15. Sequential Program Development

Jean-Raymond Abrial

2009

Purpose of this Lecture

- To present a formal approach for developing sequential programs

- To present a large number of examples:
    - array programs
    - pointer programs
    - numerical programs

- 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 (**;**)

```
while  j ≠ 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, swap (g, k + 1, j + 1)
   end
end ;
p := k
```

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

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

if $condition$ **then** $statement$ **elsif** $\ldots$ **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 ≠ 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, swap (g, k + 1, j + 1)
   end
end ;
p := k
```

```
when
   j ≠ m
   g(j + 1) > x
then
   j := j + 1
end
```

```
while  j ≠ 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, swap (g, k + 1, j + 1)
  end
end ;
p := k
```

```
when
  j ≠ m
  g(j + 1) ≤ x
  k = j
then
  k, j := k + 1, j + 1
end
```

```
while  j ≠ 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, swap (g, k + 1, j + 1)
  end
end ;
p := k
```

```
when
  j ≠ m
  g(j + 1) ≤ x
  k ≠ j
then
  k, j, g := k + 1, j + 1, swap (g, k + 1, j + 1)
end
```

```
while  j ≠ 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, swap (g, k + 1, j + 1)
   end
end ;
p := k
```

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

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
$k, j, g := \ldots$
end

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

Composing a Program from Events

- We have just decomposed a program into separate events

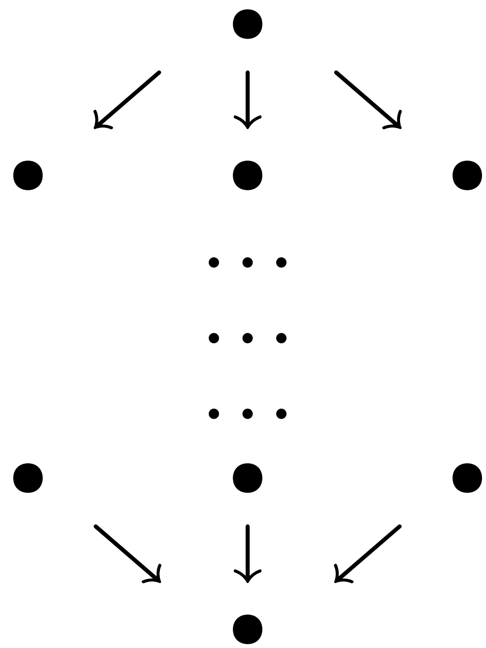
- Our approach will consists in doing the reverse operation

- We shall construct the events first

- And then compose our program from these events

Specification Phase

initial event: Specification



Design Phase

new events: Refinements

Merging Phase

final event: Program

- Sequential Programs are usually specified by means of:
  - A pre-condition
  - and a post-condition

- It is represented with a Hoare-triple

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

- We are given (Pre-condition)

- 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: $n \in \mathbb{N}$
  - n is positive: $0 < n$

Example 1: The search Program

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: $0 < n$
  - an array f of n elements built on a set S: $f \in 1\,..\,n \rightarrow S$

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: $0 < n$
  - 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: $v \in \mathrm{ran}(f)$

Example 1: The search Program

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: $0 < n$
  - 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: $v \in \mathrm{ran}(f)$

- We are looking for (Post-condition)

Example 1: The search Program

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: $0 < n$
  - 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: $v \in \operatorname{ran}(f)$

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

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: $0 < n$
  - 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: $v \in \operatorname{ran}(f)$

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

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: $0 < n$
  - 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: $v \in \mathrm{ran}(f)$

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

$$\left\{\begin{array}{l} n \in \mathbb{N} \\ 0 < n \\ f \in 1\,..\,n \rightarrow S \\ v \in \mathrm{ran}(f) \end{array}\right\} \quad \textbf{search} \quad \left\{\begin{array}{l} r \in \mathrm{dom}(f) \\ f(r) = v \end{array}\right\}$$

Encoding a Hoare-triple in an Event System

- Input parameters are constants

- The pre-condition corresponds to axioms of these constants

- Output parameters are variables

- The post-condition is in the guard of a unique event

- [When developing several programs in the same module,

- input parameters can also be variables of a special "init" event]

Encoding a Hoare-triple in an Event System

$$\left\{ \begin{array}{l} n \in \mathbb{N} \\ 0 < n \\ f \in 1\,..\,n \rightarrow S \\ v \in \mathrm{ran}(f) \end{array} \right\} \quad \textsf{search} \quad \left\{ \begin{array}{l} r \in \mathrm{dom}(f) \\ f(r) = v \end{array} \right\}$$

Encoding a Hoare-triple in an Event System

$$\left\{\begin{array}{l} n \in \mathbb{N} \\ 0 < n \\ f \in 1\,..\,n \rightarrow S \\ v \in \operatorname{ran}(f) \end{array}\right\} \quad \textbf{search} \quad \left\{\begin{array}{l} r \in \operatorname{dom}(f) \\ f(r) = v \end{array}\right\}$$

carrier sets: S

constants: n, f, v

variables: r

axm0_1: $n \in \mathbb{N}$

axm0_2: $0 < n$

axm0_3: $f \in 1\,..\,n \rightarrow S$

axm0_4: $v \in \operatorname{ran}(f)$

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

Encoding a Hoare-triple in an Event System

$$\left\{ \begin{array}{l} n \in \mathbb{N} \\ 0 < n \\ f \in 1\,..\,n \rightarrow S \\ v \in \mathrm{ran}(f) \end{array} \right\} \quad \text{search} \quad \left\{ \begin{array}{l} r \in \mathrm{dom}(f) \\ f(r) = v \end{array} \right\}$$

carrier sets: S

constants: n, f, v

variables: r

axm0_1: $n \in \mathbb{N}$

axm0_2: $0 < n$

axm0_3: $f \in 1\,..\,n \rightarrow S$

axm0_4: $v \in \mathrm{ran}(f)$

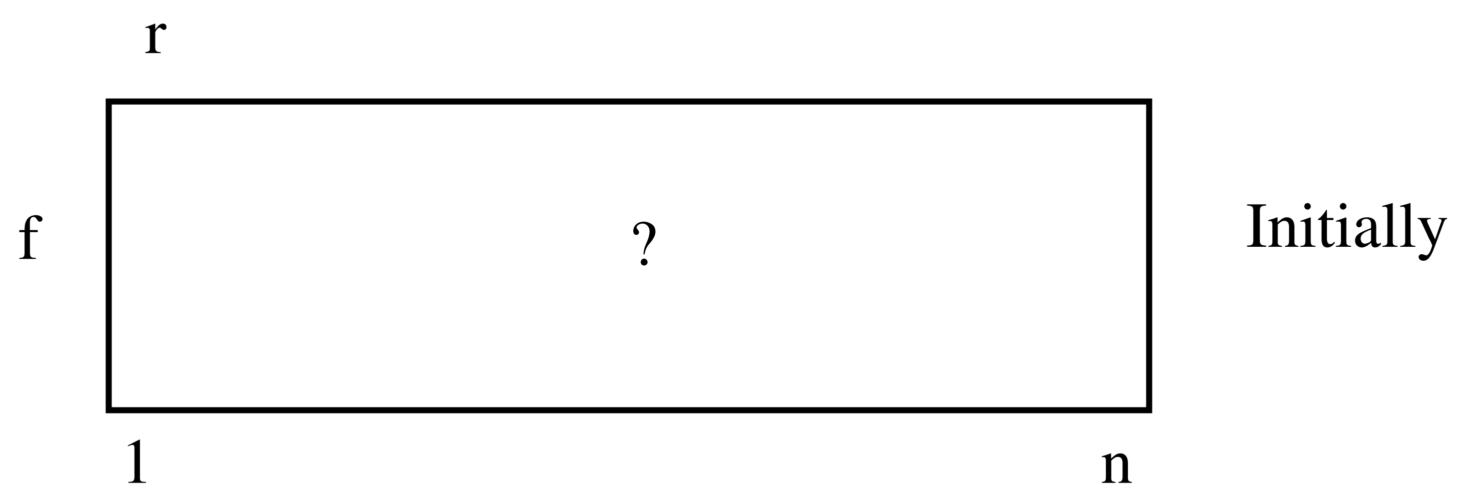
inv0_1: $r \in \mathbb{N}$

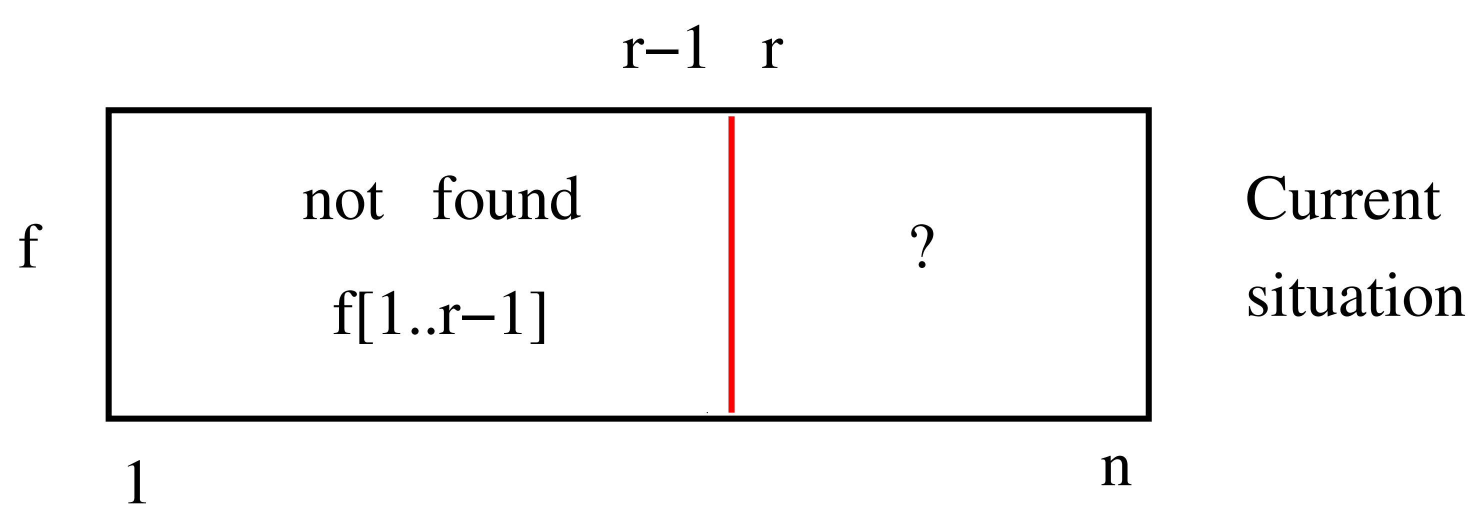
init
  $r :\in \mathbb{N}$

final
  when
    $r \in \mathrm{dom}(f)$
    $f(r) = v$
  then
    skip
  end

progress
  status
    anticipated
  then
    $r :\in \mathbb{N}$
  end

Result variable r is set to 1 initially





inv1_1: $r \in 1\,..\,n$

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

variant1: $n - r$

init
$r := 1$

progress
status
convergent
when
$f(r) \neq v$
then
$r := r + 1$
end

final
when
$f(r) = v$
then
skip
end

To be Proved (as usual)

- Events refine their abstractions

- Events maintain invariants

- The exhibited variant is a natural number

- Event progress decreases the variant

- The system is deadlock free

We are using some Merging Rules to build the final program

```
init
    r := 1
```

```
progress
    when
        f(r) ≠ v
    then
        r := r + 1
    end
```

```
final
    when
        f(r) = v
    then
        skip
    end
```

Merging Rule (1)

when P Q **then** S **end**	**when** P $\neg Q$ **then** T **end**	$\leadsto$	**when** P **then** **while** Q **do** S **end**; T **end**	M_WHILE

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

- Special Case: If P is missing the resulting "event" has no guard

Merging Rule (2)

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_IF

- Side Conditions:

    - The disjunctive negation of the previous side conditions

- Special Case: If P is missing the resulting "event" has no guard

Applying Rule M_WHILE (special case)

```
progress
   when
      f(r) ≠ v
   then
      r := r + 1
   end
```

```
final
   when
      f(r) = v
   then
      skip
   end
```

```
progress_final
   while  f(r) ≠ v  do
      r := r + 1
   end
```

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
  $r := 1$

progress_final
  while $f(r) \neq v$ **do**
    $r := r + 1$
  end

$$\left\{ \begin{array}{l} n \in \mathbb{N} \\ 0 < n \\ f \in 1\,..\,n \rightarrow S \\ v \in \operatorname{ran}(f) \end{array} \right\}$$

search_program
  $r := 1;$
  while $f(r) \neq v$ **do**
    $r := r + 1$
  end

$$\left\{ \begin{array}{l} r \in \operatorname{dom}(f) \\ f(r) = v \end{array} \right\}$$

Example 2: The Very Classical Binary Search

- Almost the same specification as in Example 1

- It will show the usage of more merging rules

- We are given (Pre-condition)

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

- We are given (Pre-condition)
    - a natural number n: $n \in \mathbb{N}$
    - n is positive: 0<n

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: 0<n
  - a sorted array f of n elements built on a set $\mathbb{N}$: $f \in 1..n \rightarrow \mathbb{N}$

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: 0<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: $v \in \mathrm{ran}(f)$

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: 0<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: $v \in \mathrm{ran}(f)$

- We are looking for (Post-condition)

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: 0<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: $v \in \mathrm{ran}(f)$

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

- We are given (Pre-condition)
  - a natural number n: $n \in \mathbb{N}$
  - n is positive: 0<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: $v \in \mathrm{ran}(f)$
- We are looking for (Post-condition)
  - an index r in the domain of the array: $r \in \mathrm{dom}(f)$
  - such that $f(r) = v$

Binary Search: the State

constants: n, f, v

variables: r

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

axm0_1: $n \in \mathbb{N}$

axm0_2: $0 < n$

axm0_3: $f \in 1\,..\,n \rightarrow \mathbb{N}$

axm0_4: $\forall i, j \cdot \begin{pmatrix} i \in 1\,..\,n \\ j \in 1\,..\,n \\ i \leq j \\ \Rightarrow \\ f(i) \leq f(j) \end{pmatrix}$

axm0_5: $v \in \mathrm{ran}(f)$

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

final
  when
    $r \in \mathrm{dom}(f)$
    $f(r) = v$
  then
    skip
  end

progress
  status
    anticipated
  then
    $r :\in \mathbb{N}$
  end

First Refinement: the State

constants: n, f, v

variables: r, p, q

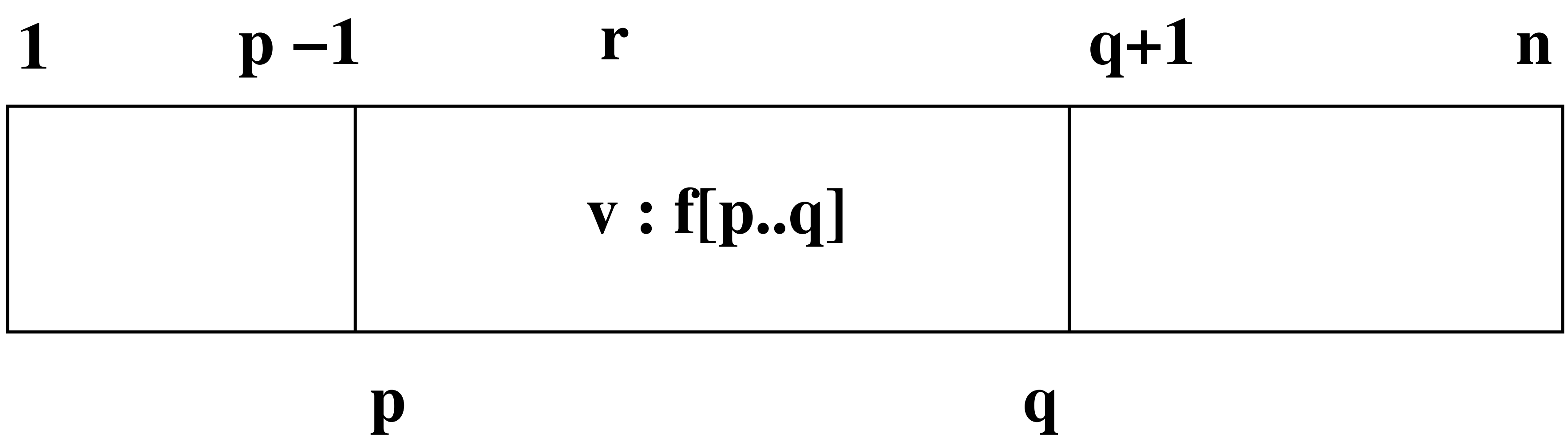
inv1_1: $p \in 1 \,..\, n$

inv1_2: $q \in 1 \,..\, n$

inv1_3: $v \in f[p \,..\, q]$

inv1_4: $r \in p \,..\, q$

- Current situation

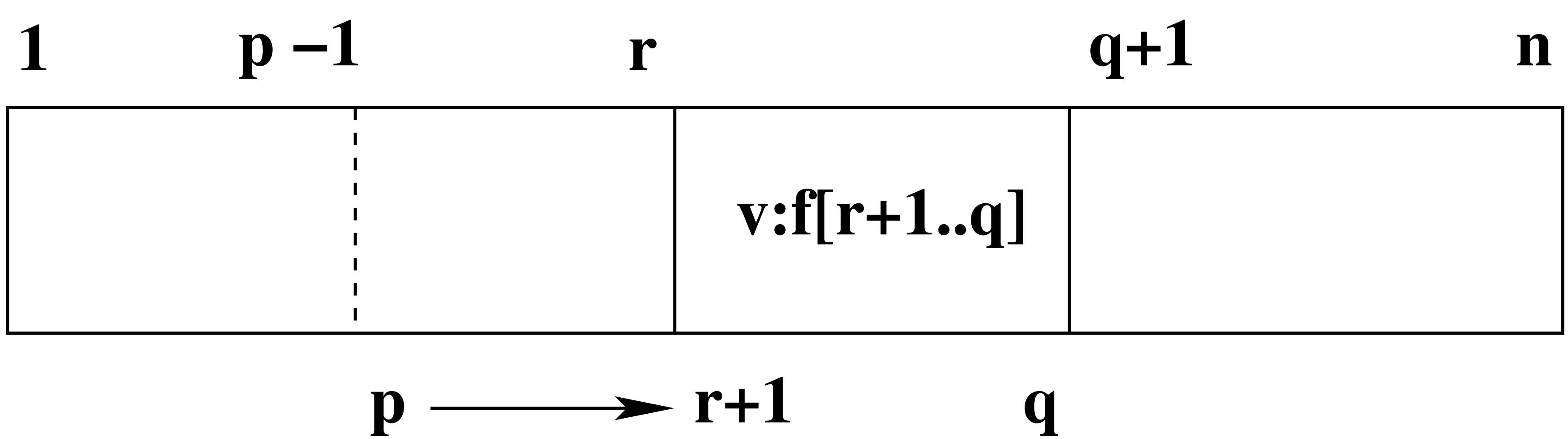


Event inc refining progress

inc
status
convergent
when
$f(r) < v$
then
$p := r + 1$
$r :\in r + 1 \,..\, q$
end

variant1: $q - p$

- Situation encountered by event inc

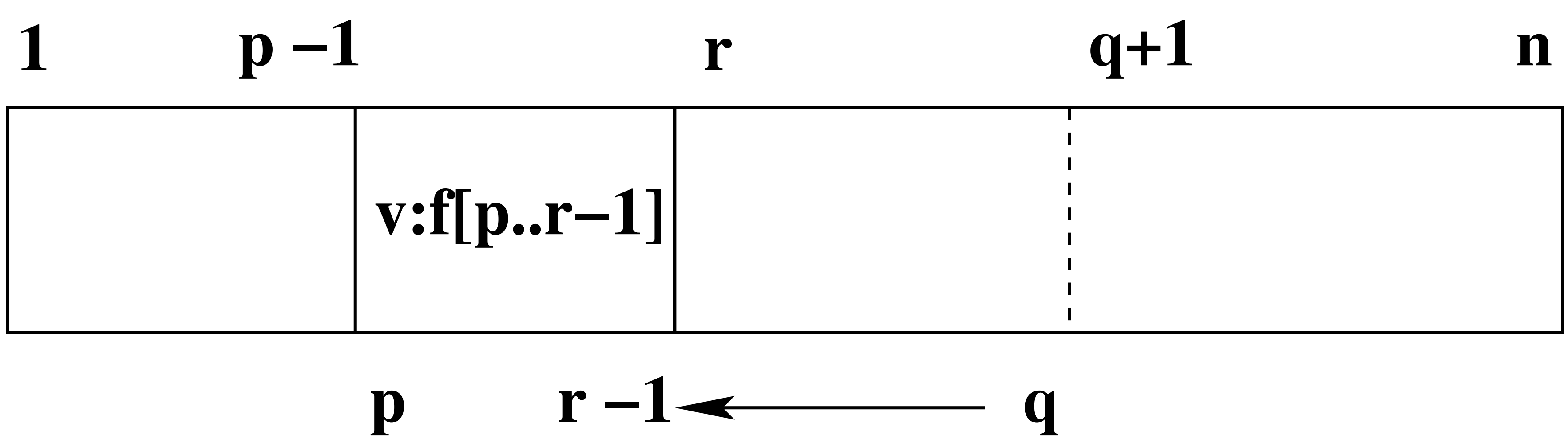


Event dec refining progress

dec
status
convergent
when
$v < f(r)$
then
$q := r - 1$
$r :\in p\,..\,r - 1$
end

variant1: $q - p$

- Situation encountered by event dec



```
init
    p := 1
    q := n
    r :∈ 1 .. n
```

```
final
  when
    f(r) = v
  then
    skip
  end
```

```
inc
  when
    f(r) < v
  then
    p := r + 1
    r :∈ r + 1 .. q
  end
```

```
dec
  when
    v < f(r)
  then
    q := r − 1
    r :∈ p .. r − 1
  end
```

- At the previous stage, inc and dec were non-deterministic

- r was chosen arbitrarily within the interval $p \mathinner{\ldotp\ldotp} q$

- We now remove the non-determinacy in inc and dec

- r is chosen to be the middle of the interval $p \mathinner{\ldotp\ldotp} q$

(abstract_)inc
when
$f(r) < v$
then
$p := r + 1$
$r :\in r + 1 \,..\, q$
end

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

(abstract_)dec
when
$f(r) < v$
then
$q := r - 1$
$r :\in p \,..\, r - 1$
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
    $f(r) < v$
  then
    $p := r + 1$
    $r := (r + 1 + q)/2$
  end

dec
  when
    $v < f(r)$
  then
    $q := r - 1$
    $r := (p + r - 1)/2$
  end

Merging Rule M_IF

when
P
Q
then
S
end
	when
	P
	$\neg\, Q$
	then
	T
	end
		$\leadsto$	**when**
			P
			then
			if Q **then**
			S
			else
			T
			end
			end
				M_IF

Merging Events inc and dec by means of Rule M_IF

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

```
dec
when
   f(r) ≠ v
   v ≤ f(r)
then
   q := r − 1
   r := (p + r − 1)/2
end
```

```
inc_dec
when
   f(r) ≠ 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
```

```
final
   when
      f(r) = v
   then
      skip
   end
```

Merging Rule M_WHILE

when P Q **then** S **end**	**when** P $\neg Q$ **then** T **end**	$\leadsto$	**when** P **then** **while** Q **do** S **end**; T **end**	M_WHILE

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

- Special Case: If P is missing the resulting "event" has no guard

Merging Events inc_dec and bin_search with Rule M_WHILE

```
inc_dec
when
   f(r) ≠ 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
```

```
inc_dec_final
   while  f(r) ≠ 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
```

```
final
   when
      f(r) = v
   then
      skip
   end
```

```
init
   p, q := 1, n
   r := (1 + n)/2
```

```
inc_dec_final
  while f(r) ≠ 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
```

```
init
  p, q := 1, n
  r := (1 + n)/2
```

```
bin_search_program
  p, q, r := 1, n, (1 + n)/2;
  while f(r) ≠ 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 3: Array Partitioning

- Given a numerical array f with n distinct elements

- Given a number x

- We construct another numerical array g with some constraints.

Array Partitioning: More Constraints

- g has the same elements as f

- there exists a number k in $0 \,..\, n$ such that elements of g are:
    - not greater than x in interval $1 \,..\, k$
    - greater than x in interval $k+1 \,..\, n$

1	$\leq x$	k	$k+1$	$> x$	n

Example

- Let the array f be the following:

3	7	2	5	8	9	4	1

- Let x be equal to 5

- The result g can be the following with k being set to 5

3	2	5	4	1	9	7	8

k

Array Partitioning: first special case

- Let the array f be the following:

3	**7**	**2**	**5**	**8**	**9**	**4**	**1**

- Let x be equal to 0

- The result g can be the following with k being set to 0

3	**7**	**2**	**5**	**8**	**9**	**4**	**1**

k

Array Partitioning: second special case

- Let the array f be the following:

$\mathbf{3}$	$\mathbf{7}$	$\mathbf{2}$	$\mathbf{5}$	$\mathbf{8}$	$\mathbf{9}$	$\mathbf{4}$	$\mathbf{1}$

- Let x be equal to 10

- The result g can be the following with k being set to 8

$\mathbf{3}$	$\mathbf{7}$	$\mathbf{2}$	$\mathbf{5}$	$\mathbf{8}$	$\mathbf{9}$	$\mathbf{4}$	$\mathbf{1}$

$\boldsymbol{k}$

Array Partitioning: the Initial State

constants: n, f, x

variables: k, g

axm0_1: $n \in \mathbb{N}$

axm0_2: $f \in 1\,..\,n \rightarrowtail \mathbb{N}$

axm0_3: $x \in \mathbb{N}$

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

inv0_2: $g \in \mathbb{N} \leftrightarrow \mathbb{N}$

Array Partitioning: the Initial Events

init
$k :\in \mathbb{N}$
$g :\in \mathbb{N} \leftrightarrow \mathbb{N}$

final
when
$k \in 0 \,..\, n$
$g \in 1 \,..\, n \rightarrowtail \mathbb{N}$
$\mathrm{ran}\,(g) = \mathrm{ran}\,(f)$
$\forall l \cdot l \in 1 \,..\, k \;\Rightarrow\; g(l) \leq x$
$\forall l \cdot l \in k+1 \,..\, n \;\Rightarrow\; g(l) > x$
then
skip
end

progress
status
anticipated
then
$k :\in \mathbb{N}$
$g :\in \mathbb{N} \leftrightarrow \mathbb{N}$
end

Introducing a new variable j ranging from 0 to n

Current situation: array g is partitioned from 1 to j

1	$\leq x$	k	$k+1$	$> x$	j	$j+1$	$?$	n

Invariant

$$k \leq j$$

$$\forall l \cdot l \in 1 \,..\, k \;\Rightarrow\; g(l) \leq x$$

$$\forall l \cdot l \in k+1 \,..\, j \;\Rightarrow\; g(l) > x$$

Array Partitioning : First Refinement: the State

constants: n, f, x

variables: k, g, j

inv1_1: $j \in 0 \,..\, n$

inv1_2: $k \leq j$

inv1_3: $\forall l \cdot l \in 1 \,..\, k \Rightarrow g(l) \leq x$

inv1_4: $\forall l \cdot l \in k+1 \,..\, j \Rightarrow g(l) > x$

Partitioning with 5

3	7	2	5	8	9	4	1

Partitioning with 5

3	7	2	5	8	9	4	1

Partitioning with 5

3	7	2	5	8	9	4	1

Partitioning with 5

3	2	7	5	8	9	4	1

Partitioning with 5

3	2	5	7	8	9	4	1

Partitioning with 5

3	2	5	7	8	9	4	1

Partitioning with 5

3	2	5	7	8	9	4	1

Partitioning with 5

3	2	5	4	8	9	7	1

Partitioning with 5

3	2	5	4	1	9	7	8

init
$$g, j, k := f, 0, 0$$

final
when
$$j = n$$
then
skip
end

1	$\leq x$	k	$k+1$	$> x$	j	$j+1$	?	n

progress_1
  refines
    progress
  status
    convergent
  when
    $j \neq n$
    $g(j+1) > x$
  then
    $j := j+1$
  end

variant1: $n - j$

Array Partitioning : New Event

1	$\leq x$	k, j	$j+1$	**?**	n

progress_2
  refines
    progress
  sattus
    convergent
  when
    $j \neq n$
    $g(j+1) \leq x$
    $k = j$
  then
    $k, j := k+1, j+1$
  end

variant1: $n - j$

Array Partitioning : New Event

1	$\leq x$	k	$k+1$	$> x$	j	$j+1$	?	n

progress_3
    progress
  sattus
    convergent
  when
    $j \neq n$
    $g(j+1) \leq x$
    $k \neq j$
  then
    $k, j, g := k+1, j+1,$
            swap $(g, k+1, j+1)$
  end

variant1: $n - j$

$$\textbf{swap}\,(g, k, j) \;=\; g \lhd\!\!- \{k \mapsto g(j)\} \lhd\!\!- \{j \mapsto g(k)\}$$

Partitioning with 5

3	2	5	7	8	9	4	1

Partitioning with 5

3	2	5	4	8	9	7	1

Putting together progress_2 and progress_3

progress_2
when
$j \neq n$
$g(j+1) \leq x$
$k = j$
then
$k, j := k+1, j+1$
end

progress_3
when
$j \neq n$
$g(j+1) \leq x$
$k \neq j$
then
$k, j, g := k+1, j+1,$
swap $(g, k+1, j+1)$
end

Merging Rule (2)

when P Q **then** S **end**	**when** P $\neg\, Q$ **then** T **end**	$\leadsto$	**when** P **then** **if** Q **then** S **else** T **end** **end**	M_IF

Array Partitioning : Final Merging (2)

Applying Rule M_IF to progress_2 and progress_3

```
progress_23
   when
      j ≠ n
      g(j + 1) ≤ x
   then
      if k = j then
         k, j := k + 1, j + 1
      else
         k, j, g := k + 1, j + 1, swap (g, k + 1, j + 1)
      end
   end
```

Array Partitioning : Final Merging (3)

Putting together progress_1 and progress_23

progress_1
when
$j \neq n$
$g(j+1) > x$
then
$j := j + 1$
end

progress_23
when
$j \neq n$
$g(j+1) \leq x$
then
if $k = j$ **then**
$k, j := k+1, j+1$
else
$k, j, g := k+1, j+1,$
swap $(g, k+1, j+1)$
end
end

Merging Rule (3)

when
  P
  Q
then
  S
end
	when
	  P
	$\neg Q$
	then
	  if R **then**
	    T
	  else
	    U
	  end
	end
		$\leadsto$	**when**
			  P
			then
			  if Q **then**
			    S
			  elsif R **then**
			    T
			  else
			    U
			  end
			end
				M_ELSIF

Applying M_ELSIF to progress_1 and progress_23

final
when
$j = n$
then
skip
end

progress_123
when $j \neq n$ **then**
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

Merging Rule M_WHILE (special case)

when Q **then** S **end**	**when** $\neg\, Q$ **then** skip **end**	$\leadsto$	**while** Q **do** S **end**	M_WHILE

Array Partitioning : Final Merging (6)

Applying M_WHILE4 to partition and progress_123

init
$g := f$
$j := 0$
$k := 0$

progress_123_final
while $j \neq n$ **do**
  if $g(j+1) > x$ **then**
    $j := j + 1$
  elsif $k = j$ **then**
    $k, j := k + 1, j + 1$
  else
    $k, j, g := k + 1, j + 1, \textbf{swap}\,(g, k + 1, j + 1)$
  end
end

Applying Rule M_INIT to init and progress_123_final yields

```
partition_program
    g, k, j := f, 0, 0 ;                          init
    while j ≠ m do
        if g(j + 1) > x then
            j := j + 1                            progress_1
        elsif k = j then
            k, j := k + 1, j + 1                  progress_2
        else
            k, j, g := k + 1, j + 1,
                    swap (g, k + 1, j + 1)        progress_3
        end
    end
```

Array Partitioning: Concluding Remarks

- The complete development requires 18 proofs.

- Among which 6 were interactive

Example 4: Array Sorting

- Given: A numerical array f
- Result is: Another numerical array g
- 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

axm0_1: $n \in \mathbb{N}$

axm0_2: $0 < n$

axm0_3: $f \in 1\,..\,n \rightarrowtail \mathbb{N}$

variables: g

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

init

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

final

when

$g \in 1 \,..\, n \rightarrow \mathbb{N}$

$\mathrm{ran}\,(g) = \mathrm{ran}\,(f)$

$$\forall\, i, j \cdot \begin{pmatrix} i \in 1 \,..\, n-1 \\ j \in i+1 \,..\, n \\ \Rightarrow \\ g(i) < g(j) \end{pmatrix}$$

then

skip

end

progress

status

anticipated

then

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

end

- Introducing a new variable k ranging form 1 to n

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

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

Array Sorting First Refinement: the State

variables: g, k, l

inv1_1: $g \in 1\,..\,n \rightarrowtail \mathbb{N}$

inv1_2: $\mathrm{ran}(g) = \mathrm{ran}(f)$

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

inv1_4: $\forall i, j \cdot \begin{pmatrix} i \in 1\,..\,k-1 \\ j \in i+1\,..\,n \\ \Rightarrow \\ g(i) < g(j) \end{pmatrix}$

inv1_5: $l \in \mathbb{N}$

- We introduce an anticipated variable l

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

init
  $g, k := f, 1$
  $l :\in \mathbb{N}$

final
  when $k = n$ **then** skip **end**

progress
  any l **where**
    $k < n$
    $l \in k \,..\, n$
    $g(l) = \min(g[k \,..\, n])$
  then
    $g := g \mathbin{\lhd\!\!-} \{k \mapsto g(l)\} \mathbin{\lhd\!\!-} \{l \mapsto g(k)\}$
    $k := k + 1$
    $l :\in \mathbb{N}$
  end

prog
  status
    anticipated
  then
    $l :\in \mathbb{N}$
  end

variant1: $n - k$

Introducing one new variables j in $k \,..\, n$

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

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

Array Sorting Second Refinement: the State

variables: g, k, j, l

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

inv2_2: $l \in k \,..\, j$

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

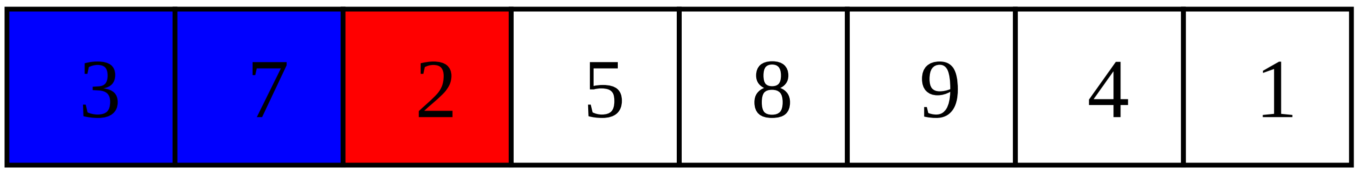
Sorting

3	7	2	5	8	9	4	1

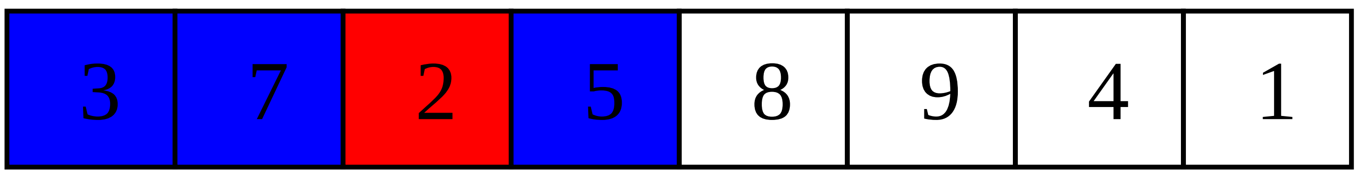
Sorting

3	7	2	5	8	9	4	1

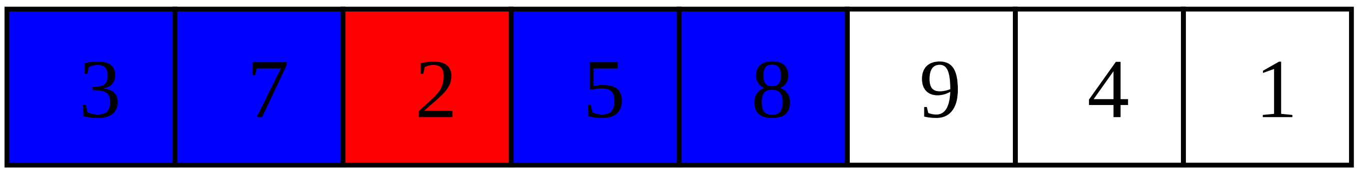
Sorting



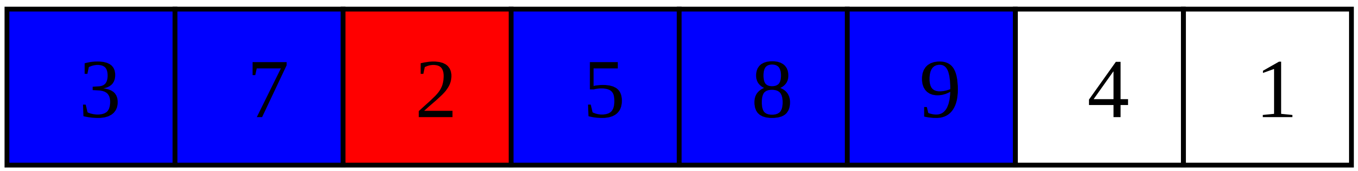
Sorting



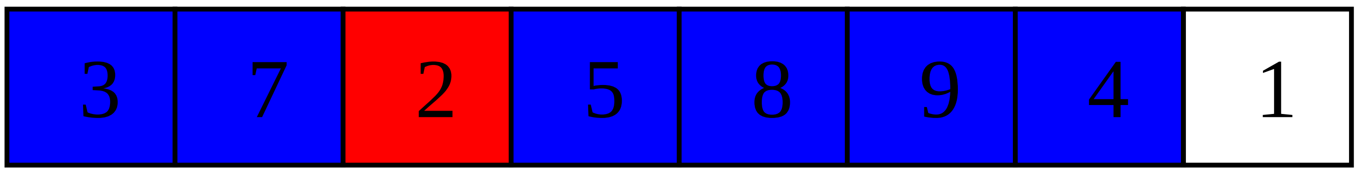
Sorting



Sorting



Sorting



Sorting

3	7	2	5	8	9	4	1

Sorting

1	7	2	5	8	9	4	3

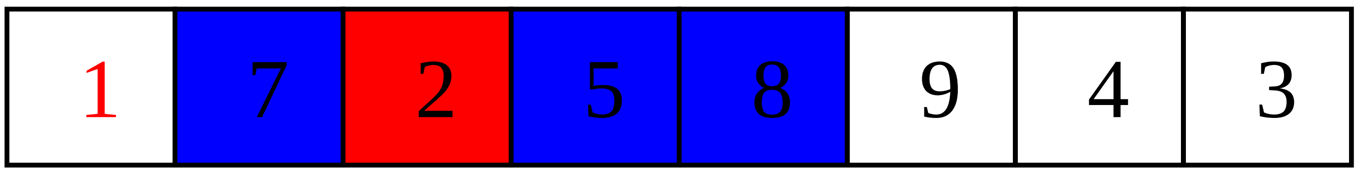
Sorting

1	7	2	5	8	9	4	3

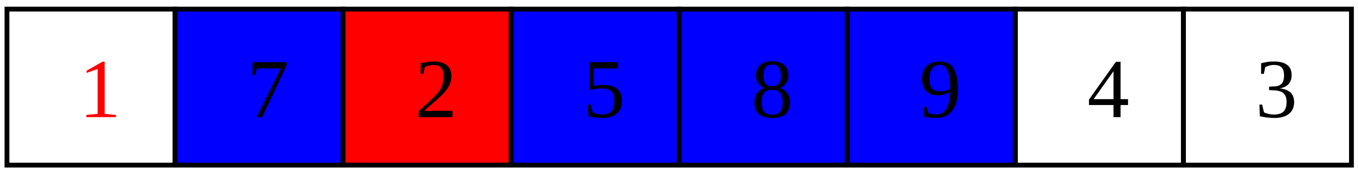
Sorting

1	7	2	5	8	9	4	3

Sorting



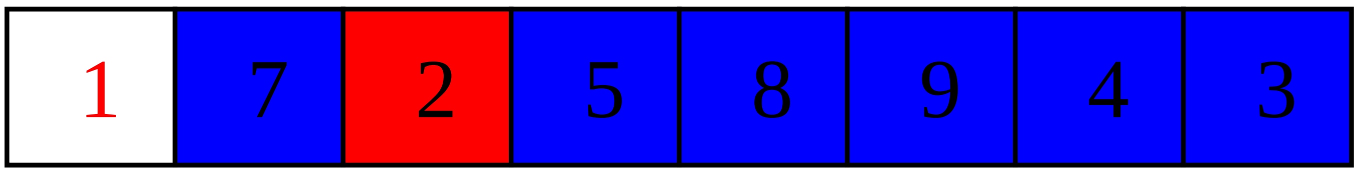
Sorting



Sorting

1	7	2	5	8	9	4	3

Sorting



Sorting

1	2	7	5	8	9	4	3

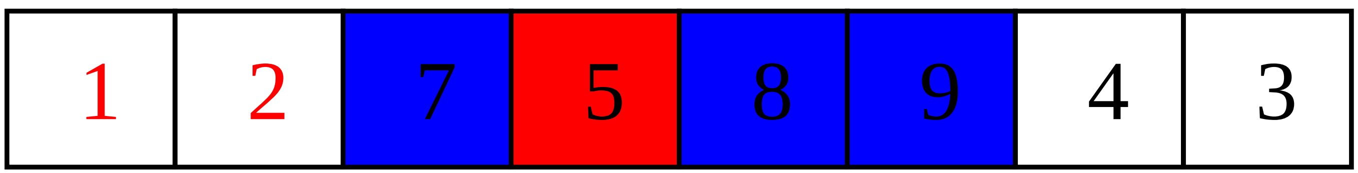
Sorting

1	2	7	5	8	9	4	3

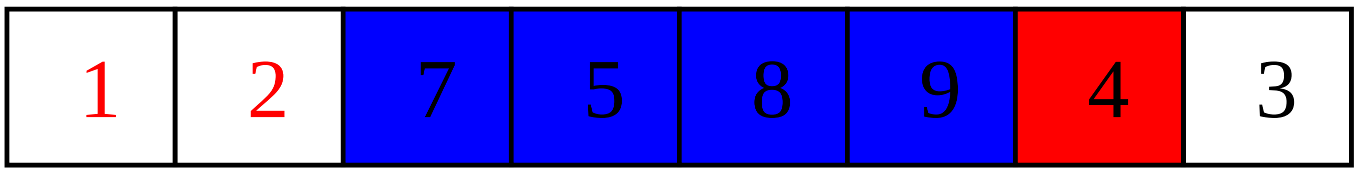
Sorting

1	2	7	5	8	9	4	3

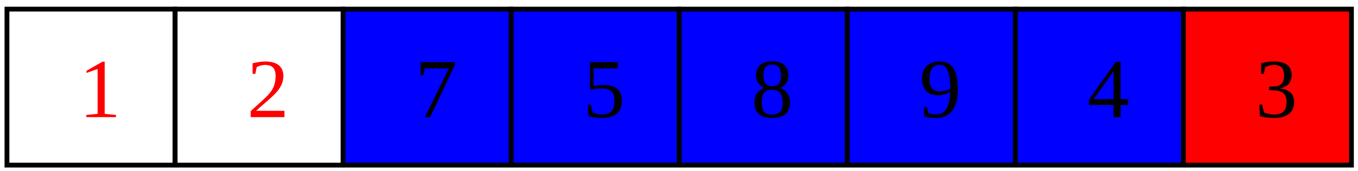
Sorting



Sorting



Sorting



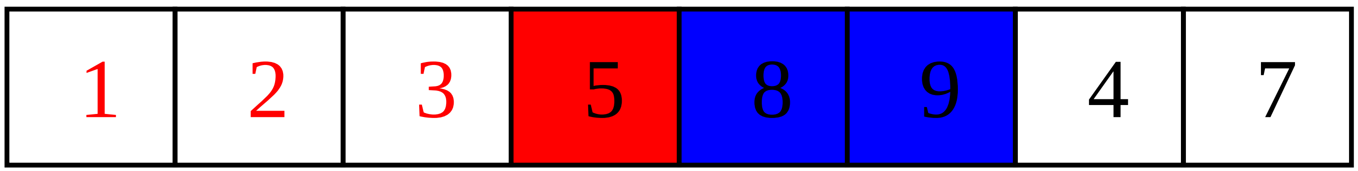
Sorting

1	2	3	5	8	9	4	7

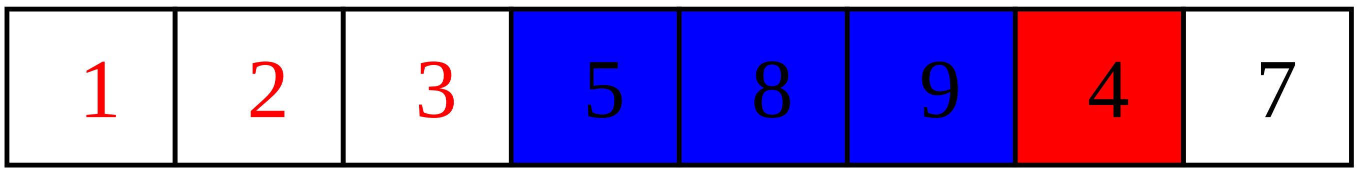
Sorting

1	2	3	5	8	9	4	7

Sorting



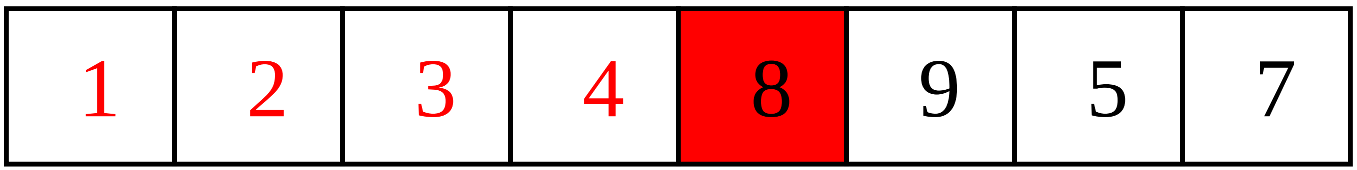
Sorting



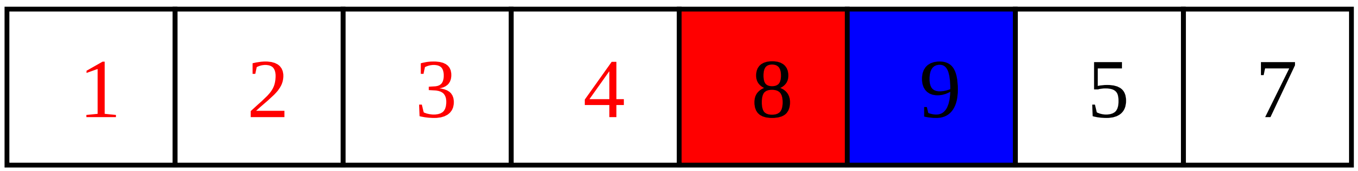
Sorting

1	2	3	5	8	9	4	7

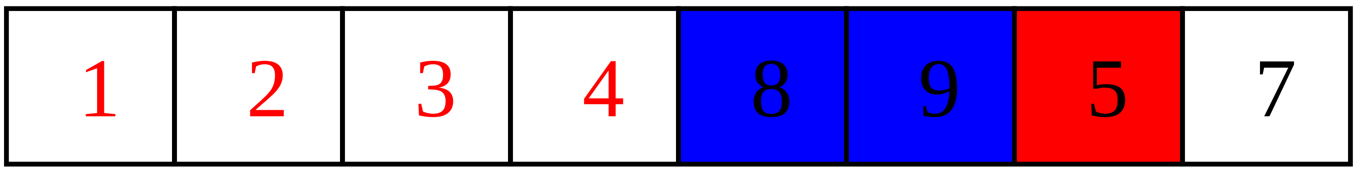
Sorting



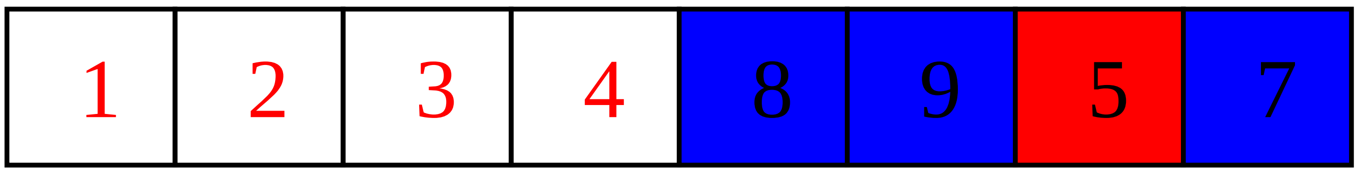
Sorting



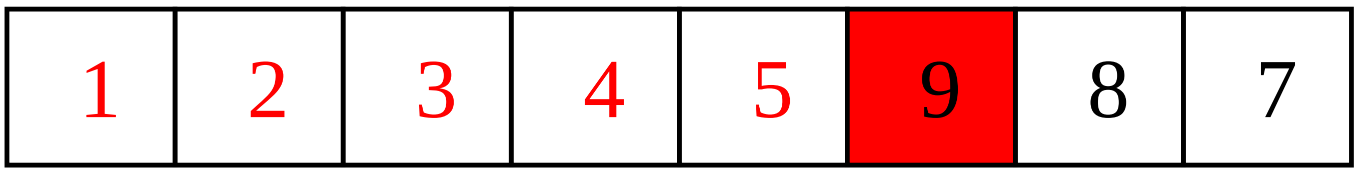
Sorting



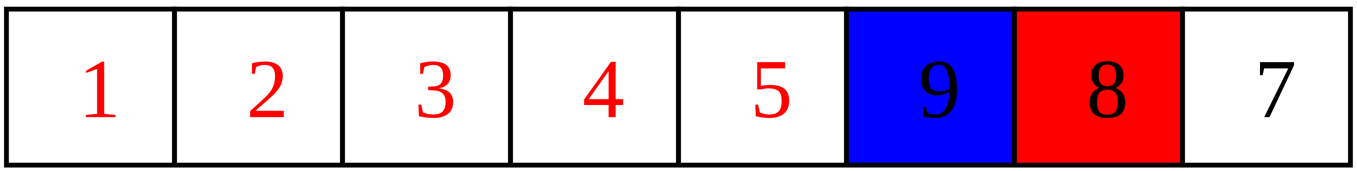
Sorting



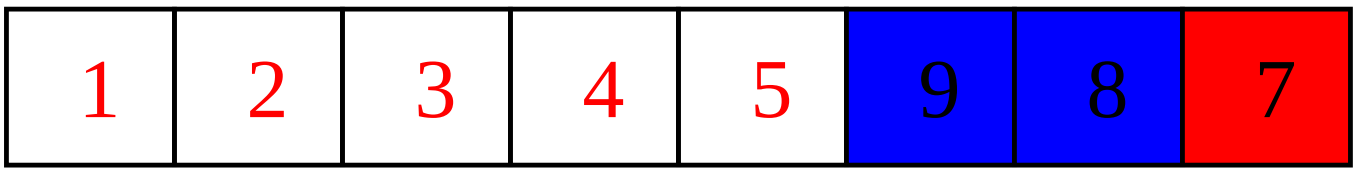
Sorting



Sorting



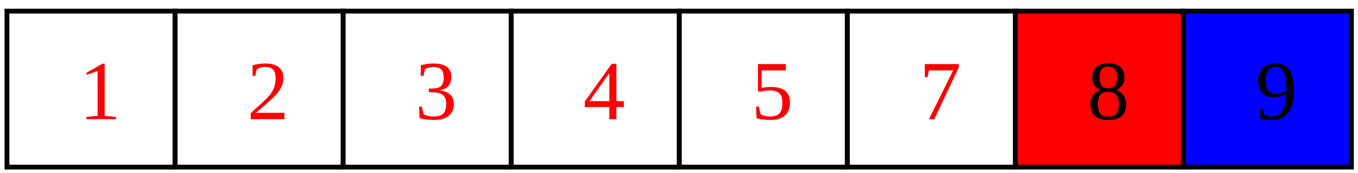
Sorting



Sorting

1	2	3	4	5	7	8	9

Sorting



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

final
when
$k = n$
then
skip
end

progress
when
$k < n$
$j = n$
then
$g := g \Leftarrow \{k \mapsto g(l)\} \Leftarrow \{l \mapsto g(k)\}$
$k, j, l := k + 1, k + 1, k + 1$
end

Sorting 2nd Refinement: Adding Events Refining event "prog"

prog1
  refines
    prog
  status
    convergent
  when
    $k < n$
    $j < n$
    $g(l) \leq g(j+1)$
  then
    $j := j + 1$
  end

prog2
  refines
    prog
  status
    convergent
  when
    $k < n$
    $j < n$
    $g(l) > g(j+1)$
  then
    $j, l := j + 1, j + 1$
  end

variant1: $n - j$

```
sort_program
  begin
    [ g, k, j, l := f, 1, 1, 1 ] ;                                   init
    while  k < n  do
      while  j < n  do
        if  g(l) ≤ g(j + 1)  then
          [ j := j + 1 ]                                             prog1
        else
          [ j, l := j + 1, j + 1 ]                                   prog2
        end
      end;
      [ k, j, l, g := k + 1, k + 1, k + 1, swap (g, k, l) ]          progress
    end
  end
```

- 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

axm0_1: $n \in \mathbb{N}$

axm0_2: $0 < n$

axm0_3: $f \in 1 \,..\, n \longrightarrow \mathbb{N}$

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

In Place Reversing of an Array: Example

Here is an array

$\mathbf{3}$	$\mathbf{2}$	$\mathbf{5}$	$\mathbf{4}$	$\mathbf{1}$	$\mathbf{9}$	$\mathbf{7}$	$\mathbf{8}$

Here is the reverse array

$\mathbf{8}$	$\mathbf{7}$	$\mathbf{9}$	$\mathbf{1}$	$\mathbf{4}$	$\mathbf{5}$	$\mathbf{2}$	$\mathbf{3}$

An element which was at index $\boldsymbol{i}$ is now at index $\mathbf{8} - \boldsymbol{i} + \mathbf{1}$

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

final
  when
    $g \in 1\,..\,n \rightarrow S$
    $\forall k \cdot \begin{pmatrix} k \in 1\,..\,n \\ \Rightarrow \\ g(k) = f(n - k + 1) \end{pmatrix}$
  then
    skip
  end

progress
  status
    anticipated
  then
    $g :\in \mathbb{N} \leftrightarrow S$
  end

- We introduce two additional variables i and j, both in $1 \,..\, n$

- 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

variables: g, i, j

inv1_1: $g \in 1\,..\,n \rightarrow S$

inv1_2: $i \in 1\,..\,n$

inv1_3: $j \in 1\,..\,n$

inv1_4: $i + j = n + 1$

inv1_5: $i \leq j + 1$

inv1_4: $i + j = n + 1$

inv1_5: $i \leq j + 1$

inv1_6: $\forall k \cdot k \in 1 \,..\, i - 1 \;\Rightarrow\; g(k) = f(n - k + 1)$

inv1_7: $\forall k \cdot k \in i \,..\, j \;\Rightarrow\; g(k) = f(k)$

inv1_8: $\forall k \cdot k \in j + 1 \,..\, n \;\Rightarrow\; g(k) = f(n - k + 1)$

1 **reversed**	i **unchanged** j	**reversed** n

init
$i := 1$
$j := n$
$g := f$

final
when
$j \leq i$
then
skip
end

variant1: $j - i$

progress
status
convergent
when
$i < j$
then
$g := g \Leftarrow \{i \mapsto g(j)\} \Leftarrow \{j \mapsto g(i)\}$
$i, j := i + 1, j - 1$
end

Final Program

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

$$\boxed{\quad f \quad} \longrightarrow \boxed{\qquad} \longrightarrow \ldots \longrightarrow \boxed{\qquad} \longrightarrow \boxed{\quad l \quad}$$

- 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

The chain is represented by a bijection c

carrier set: S

constants: d, f, l, c

axm0_1: $d \subseteq S$

axm0_2: $f \in d$

axm0_3: $l \in d$

axm0_4: $f \neq l$

axm0_5: $c \in d \setminus \{l\} \rightarrowtail\!\!\!\!\twoheadrightarrow d \setminus \{f\}$

axm0_6: $\forall T \cdot T \subseteq c[T] \Rightarrow T = \varnothing$

Reversing the Chain

- Given the following initial chain

$$\boxed{\boldsymbol{f}} \longrightarrow \boxed{\boldsymbol{x}} \longrightarrow \ldots \longrightarrow \boxed{\boldsymbol{z}} \longrightarrow \boxed{\boldsymbol{l}}$$

- Then the transformed chain should look like this:

$$\boxed{\boldsymbol{f}} \longleftarrow \boxed{\boldsymbol{x}} \longleftarrow \ldots \longleftarrow \boxed{\boldsymbol{z}} \longleftarrow \boxed{\boldsymbol{l}}$$

constants: d, f, l, c

inv0_1: $r \in S \leftrightarrow S$

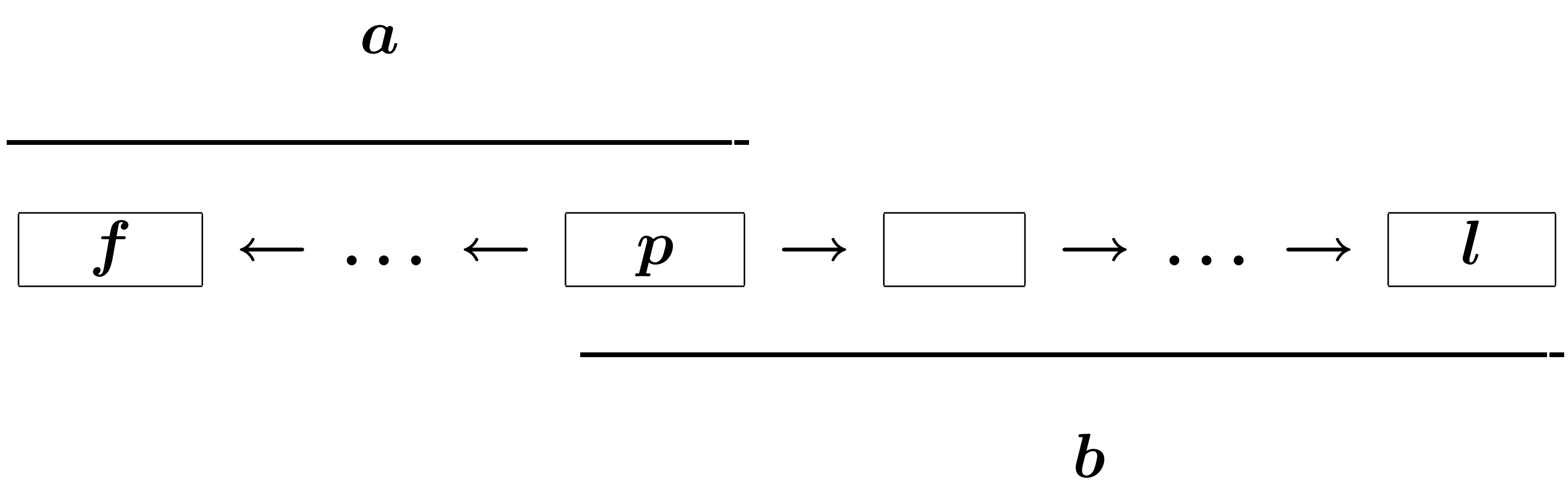
init

$r :\in S \leftrightarrow S$

reverse

$r := c^{-1}$

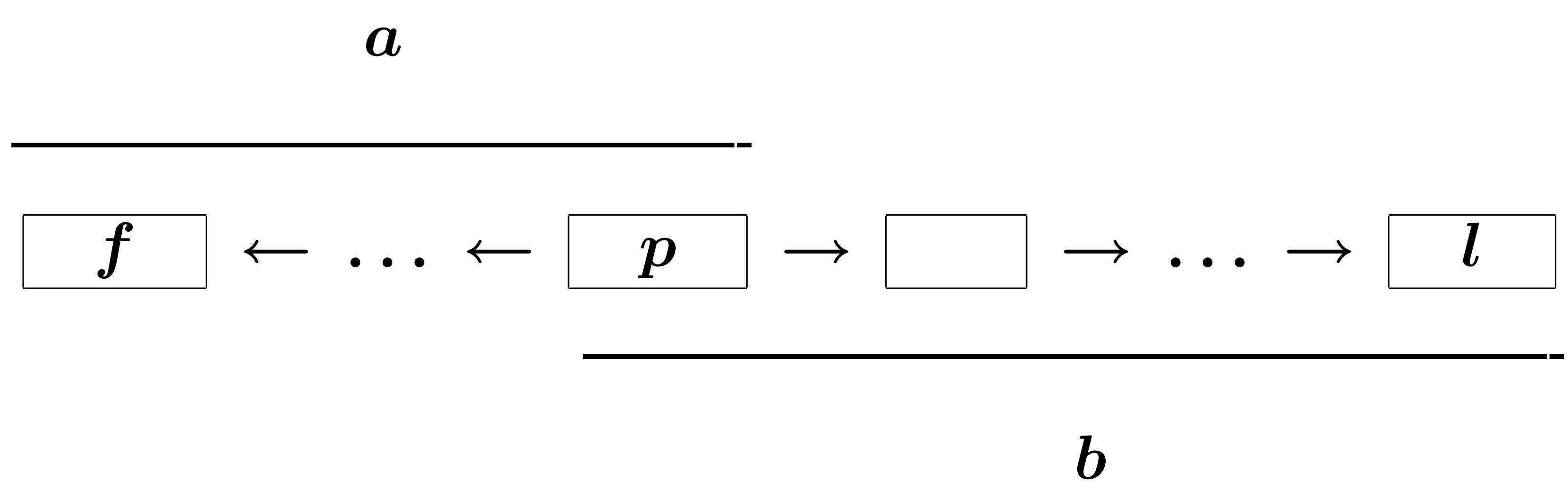
We introduce two additional chains a and b and a pointer p

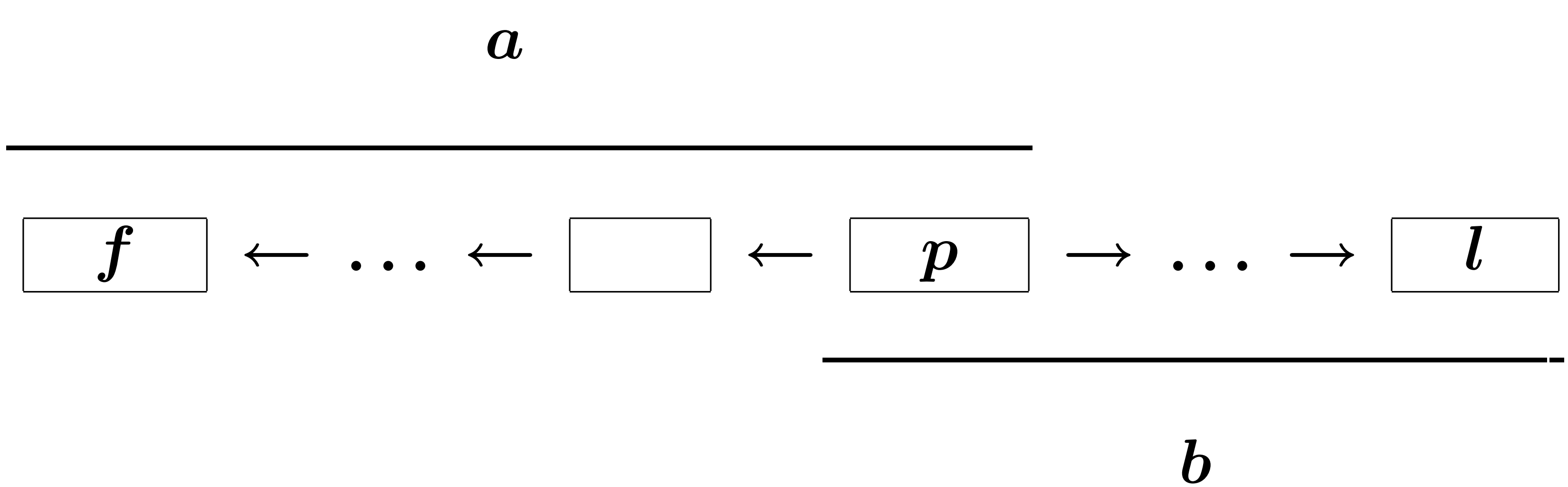


- Node p starts both chains

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

Progressing





First Refinement: the State (1)

variables: r, a, b, p

"cl" is the irreflexive transitive closure operator

inv1_1: $p \in d$

inv1_2: $a \in (\mathrm{cl}(c^{-1})[p] \cup p) \setminus \{f\} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathrm{cl}(c^{-1})[p]$

inv1_3: $b \in (\mathrm{cl}(c)[p] \cup p) \setminus \{l\} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathrm{cl}(c)[p]$

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

First Refinement: the Events

init
$\quad r :\in S \leftrightarrow S$
$\quad a, b, p := \varnothing, c, f$

reverse
$\quad$ **when**
$\quad\quad b = \varnothing$
$\quad$ **then**
$\quad\quad r := a$
$\quad$ **end**

progress
$\quad$ **when**
$\quad\quad p \in \mathrm{dom}(b)$
$\quad$ **then**
$\quad\quad p := b(p)$
$\quad\quad a(b(p)) := p$
$\quad\quad b := \{p\} \lhd b$
$\quad$ **end**

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

constants: f, l, c, nil

axm2_1: $nil \in S$

axm2_2: $nil \notin d$

variables: r, a, bn, p, q

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

inv2_2: $q = bn(p)$

Second Refinement: the Events

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

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

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

- The previous situation with two chains a and bn

$$\overbrace{\boxed{f} \leftarrow \ldots \leftarrow \boxed{p}}^{a} \quad \underbrace{\boxed{p} \rightarrow \boxed{q} \rightarrow \ldots \rightarrow \boxed{l} \rightarrow \boxed{nil}}_{bn}$$

- The new situation with a single chain d

$$\underbrace{\boxed{f} \leftarrow \ldots \leftarrow \boxed{p} \qquad \boxed{q} \rightarrow \ldots \rightarrow \boxed{l} \rightarrow \boxed{nil}}_{d}$$

variables: r, p, q, d

inv3_1: $d \in S \rightarrowtail\!\!\!\!\!+ S$

inv3_2: $d = (\{f\} \lhd bn) \lhd\!\!\!- a$

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

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

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

```
reverse_program
   p, q, d := f, c(f), {f} ⩤ (c ∪ {l ↦ nil});
   while q ≠ nil do
      p := q
      d(q) := p
      q := d(q)
   end;
   r := d ⩥ {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 is is not defined for all natural number

- We want to make it total

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

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

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

$$\mathbf{4^2 \leq 17 < 5^2}$$

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

$$\mathbf{4^2 \leq 16 < 5^2}$$

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

$$\mathbf{3^2 \leq 15 < 4^2}$$

Integer Square Root: Initial State and Events

constants: n

variables: r

axm0_1: $n \in \mathbb{N}$

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

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

final
  when
    $r^2 \leq n$
    $n < (r+1)^2$
  then
    skip
  end

progress
  status
    anticipated
  then
    $r :\in \mathbb{N}$
  end

inv1_1: $r^2 \leq n$

variant1: $n - r^2$

init
  $r := 0$

square_root
  when
    $n < (r+1)^2$
  then
    skip
  end

progress
  status
    convergent
  when
    $(r+1)^2 \leq n$
  then
    $r := r + 1$
  end

Program after First Refinement

We obtain the following program:

```
square_root_program
   r := 0;
   while (r + 1)^2 ≤ n do
      r := r + 1
   end
```

- We do not want to compute $(r+1)^2$ at each step

- We observe the following

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

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

- We introduce two numbers a and b such that

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

$$b = 2r+3$$

constants: n

variables: r, a, b

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

inv2_2: $b = 2r + 3$

init
$r := 0$
$a := 1$
$b := 3$

final
when
$n < a$
then
skip
end

progress
when
$a \leq n$
then
$r := r + 1$
$a := a + b$
$b := b + 2$
end

Program after Second Refinement

We obtain the following program:

square_root_program
  $r, a, b := 0, 1, 3;$
  while $a \leq n$ **do**
    $r, a, b := r + 1, a + b, b + 2$
  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 form the binary search development

Inverse of an Injective Numerical Function: the State

constants: f, n

variables: r

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

axm0_1: $f \in \mathbb{N} \rightarrow \mathbb{N}$

axm0_2: $\forall i, j \cdot \begin{pmatrix} i \in \mathbb{N} \\ j \in \mathbb{N} \\ i < j \\ \Rightarrow \\ f(i) < f(j) \end{pmatrix}$

axm0_3: $n \in \mathbb{N}$

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

```
init
   r :∈ ℕ
```

```
final
   when
      f(r) ≤ n < f(r + 1)
   then
      skip
   end
```

```
progress
   status
      anticipated
   then
      r :∈ ℕ
   end
```

- 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 $a \mathrel{..} b$

- We try to make this interval narrower

- We introduce a constant q such that:

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

constants: f, n, a, b

variables: r, q

axm1_1: $a \in \mathbb{N}$

axm1_2: $b \in \mathbb{N}$

axm1_3: $f(a) \leq n$

axm1_4: $n < f(b+1)$

First Refinement: the State (2)

inv1_1: $q \in \mathbb{N}$

inv1_2: $r \leq q$

inv1_3: $f(r) \leq n$

inv1_4: $n < f(q+1)$

First Refinement: the Events (1)

init
$r, q := a, b$

final
when
$r = q$
then
skip
end

```
dec
   refines
      progress
   status
      convergent
   any x where
      r ≠ q
      x ∈ r + 1 .. q
      n < f(x)
   then
      q := x − 1
   end
```

```
inc
   refines
      progress
   status
      convergent
   any x where
      r ≠ q
      x ∈ r + 1 .. q
      f(x) ≤ n
   then
      r := x
   end
```

variant1: $q - r$

- We reduce the non-determinacy

dec	inc
when	**when**
$r \neq q$	$r \neq q$
$n < f((r+1+q)/2)$	$f((r+1+q)/2) \leq n$
then	**then**
$q := (r+1+q)/2 - 1$	$r := (r+1+q)/2$
end	**end**

```
inverse_program
   r, q := a, b;
   while r ≠ q do
      if n < f((r + 1 + q)/2) then
         q := (r + 1 + q)/2 − 1
      else
         r := (r + 1 + q)/2
      end
   end
```

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:

  - **axm0_1**: f is a total function
  - **axm0_2**: f is increasing
  - **axm1_3** and **axm1_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
   r, q := 0, n;
   while r ≠ q do
      if n < ((r + 1 + q)/2)^2 then
         q := (r + 1 + q)/2 − 1
      else
         r := (r + 1 + q)/2
      end
   end
```

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
   r, q := 0, n;
   while p ≠ q do
      if n < m × (r + 1 + q)/2) then
         q := (r + 1 + q)/2 − 1
      else
         r := (r + 1 + q)/2
      end
   end
```