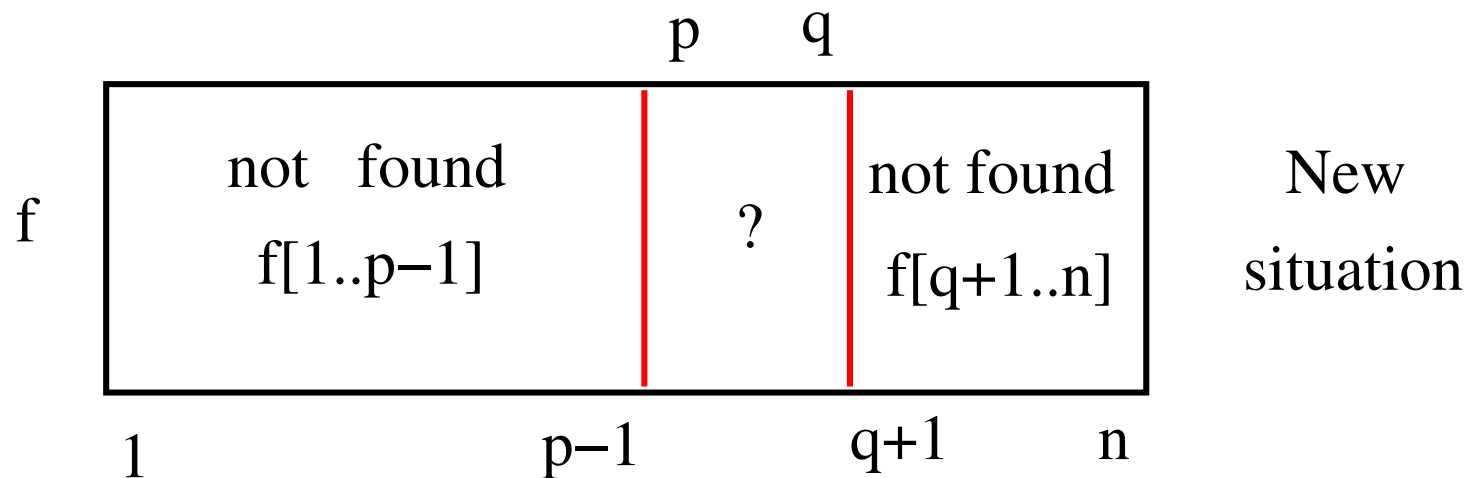
- We choose an arbitrary number r between p and q



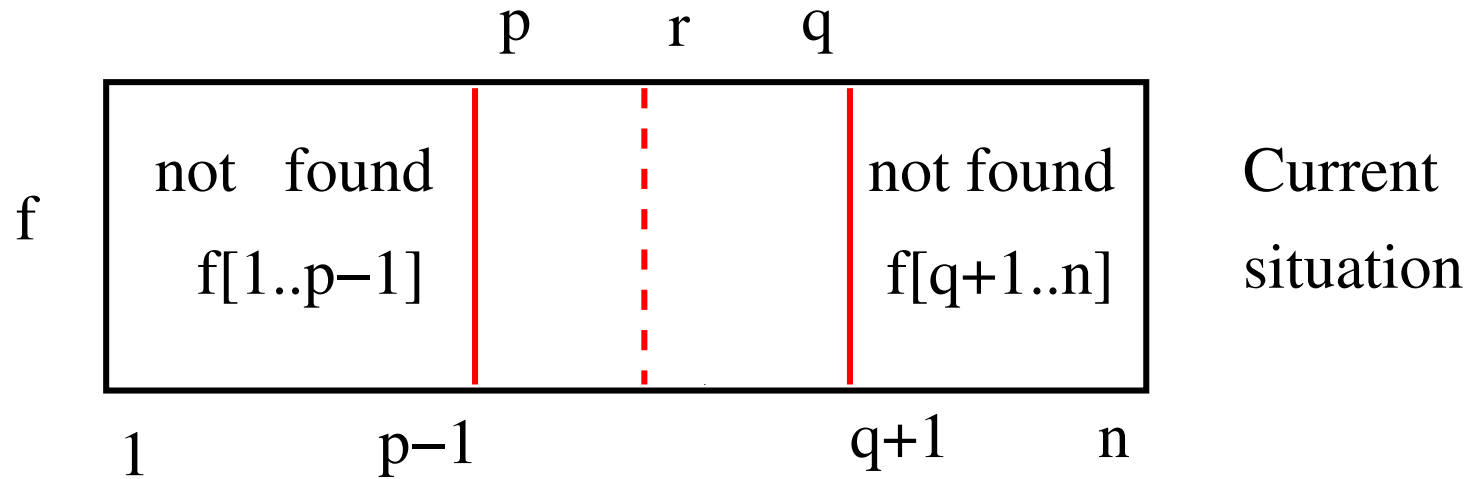
Ideas for a Refinement



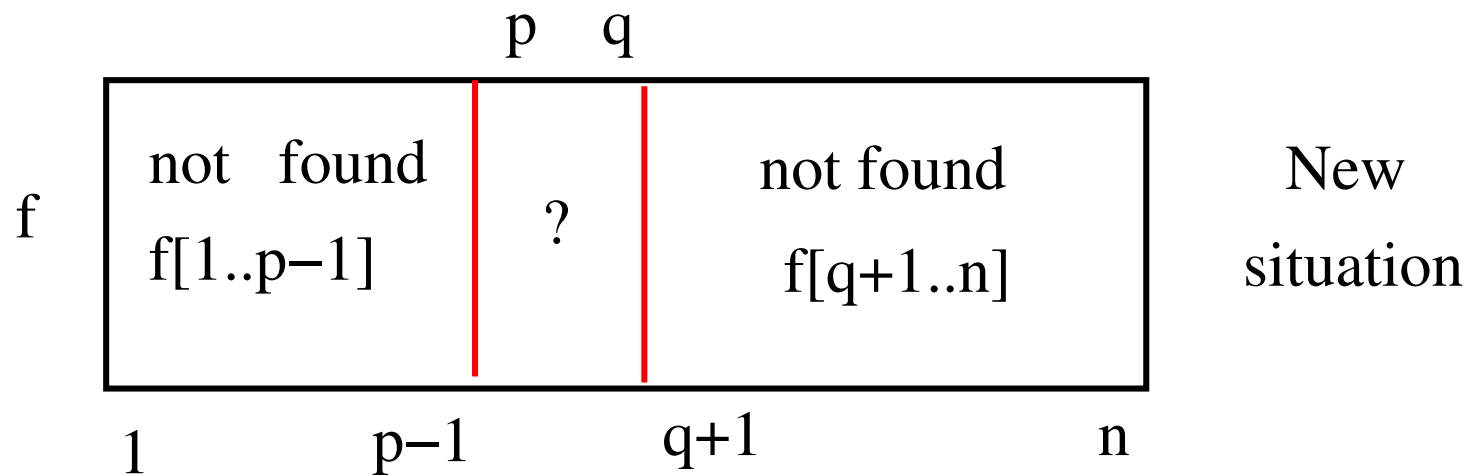
We suppose $f(r) < v$. We can then move p to $r+1$



Ideas for a Refinement



We suppose $v < f(r)$ We can then move q to $r-1$



First Refinement: the State

constants: n, f, v

variables: r, p, q

inv1_1: $p \in 1 .. n$

inv1_2: $q \in 1 .. n$

inv1_3: $p \leq q$

inv1_4: $v \notin f[1 .. p - 1]$

inv1_5: $v \notin f[q + 1 .. n]$

inv1_6: $r \in p .. q$

First Refinement: the Events

init

$p, q := 1, n$

$r \in 1 .. n$

bin_search

when

$f(r) = v$

then

skip

end

inc

when

$f(r) < v$

then

$p := r + 1$

$r \in r + 1 .. q$

end

dec

when

$v < f(r)$

then

$q := r - 1$

$r \in p .. r - 1$

end

Second Refinement

- At the previous stage, **inc** and **dec** were **non-deterministic**
- r was chosen **arbitrarily** within the interval $p .. q$
- We now remove the non-determinacy in **inc** and **dec**
- r is chosen to be the **middle of the interval** $p .. q$

Reducing Non-determinacy

(abstract_)inc

when

$f(r) < v$

then

$p := r + 1$

$r := r + 1 .. q$

end

(concrete_)inc

when

$f(r) < v$

then

$p := r + 1$

$r := (r + 1 + q) / 2$

end

(abstract_)dec

when

$f(r) < v$

then

$q := r - 1$

$r := p .. r - 1$

end

(concrete_)dec

when

$f(r) < v$

then

$q := r - 1$

$r := (p + r - 1) / 2$

end

Second Refinement: the Events

init

$p, q := 1, n$

$r := (1 + n)/2$

bin_search

when

$f(r) = v$

then

skip

end

inc

when

$f(r) \neq v$

$f(r) < v$

then

$p := r + 1$

$r := (r + 1 + q)/2$

end

dec

when

$f(r) \neq v$

$v \leq f(r)$

then

$q := r - 1$

$r := (p + r - 1)/2$

end

Merging Rule M_IF1

when
P
Q
then
S
end

when
P
 $\neg Q$
then
T
end

\rightsquigarrow

when
P
then
 if *Q* **then**
 S
 else
 T
 end
end

M_IF1

Merging Events **inc** and **dec** by means of Rule M_IF1

```
inc
when
   $f(r) \neq v$ 
   $f(r) < v$ 
then
   $p := r + 1$ 
   $r := (r + 1 + q)/2$ 
end
```

```
dec
when
   $f(r) \neq v$ 
   $v \leq f(r)$ 
then
   $q := r - 1$ 
   $r := (p + r - 1)/2$ 
end
```

```
inc_dec
when
   $f(r) \neq v$ 
then
  if  $f(r) < v$  then
     $p, r := r + 1, (r + 1 + q)/2$ 
  else
     $q, r := r - 1, (p + r - 1)/2$ 
  end
end
```

```
bin_search
when
   $f(r) = v$ 
then
  skip
end
```

Merging Rule M_WHILE4

when
 Q
then
 S
end

when
 $\neg Q$
then
 skip
end

\rightsquigarrow

while Q **do**
 S
end

M_WHILE4

Merging Events **inc_dec** and **bin_search** with Rule M_WHILE4

```
inc_dec_bin_search
  while  $f(r) \neq v$  do
    if  $f(r) < v$  then
       $p, r := r + 1, (r + 1 + q)/2$ 
    else
       $q, r := r - 1, (p + r - 1)/2$ 
    end
  end
end
```

```
init
```

```
 $p, q := 1, n$ 
```

```
 $r := (1 + n)/2$ 
```

Merging Events **inc_dec_bin_search** and **init** with Rule M_INIT

```
bin_search_program
   $p, q, r := 1, n, (1 + n)/2;$ 
  while  $f(r) \neq v$  do
    if  $f(r) < v$  then
       $p, r := r + 1, (r + 1 + q)/2$ 
    else
       $q, r := r - 1, (p + r - 1)/2$ 
    end
  end
```

Example 4: Array Sorting

- Given:
 - A numerical array f
- Result is:
 - Another numerical array g
- Such that:
 - g has the **same elements as f**
 - g is sorted in **ascending order**

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 7 | 2 | 5 | 8 | 9 | 4 | 1 |
|---|---|---|---|---|---|---|---|

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Sorting Initial State

constants: n, f

variables: g

prp0_1: $n \in \mathbb{N}$

prp0_2: $0 < n$

prp0_3: $f \in 1 .. n \mapsto \mathbb{N}$

inv0_1: $g \in \mathbb{N} \leftrightarrow \mathbb{N}$

Sorting Initial Events

init

$$g : \in \mathbb{N} \mapsto \mathbb{N}$$

sort

$$g : \left(\begin{array}{l} g' \in 1..n \rightarrow \mathbb{N} \\ \text{ran}(g') = \text{ran}(f) \\ \forall i, j. \left(\begin{array}{l} i \in 1..n-1 \\ j \in i+1..n \end{array} \right) \\ \Rightarrow \\ g'(i) < g'(j) \end{array} \right)$$

Sorting : 1st Refinement

Introducing a new variable k ranging from 1 to n

We also introduce a new variable h which is an injection

Current situation: array h is sorted from 1 to $k - 1$

| | | | | | | |
|---|-------------------|--------|---------|-----|----------|-----|
| 1 | sorted and | \leq | $k - 1$ | k | ? | n |
|---|-------------------|--------|---------|-----|----------|-----|

Invariant

$$\forall i, j \cdot \left(\begin{array}{l} i \in 1 .. k - 1 \\ j \in i + 1 .. n \\ \Rightarrow \\ h(i) \leq h(j) \end{array} \right)$$

Array Sorting First Refinement: the State

constants: n, f

variables: g, h, k

inv1_1: $h \in 1 .. n \mapsto \mathbb{N}$

inv1_2: $\text{ran}(h) = \text{ran}(f)$

inv1_3: $k \in 1 .. n$

inv1_4: $\forall i, j \cdot \left(\begin{array}{l} i \in 1 .. k - 1 \\ j \in i + 1 .. n \\ \Rightarrow \\ h(i) < h(j) \end{array} \right)$

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 7 | 2 | 5 | 8 | 9 | 4 | 1 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 7 | 2 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 7 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 8 | 9 | 4 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 8 | 9 | 5 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 9 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

First Refinement: Events

init

$g, h, k := f, f, 1$

sort

when $k = n$ **then** $g := h$ **end**

progr

when

$k < n$

then

var l **where**

$l \in k .. n$

$h(l) = \min(h[k .. n])$

then

$h := h \leftarrow \{k \mapsto h(l)\} \leftarrow \{l \mapsto h(k)\}$

$k := k + 1$

end

end

Sorting : 2nd Refinement

Introducing **two new variables** j in $k .. n$ and l in $k .. j$

Current situation: $h(l)$ is the minimum of $h[k .. j]$

| | | |
|--------------------------------------|------------------|----------------------|
| 1 sorted and \leq $k - 1$ | k ? j | $j + 1$? n |
|--------------------------------------|------------------|----------------------|

Invariant:

$$\begin{aligned}j &\in k .. n \\l &\in k .. j \\h(l) &= \min (h[k .. j])\end{aligned}$$

Array Sorting Second Refinement: the State

constants: n, f

variables: g, h, k, j, l

inv2_1: $j \in k .. n$

inv2_2: $l \in k .. j$

inv2_3: $h(l) = \min(h[k .. j])$

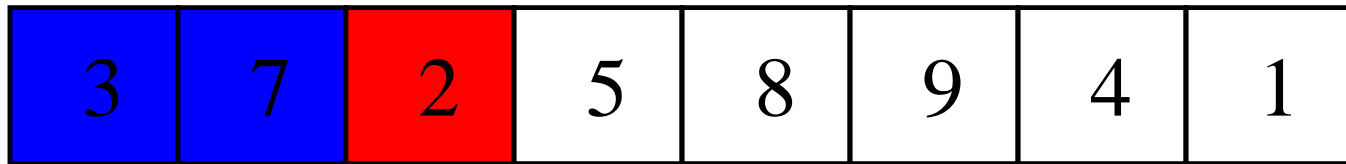
Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 7 | 2 | 5 | 8 | 9 | 4 | 1 |
|---|---|---|---|---|---|---|---|

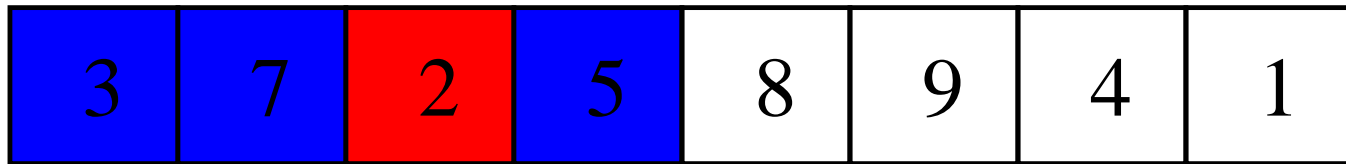
Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 7 | 2 | 5 | 8 | 9 | 4 | 1 |
|---|---|---|---|---|---|---|---|

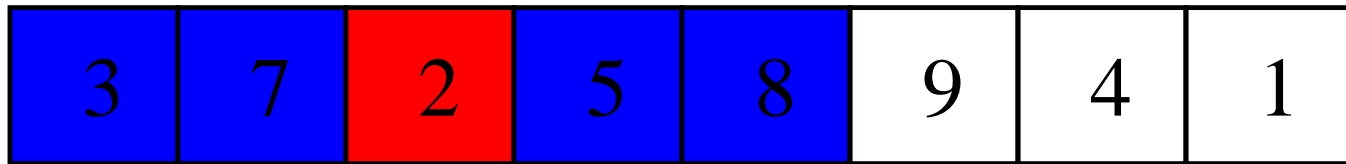
Sorting



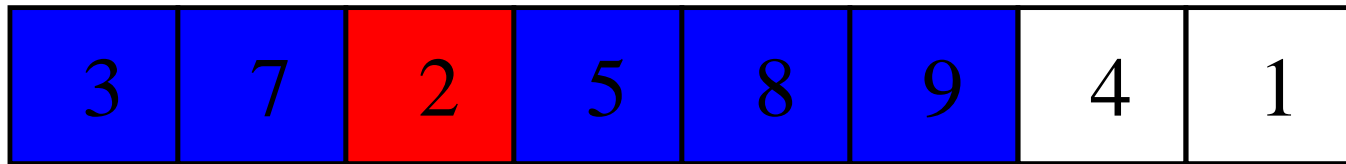
Sorting



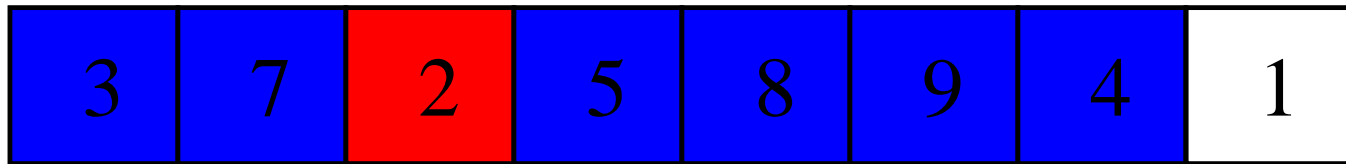
Sorting



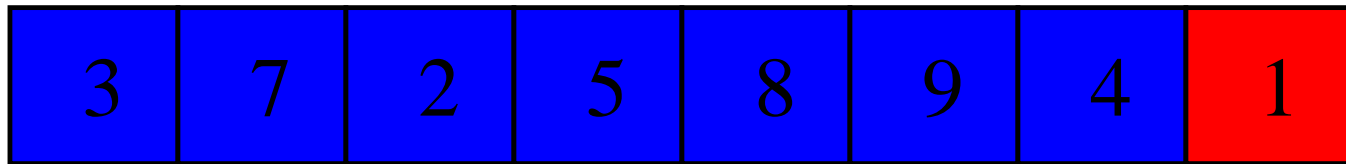
Sorting



Sorting



Sorting



Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 7 | 2 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 7 | 2 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

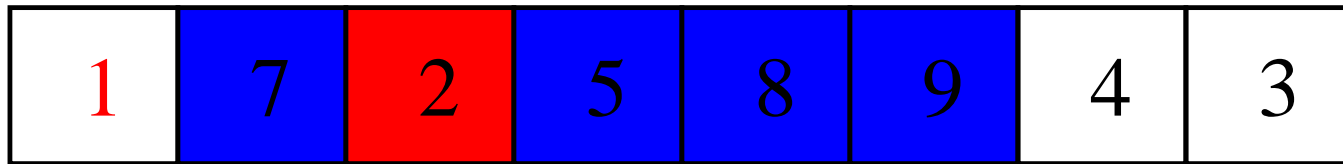
Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 7 | 2 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

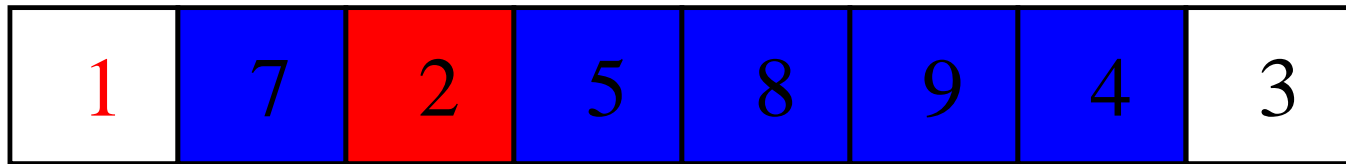
Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 7 | 2 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

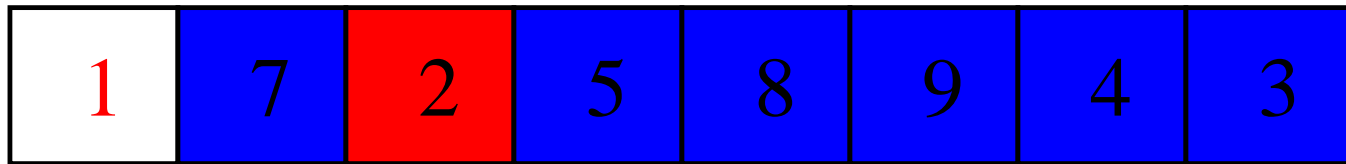
Sorting



Sorting



Sorting



Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 7 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 7 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 7 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 7 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 7 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 7 | 5 | 8 | 9 | 4 | 3 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 8 | 9 | 4 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 8 | 9 | 4 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 8 | 9 | 4 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 8 | 9 | 4 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 8 | 9 | 4 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 8 | 9 | 5 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 8 | 9 | 5 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 8 | 9 | 5 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 8 | 9 | 5 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 9 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 9 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 9 | 8 | 7 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Sorting

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Sorting 2nd Refinement: Refining Existing Events

init

$g, h, k := f, f, 1$
 $j, l := 1, 1$

sort

when

$k = n$

then

skip

end

progr

when

$k < n$

$j = n$

then

$h := h \triangleleft \{k \mapsto h(l)\} \triangleleft \{l \mapsto h(k)\}$

$k, j, l := k + 1, k + 1, k + 1$

end

Sorting 2nd Refinement: Adding New Events

```
prog1
  when
     $k < n$ 
     $j < n$ 
     $h(l) \leq h(j + 1)$ 
  then
     $j := j + 1$ 
  end
```

```
prog2
  when
     $k < n$ 
     $j < n$ 
     $h(l) > h(j + 1)$ 
  then
     $j, l := j + 1, j + 1$ 
  end
```

Sorting: Third Refinement

- We now remove variable g

constants: n, f

variables: h, k, j, l

inv3_1: $g = f \vee g = h$

inv3_2: $k \neq n \Rightarrow g = f$

Sorting 3rd Refinement: Refining Existing Events

```
init
   $h, k := f, 1$ 
   $j, l := 1, 1$ 
```

```
sort
  when
     $k = n$ 
  then
    skip
  end
```

```
progr
  when
     $k < n$ 
     $j = n$ 
  then
     $h := h \triangleleft \{k \mapsto h(l)\} \triangleleft \{l \mapsto h(k)\}$ 
     $k, j, l := k + 1, k + 1, k + 1$ 
  end
```

Sorting 2nd Refinement: Adding New Events

```
prog1
  when
     $k < n$ 
     $j < n$ 
     $h(l) \leq h(j + 1)$ 
  then
     $j := j + 1$ 
  end
```

```
prog2
  when
     $k < n$ 
     $j < n$ 
     $h(j + 1) < h(l)$ 
  then
     $j, l := j + 1, j + 1$ 
  end
```

Sorting: Merging (1)

Applying **M_IF1** to progr1 and progr2

```
progr_12  $\hat{=}$   
  when  
     $k < n$   
     $j < n$   
  then  
    if  $h(l) \leq h(j + 1)$  then  
       $j := j + 1$   
    else  
       $j, l := j + 1, j + 1$   
    end  
  end
```

Merging `progr` and `progr_12`

```
progr ≐  
  when  
     $k < n$   
     $j = n$   
  then  
     $k := k + 1$   
     $j := k + 1$   
     $l := k + 1$   
     $h := \text{swap}(h, k, l)$   
  end
```

```
progr_12 ≐  
  when  
     $k < n$   
     $j < n$   
  then  
    if  $h(l) \leq h(j + 1)$  then  
       $j := j + 1$   
    else  
       $j, l := j + 1, j + 1$   
    end  
  end
```

`inv2_1`: $j \in k .. n$

Sorting : Merging (2)

Applying **Rule M_WHILE_1** to `progr` and `progr_12`

```
progr_progr_12  $\hat{=}$   
  when  
     $k < n$   
  then  
    while  $j < n$  do  
      if  $h(l) \leq h(j + 1)$  then  
         $j := j + 1$   
      else  
         $j, l := j + 1, j + 1$   
      end  
    end;  
     $k, j, l, h := k + 1, k + 1, k + 1, \text{swap}(g, k, l)$   
  end
```

Merging sort and `progr_progr_12`

```
sort  $\hat{=}$   
when  
   $k = n$   
then  
  skip  
end
```

```
progr_progr_12  $\hat{=}$   
  when  
     $k < n$   
  then  
    while  $j < n$  do  
      if  $g(l) \leq g(j + 1)$  then  
         $j := j + 1$   
      else  
         $j, l := j + 1, j + 1$   
      end  
    end;  
     $k, j, l, g := k + 1, k + 1, k + 1, \text{swap}(g, k, l)$   
  end
```

`inv1_3:` $k \in 1 .. n$

Sorting : Merging (3)

Applying **Rule M_WHILE4** to sort and progr_progr_12

```
sort_progr_progr_12  $\hat{=}$   
  while  $k < n$  do  
    while  $j < n$  do  
      if  $h(l) \leq h(j + 1)$  then  
         $j := j + 1$   
      else  
         $j, l := j + 1, j + 1$   
      end  
    end;  
     $k, j, l, h := k + 1, k + 1, k + 1, \text{swap}(h, k, l)$   
  end
```

Merging **init** and **sort_progr_progr_12**

```
init  
 $h := f$   
 $k := 1$   
 $j := 1$   
 $l := 1$ 
```

```
sort_progr_progr_12  $\hat{=}$   
  while  $k < n$  do  
    while  $j < n$  do  
      if  $h(l) \leq h(j + 1)$  then  
         $j := j + 1$   
      else  
         $j, l := j + 1, j + 1$   
      end  
    end;  
     $k, j, l, h := k + 1, k + 1, k + 1, \text{swap}(h, k, l)$   
  end
```

Final Program: Applying Rule **M_INIT**

```
sort_program  $\hat{=}$   
  begin  
     $h, k, j, l := f, 1, 1, 1$  ;                               init  
    while  $k < n$  do  
      while  $j < n$  do  
        if  $h(l) \leq h(j + 1)$  then  
           $j := j + 1$                                          progr_1  
        else  
           $j, l := j + 1, j + 1$                                progr_2  
        end  
      end;  
       $k, j, l, h := k + 1, k + 1, k + 1, \text{swap}(h, k, l)$    progr  
    end  
  end
```

Sorting: Concluding Remarks

- The overall development requires **28 proofs**.
- Among which **7 were interactive**

Example 5: In Place Reversing of an Array

carrier set: S

constants: n, f

variables: g

prp0_1: $n \in \mathbb{N}$

prp0_2: $0 < n$

prp0_3: $f \in 1 .. n \rightarrow \mathbb{N}$

inv0_1: $g \in \mathbb{N} \leftrightarrow S$

In Place Reversing of an Array: Example

Here is an array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 2 | 5 | 4 | 1 | 9 | 7 | 8 |
|---|---|---|---|---|---|---|---|

Here is the reverse array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 8 | 7 | 9 | 1 | 4 | 5 | 2 | 3 |
|---|---|---|---|---|---|---|---|

An element which was at index i is now at index $8 - i + 1$

In Place Reversing of an Array: Events

init

$$g : \in \mathbb{N} \leftrightarrow S$$

reverse

$$g : \left(\begin{array}{l} g' \in 1 .. n \rightarrow S \\ \forall k \cdot \left(\begin{array}{l} k \in 1 .. n \\ \Rightarrow \\ g'(k) = f(n - k + 1) \end{array} \right) \end{array} \right)$$

In Place Reversing of an Array: Refinement

- We introduce two additional variables i and j , both in $1 \dots n$
- Initially i is equal to 1 and j is equal to n
- Here is the current situation:

| | | | | | | |
|---|-----------------|-----|------------------|-----|-----------------|-----|
| 1 | reversed | i | unchanged | j | reversed | n |
|---|-----------------|-----|------------------|-----|-----------------|-----|

- A new event is going to exchange elements in i and j .

Refinement: the New State

carrier sets: S

constants: n, f

variables: g, i, j

inv1_1: $h \in 1 .. n \rightarrow S$

inv1_2: $i \in 1 .. n$

inv1_3: $j \in 1 .. n$

inv1_4: $i + j = n + 1$

inv1_5: $i \leq j + 1$

Refinement: the Main Invariants

$$\text{inv1_4: } i + j = n + 1$$

$$\text{inv1_5: } i \leq j + 1$$

$$\text{inv1_6: } \forall k \cdot (k \in 1 .. i - 1 \Rightarrow h(k) = f(n - k + 1))$$

$$\text{inv1_7: } \forall k \cdot (k \in i .. j \Rightarrow h(k) = f(k))$$

$$\text{inv1_8: } \forall k \cdot (k \in j + 1 .. n \Rightarrow h(k) = f(n - k + 1))$$

| | | | | | | |
|---|----------|-----|-----------|-----|----------|-----|
| 1 | reversed | i | unchanged | j | reversed | n |
|---|----------|-----|-----------|-----|----------|-----|

Refinement: the Events

init

$i := 1$

$j := n$

$g := f$

$h := f$

reverse

when

$j \leq i$

then

$g := h$

end

swap

when

$i < j$

then

$h := h \Leftarrow \{i \mapsto h(j)\} \Leftarrow \{j \mapsto h(i)\}$

$i, j := i + 1, j - 1$

end

Final Program

- We can perform a **second refinement** in order to **eliminate g**
- This leads to the following final program:

```
reverse_program
   $i, j, g := 1, n, f;$ 
  while  $i < j$  do
     $i, j, g := i + 1, j - 1, \text{swap}(g, i, j)$ 
  end
```

Example 6: Reversing a Linear Chain

- So far, all our examples were dealing with **arrays**.
- This new example deals with **pointers**
- We want to reverse a **linear chain**
- A linear chain is made of **nodes**
- The nodes are pointing to each other by means of **pointers**
- To simplify, the nodes have **no information fields**

A Linear Chain

- Here is a linear chain:



- The first node of the chain is denoted by f
- The last node is a special node denoted by l
- We suppose that f and l are distinct
- The nodes of the chain are taken in a set S

Formalizing the Linear Chain

The chain is represented by a **partial injection** c

carrier set: S

constants: f, l, c

variables: r

prp0_1: $\text{finite}(S)$

prp0_2: $f \in S$

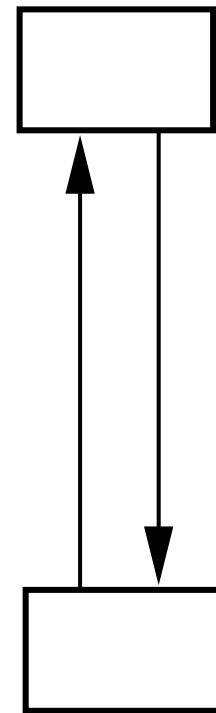
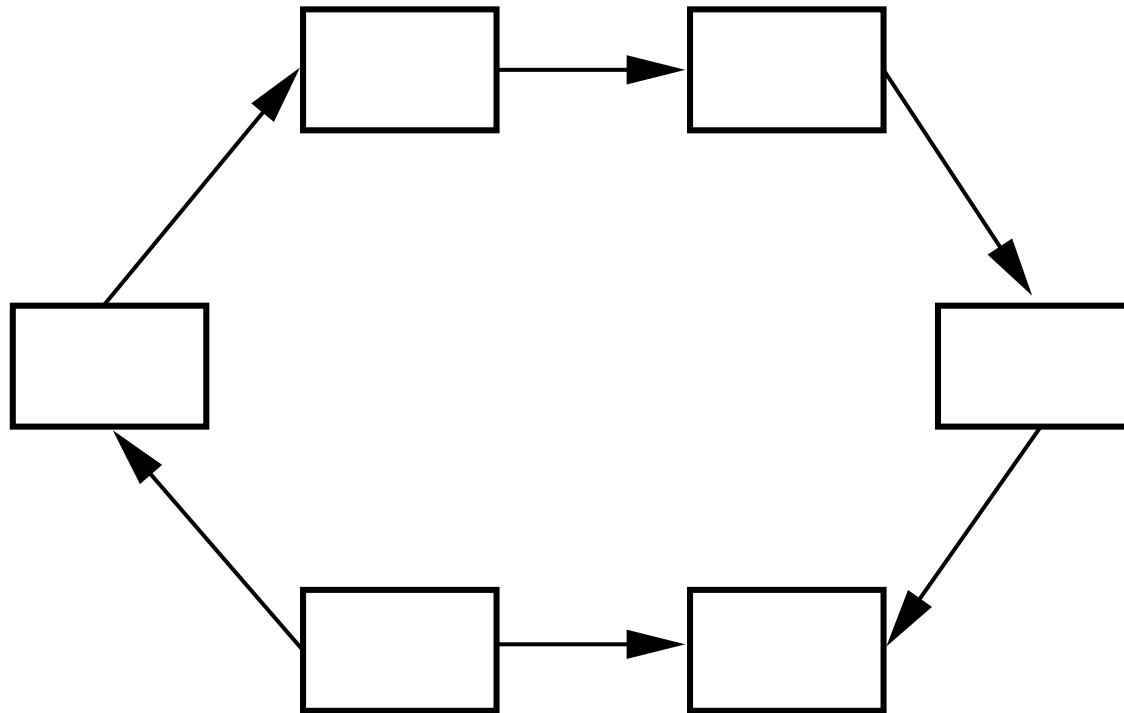
prp0_3: $l \in S$

prp0_4: $f \neq l$

prp0_5: $c \in S \setminus \{l\} \rightsquigarrow S \setminus \{f\}$

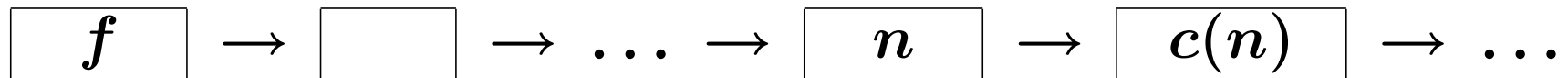
prp0_6: $\text{dom}(c) \cup \{l\} = \text{ran}(c) \cup \{f\}$

An injection is not Sufficient: we may have Cycles



- By following pointers from f to l we must cover all nodes

Characterizing a set of nodes T “departing” from l



- f belongs to T
- if n belongs to T , then $c(n)$ (if it exists) also belongs to T
- Hence we have

$$f \in T$$

$$c[T] \subseteq T$$

- We must state that the set T is exactly the set $\text{ran}(c) \cup \{f\}$

Characterizing a Linear Chain

$$\text{prp0_7: } \forall T \cdot \left(\begin{array}{l} T \subseteq \text{ran}(c) \cup \{f\} \\ f \in T \\ c[T] \subseteq T \\ \Rightarrow \\ \text{ran}(c) \cup \{f\} \subseteq T \end{array} \right)$$

Reversing the Chain

- Given the following initial chain



- Then the transformed chain should look like this:



Initial Model: the Events

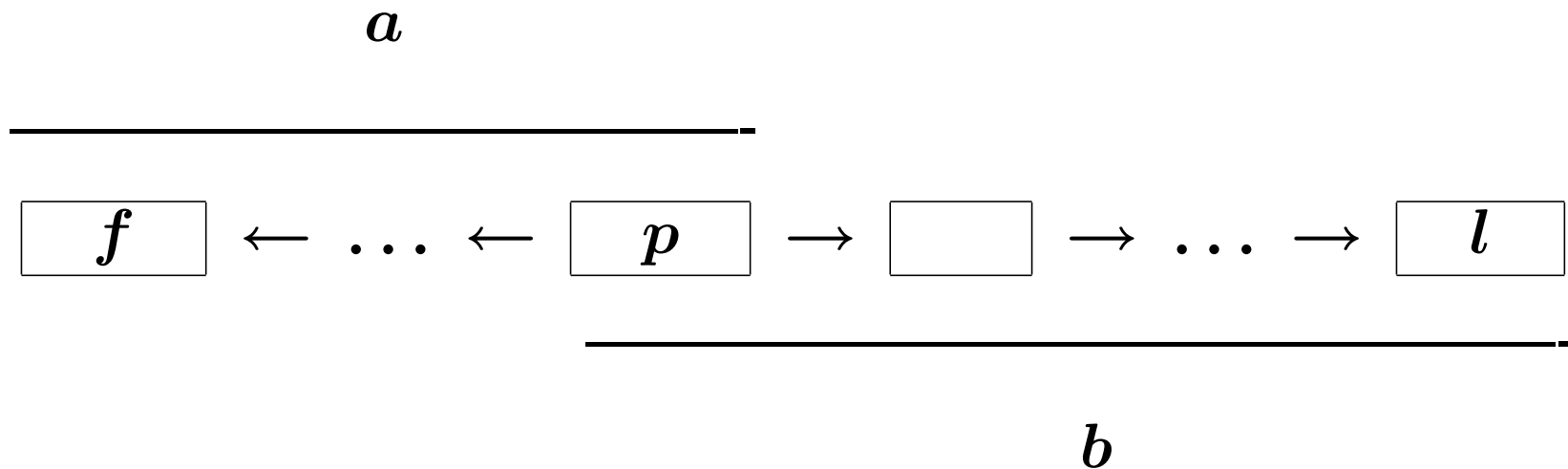
inv0_1: $r \in S \leftrightarrow S$

init
 $r := S \leftrightarrow S$

reverse
 $r := c^{-1}$

First Refinement

We introduce two additional chains a and b and a pointer p

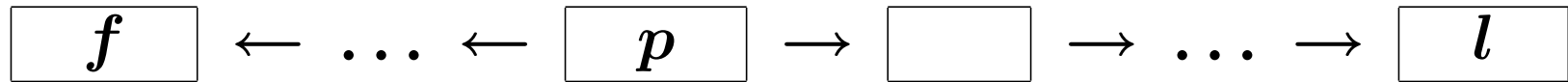


- Node p starts both chains

- Main invariant: $a \cup b^{-1} = c^{-1}$

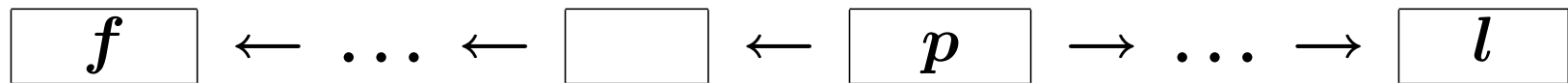
Progressing

a



b

a



b

First Refinement: the State (1)

carrier set: S

constants: f, l, c

variables: r, a, b, p

inv1_1: $p \in S$

inv1_2: $a \in S \setminus \{f\} \leftrightarrow S \setminus \{p\}$

inv1_3: $b \in S \setminus \{l\} \leftrightarrow S \setminus \{p\}$

inv1_4: $c^{-1} = a \cup b^{-1}$

inv1_5: $\text{ran}(b) \cap \text{dom}(a) = \emptyset$

inv1_6: $p \in \text{dom}(a) \cup \text{dom}(b)$

inv1_7: $\text{dom}(a) \cap \text{dom}(b) \subseteq \{p\}$

First Refinement: the State (2)

inv1_8: $b \neq \emptyset \Rightarrow p \in \text{dom}(b)$

inv1_9: $b \neq \emptyset \Rightarrow l \in \text{ran}(b)$

inv1_10: $a \neq \emptyset \Rightarrow p \in \text{dom}(a)$

inv1_11: $a \neq \emptyset \Rightarrow f \in \text{ran}(a)$

inv1_12: $b = \emptyset \Rightarrow p = l$

inv1_13: $a = \emptyset \Rightarrow p = f$

First Refinement: the State (3)

The variable b is a linear chain

$$\mathbf{inv1_13:} \quad \forall T . \left(\begin{array}{l} T \subseteq \text{ran}(b) \cup \{p\} \\ p \in T \\ b[T] \subseteq T \\ \Rightarrow \\ \text{ran}(b) \cup \{p\} \subseteq T \end{array} \right)$$

First Refinement: the Events

```
progress
  when
     $p \in \text{dom}(b)$ 
  then
     $p := b(p)$ 
     $a(b(p)) := p$ 
     $b := \{p\} \triangleleft b$ 
  end
```

```
reverse
  when
     $b = \emptyset$ 
  then
     $r := a$ 
  end
```

```
init
   $r := S \leftrightarrow S$ 
   $a, b, p := \emptyset, c, f$ 
```

Second Refinement: the State

- We introduce a new constant nil
- We replace the chain b by the chain bn
- And we introduce a new pointer q

carrier set: S

constants: f, l, c, nil

variables: r, a, bn, p, q

prp2_1: $nil \in S$

prp2_2: $nil \notin \text{dom}(c)$

prp2_3: $nil \notin \text{ran}(c)$

inv2_1: $bn = b \cup \{l \mapsto nil\}$

inv2_2: $q = bn(p)$

Second Refinement: the Events

```
progress
  when
     $q \neq nil$ 
  then
     $p := q$ 
     $a(q) := p$ 
     $q := bn(q)$ 
     $bn := \{p\} \triangleleft bn$ 
  end
```

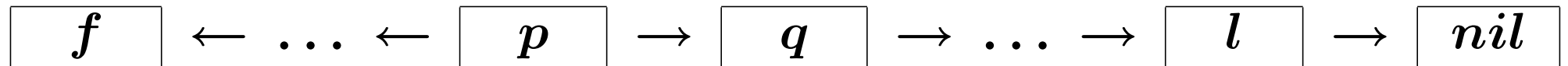
```
reverse
  when
     $q = nil$ 
  then
     $r := a$ 
  end
```

```
init
   $r := \in S \leftrightarrow S$ 
   $a, bn := \emptyset, c \cup \{l \mapsto nil\}$ 
   $p, q := f, c(f)$ 
```

Third Refinement

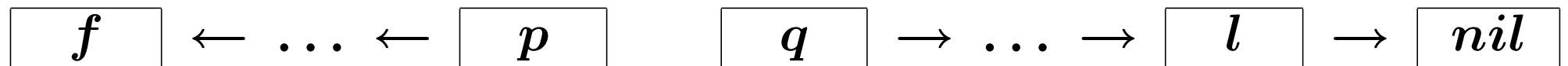
- The previous situation with two chains a and bn

a



bn

- The new situation with a single chain d



d

Third Refinement: the State

carrier set: S

constants: f, l, c

variables: r, p, q, d

inv3_1: $d \in S \leftrightarrow S$

inv3_2: $d = (\{f\} \triangleleft bn) \triangleleft a$

Third Refinement: the Events

```
progress
  when
     $q \neq nil$ 
  then
     $p := q$ 
     $d(q) := p$ 
     $q := d(q)$ 
  end
```

```
reverse
  when
     $q = nil$ 
  then
     $r := d \triangleright \{nil\}$ 
  end
```

```
init
   $r := \in S \leftrightarrow S$ 
   $d := \{f\} \triangleleft (c \cup \{l \mapsto nil\})$ 
   $p, q := f, c(f)$ 
```

Merging

reverse_program

$p, q, d := f, c(f), \{f\} \triangleleft (c \cup \{l \mapsto nil\});$

while $q \neq nil$ **do**

$p := q$

$d(q) := p$

$q := d(q)$

end;

$r := d \triangleright \{nil\}$

Example 7: Integer Square root

- The squaring function is defined on all natural numbers
- And it is injective
- Therefore the inverse function, the square root function, exists
- But it is not defined for all natural number
- We want to make it total

Integer Square Root

- The integer square root of n by defect is a number r such that

$$r^2 \leq n < (r + 1)^2$$

Integer Square Root (cont'd)

- The integer square root of 17, is 4 since we have

$$4^2 \leq 17 < 5^2$$

- The integer square root of 16, is 4 since we have

$$4^2 \leq 16 < 5^2$$

- The integer square root of 15, is 3 since we have

$$3^2 \leq 15 < 4^2$$

Integer Square Root: Initial State and Events

constants: n

variables: r

prp0_1: $n \in \mathbb{N}$

inv0_1: $r \in \mathbb{N}$

init

$r \in \mathbb{N}$

square_root

$r \text{ :| } \left(\begin{array}{l} r' \in \mathbb{N} \\ r'^2 \leq n < (r' + 1)^2 \end{array} \right)$

First Refinement

constants: n

variables: r

inv1_1: $s^2 \leq n$

init

$r, s := 0, 0$

square_root

when

$n < (s + 1)^2$

then

$r := s$

end

progress

when

$(s + 1)^2 \leq n$

then

$s := s + 1$

end

Program after First Refinement

After eliminating r as usual, we obtain the following program:

```
square_root_program
   $s := 0;$ 
  while  $(s + 1)^2 \leq n$  do
     $s := s + 1$ 
  end
```

Second Refinement

- We do not want to compute $(s + 1)^2$ at each step
- We observe the following

$$((s + 1) + 1)^2 = (s + 1)^2 + (2s + 3)$$

$$2(s + 1) + 3 = (2s + 3) + 2$$

- We introduce two numbers a and b such that

$$a = (s + 1)^2$$

$$b = 2s + 3$$

Second Refinement: State and Events

constants: n

variables: r, s, a, b

inv2_1: $a = (r + 1)^2$

inv2_2: $b = 2r + 3$

init

$r := 0$

$s := 0$

$a := 1$

$b := 3$

square_root

when

$n < a$

then

$r := s$

end

progress

when

$a \leq n$

then

$s := s + 1$

$a := a + b$

$b := b + 2$

end

Program after Second Refinement

After eliminating r , we obtain the following program:

```
square_root_program
   $s, a, b := 0, 1, 3;$ 
  while  $a \leq n$  do
     $s, a, b := s + 1, a + b, b + 2$ 
  end
```

Example 8: Inverse of an Injective Numerical Function

- Same problem as in previous example but more general
- We are given a total numerical function f
- The function f is supposed to be strictly increasing
- Hence it is injective
- We want to compute its inverse by defect
- We shall borrow ideas from the binary search development

Inverse of an Injective Numerical Function: the State

constants: f, n

variables: r

inv0_1: $r \in \mathbb{N}$

prp0_1: $f \in \mathbb{N} \rightarrow \mathbb{N}$

prp0_2: $\forall i, j \cdot \left(\begin{array}{l} i \in \mathbb{N} \\ j \in \mathbb{N} \\ i < j \\ \Rightarrow \\ f(i) < f(j) \end{array} \right)$

prp0_3: $n \in \mathbb{N}$

thm0_1: $f \in \mathbb{N} \mapsto \mathbb{N}$

Inverse of an Injective Numerical Function: the Events

init

$r \in \mathbb{N}$

inverse

$$r : \mid \left(\begin{array}{l} r' \in \mathbb{N} \\ f(r') \leq n < f(r' + 1) \end{array} \right)$$

First Refinement

- We are supposedly given two constant numbers a and b such that

$$f(a) \leq n < f(b + 1)$$

- We are thus certain that our result is within the interval $a .. b$
- We try to make this interval **narrower**
- We introduce two constants p and q in $a .. b$ and such that

$$f(p) \leq n < f(q + 1)$$

First Refinement: the State (1)

constants: f, n, a, b

variables: r, p, q

prp1_1: $a \in \mathbb{N}$

prp1_2: $b \in \mathbb{N}$

prp1_3: $f(a) \leq n$

prp1_4: $n < f(b + 1)$

First Refinement: the State (2)

$$\mathbf{inv1_1:} \quad p \in \mathbb{N}$$

$$\mathbf{inv1_2:} \quad q \in \mathbb{N}$$

$$\mathbf{inv1_3:} \quad p \leq q$$

$$\mathbf{inv1_4:} \quad f(p) \leq n$$

$$\mathbf{inv1_5:} \quad n < f(q + 1)$$

First Refinement: the Events (1)

init

$r \in \mathbb{N}$

$p, q := a, b$

inverse

when

$p = q$

then

$r := p$

end

First Refinement: the Events (2)

dec

when

$p \neq q$

then

var x **where**

$x \in p + 1 .. q$

$n < f(x)$

then

$q := x - 1$

end

end

inc

when

$p \neq q$

then

var x **where**

$x \in p + 1 .. q$

$f(x) \leq n$

then

$p := x$

end

end

Local Variables

- The construct:

```
var  $x$  where  
     $C(x, c, v)$   
then  
     $x :| P(x, c, v, v')$   
end
```

- is a shorthand for:

$$x :| \exists x \cdot (C(x, c, v) \wedge P(x, c, v, v'))$$

What we Have to Prove

- Event **init** refines its abstraction
- Event **inverse** refines its abstraction
- Events **inc** and **dec** refine skip
- Events **inc** and **dec** decrease a variant
- The system is deadlock-free

Second Refinement: the Events

- We reduce the non-determinacy

```
dec
  when
     $p \neq q$ 
     $n < f((p + 1 + q)/2)$ 
  then
     $q := (p + 1 + q)/2 - 1$ 
  end
```

```
inc
  when
     $p \neq q$ 
     $f((p + 1 + q)/2) \leq n$ 
  then
     $p := (p + 1 + q)/2$ 
  end
```

Final Program

```
inverse_program
   $p, q := a, b;$ 
  while  $p \neq q$  do
    if  $n < f((p + 1 + q)/2)$  then
       $q := (p + 1 + q)/2 - 1$ 
    else
       $p := (p + 1 + q)/2$ 
    end
  end;
   $r := p$ 
```

Genericity

- The development made in this example is **generic**
- We can consider that the constants f , a , and b are **parameters**
- **By instantiating them** we obtain some new programs **almost for free**
- But we have to **prove the properties** of the instantiated constants:

In our case we have to prove:

- **prp0_1**: f is a total function
- **prp0_2**: f is increasing
- **prp1_3** and **prp1_4**: $f(a) \leq n < f(b + 1)$

First Instantiation (1)

- f is instantiated to the squaring function
- a and b are instantiated to 0 and n since we have

$$0^2 \leq n < (n + 1)^2$$

- We shall obtain an **integer square root** program

First Instantiation (2)

```
square_root_program
   $p, q := 0, n;$ 
  while  $p \neq q$  do
    if  $n < ((p + 1 + q)/2)^2$  then
       $q := (p + 1 + q)/2 - 1$ 
    else
       $p := (p + 1 + q)/2$ 
    end
  end;
   $r := p$ 
```

Second Instantiation (1)

- f is instantiated to the function which “multiply by m ”
- a and b are instantiated to 0 and n since we have

$$m \times 0 \leq n < m \times (n + 1)$$

- We shall obtain an **integer division** program: n/m

Second Instantiation (2)

```
integer_division_program
   $p, q := 0, n;$ 
  while  $p \neq q$  do
    if  $n < m \times (p + 1 + q)/2$  then
       $q := (p + 1 + q)/2 - 1$ 
    else
       $p := (p + 1 + q)/2$ 
    end
  end;
   $r := p$ 
```