# Event-B Course

# 9. Sequential Program Development

Jean-Raymond Abrial

September-October-November 2011

# 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 (;)

# An Example

```
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 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 Approach (2)

- Each 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
```

# 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**
$k, j, g := \ldots$
**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

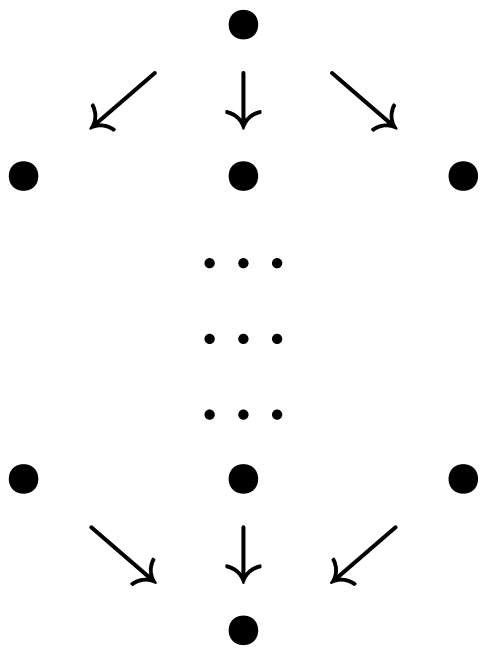
- We shall construct the events first

- And then compose our program from these events

Specification Phase

Design Phase

Composing Phase



initial event: Specification

new events: Refinements

final event: Program

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

- It is expressed by means of a Hoare-triple

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

# Encoding a Hoare triple in Event-B

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

- The parameters are constants.

- The pre-conditions are the axioms of these constants.

- The results are variables.

- The post-conditions are the guards of an event with a skip action.

# 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 $n$: $n \in \mathbb{N}$
  - $n$ 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 $S$: $f \in 1..n \rightarrow S$

# 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: $v \in \mathbf{ran}(f)$

# 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: $v \in \mathbf{ran}(f)$

- 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: $v \in \mathbf{ran}(f)$

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

# 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: $v \in \mathbf{ran}(f)$

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

# Example 1: The search Program

- We are given (Pre-condition)
  - a natural number $n$: $n \in \mathbb{N}$
  - $n$ 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 the Hoare-triple in an Event System



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

# Encoding the Hoare-triple in an Event System



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

**sets:** $S$

**constants:** $n$, $f$, $v$

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

**axm0_2:** $0 < n$

**axm0_3:** $f \in 1 \mathinner{..} n \to S$

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

# Encoding the Hoare-triple in an Event System



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

**sets:** $S$

**constants:** $n$, $f$, $v$

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

**axm0_2:** $0 < n$

**axm0_3:** $f \in 1 \mathinner{..} n \to S$

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

**variables:** $r$

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

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

final
**when**
$r \in 1 \mathinner{..} n$
$f(r) = v$
**then**
skip
**end**

# Introducing an additional anticipated Event



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

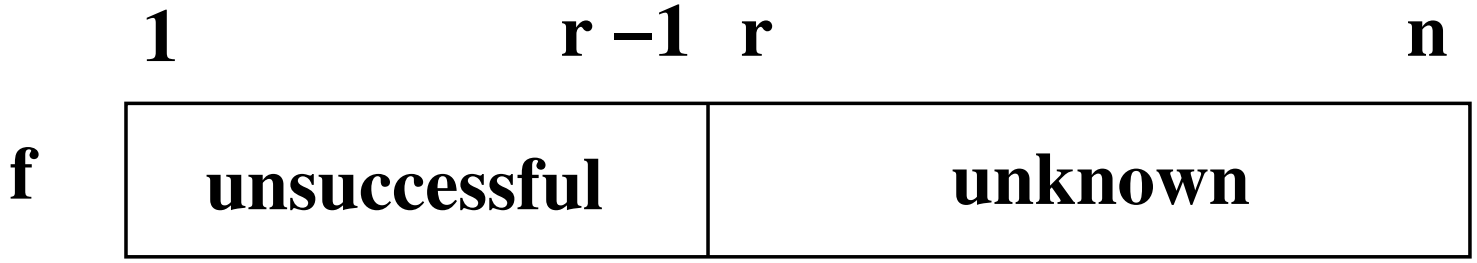
- This event modifies $r$ non-deterministically

We introduce more invariants for the result $r$

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

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

- This can be illustrated in the following figure:



# Refinement: the Events

init
$r := 1$

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

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

- The event progress is now made convergent

- We thus propose a variant:

**variant1:** $n - r$

# To be Proved (as usual)

- Events search and init refine their abstractions

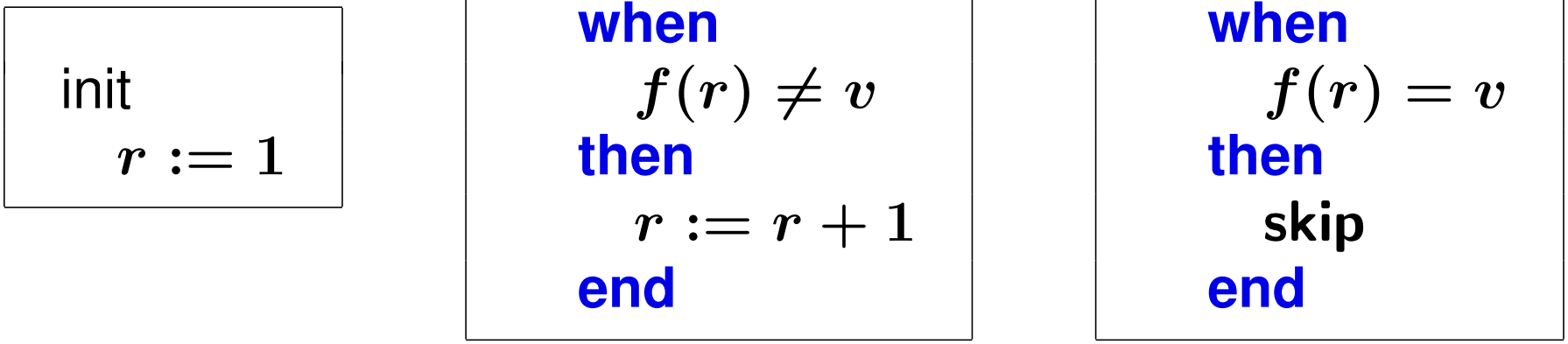
- The exhibited variant is a natural number

- "New" event progress decreases the variant

- The system is deadlock free

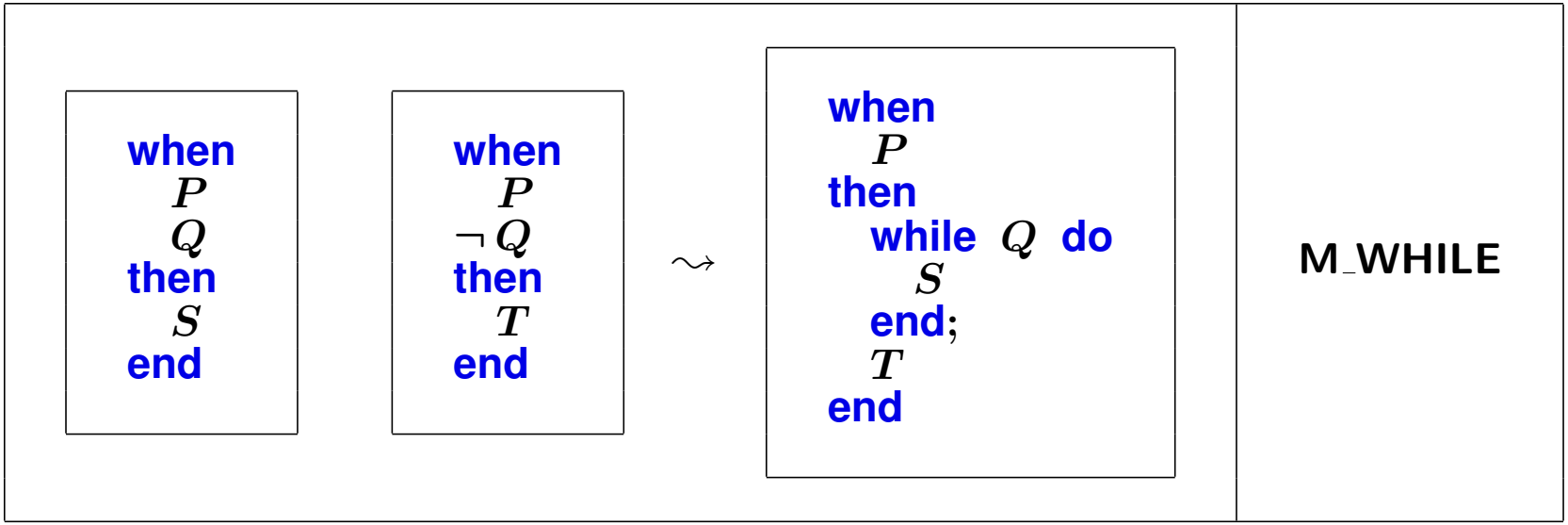
# Constructing the Final Program

We are using some Merging Rules to build the final program



# Merging Rule M_WHILE

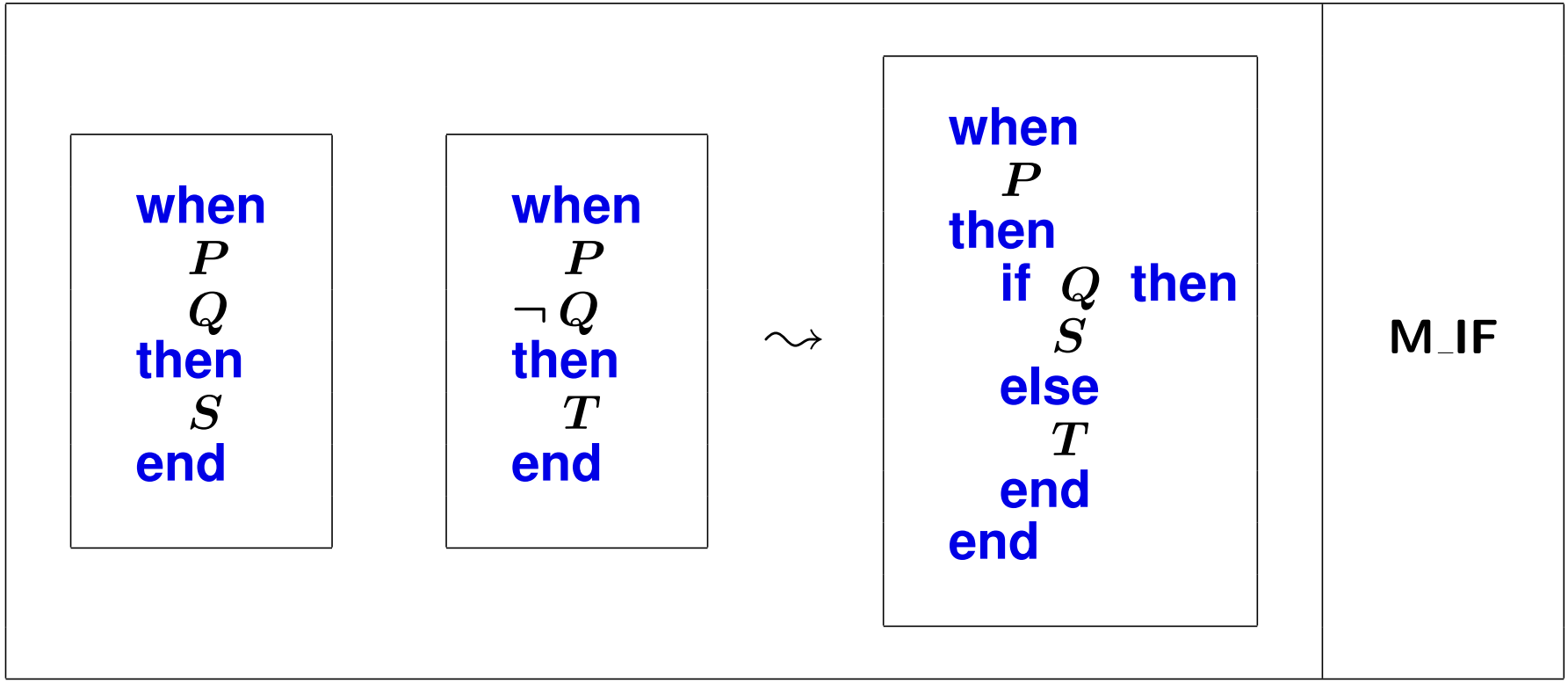





- Side Conditions:
  - $P$ must be invariant under $S$
  - The first event introduced at one level below the second one.
- The resulting level is that of the second event
- Special Case: If $P$ is missing the resulting "event" has no guard

# Merging Rule M_IF






- Side Conditions:
  - The two events introduced at the same refinement level
- The resulting level is the same
- Special Case: If $P$ is missing the resulting "event" has no guard

# Applying Rule M_WHILE (special case)



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

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

progress_final
**while** $f(r) \neq v$ **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**

$\{ n \in \mathbb{N},\ 0 < n,\ f \in 1 \mathinner{..} n \rightarrow S,\ v \in \mathrm{ran}(f) \}$

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

$\{ r \in \mathrm{dom}(f),\ f(r) = v \}$

# Example 2: The Very Classical Binary Search

- Almost the same specification as in Example 1

- It will show the usage of more merging rules

- 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: $v \in \mathbf{ran}(f)$

# Binary Search

- 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: $v \in \mathbf{ran}(f)$

- We are looking for (Post-condition)

# Binary Search

- 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: $v \in \mathbf{ran}(f)$

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

# Binary Search

- 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: $v \in \mathbf{ran}(f)$

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

# Binary Search: Encoding of the pre-condition



**constants:** $n$, $f$, $v$

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

**axm0_2:** $f \in 1 \mathinner{..} n \to \mathbb{N}$

**axm0_3:** $v \in \mathrm{ran}(f)$

**thm0_1:** $n \geq 1$

**axm0_4:** $\forall i, j \cdot \; i \in 1 \mathinner{..} n \;\; j \in 1 \mathinner{..} n \;\; i \leq j \;\; \Rightarrow \;\; f(i) \leq f(j)$

# Binary Search: Encoding of the post-condition



**variables:** $r$

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

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

final
**when**
$r \in 1 \mathinner{..} n$
$f(r) = v$
**then**
**skip**
**end**

- We have also an anticipated event:

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

# First Refinement: the State



- We introduce two new variables $p$ and $q$

**variables:** $r$, $p$, $q$

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

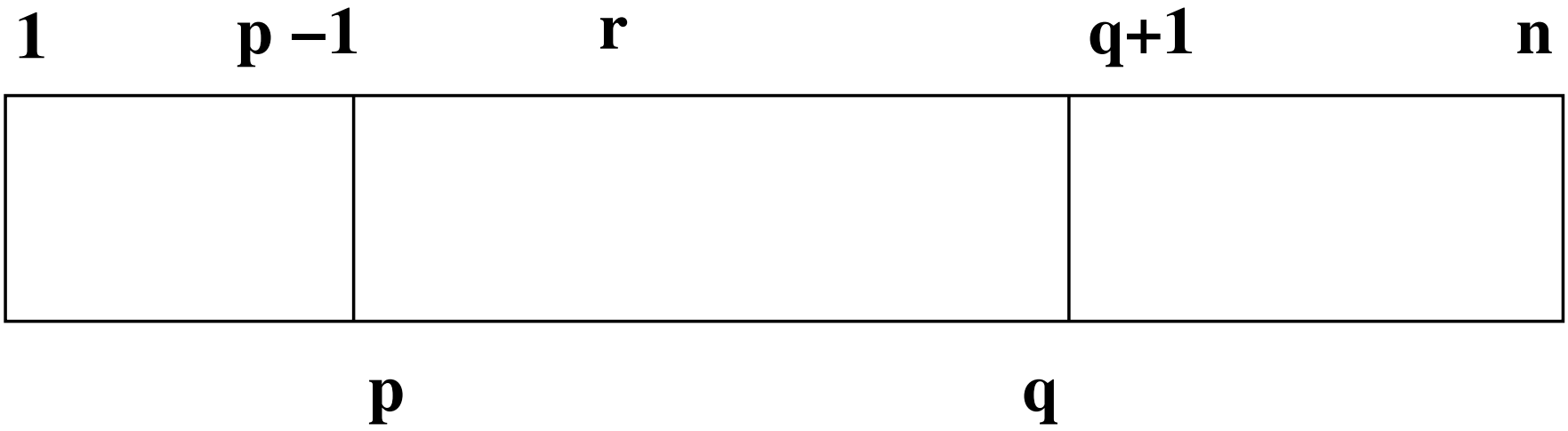
**inv1_2:** $q \in 1 \,..\, n$

**inv1_3:** $r \in p \,..\, q$

**inv1_4:** $v \in f[p \,..\, q]$

**variant1:** $q - p$

- The current situation is illustrated in the following figure:



# First Refinement: the Events



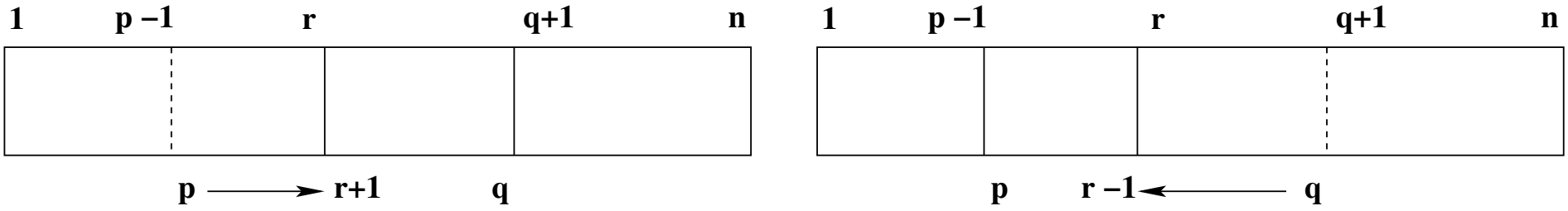
init
$p := 1$
$q := n$
$r :\in 1 \,..\, n$

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

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

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

The following figure illustrates the situation encountered by events inc (left) and dec (right)



# Proofs

- Proofs of inc

- Feasibility of inc

- Proofs and feasibility for dec (similar to those for inc)

- Proofs for final (obvious)

- Proofs of non-divergence of inc and dec (variant: $q - p$)

- Proof of dealock freeness (easy)

# 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-determincy



- $r$ is chosen in the "middle" of the intervals $r+1\,..\,q$ or $p\,..\,r-1$.

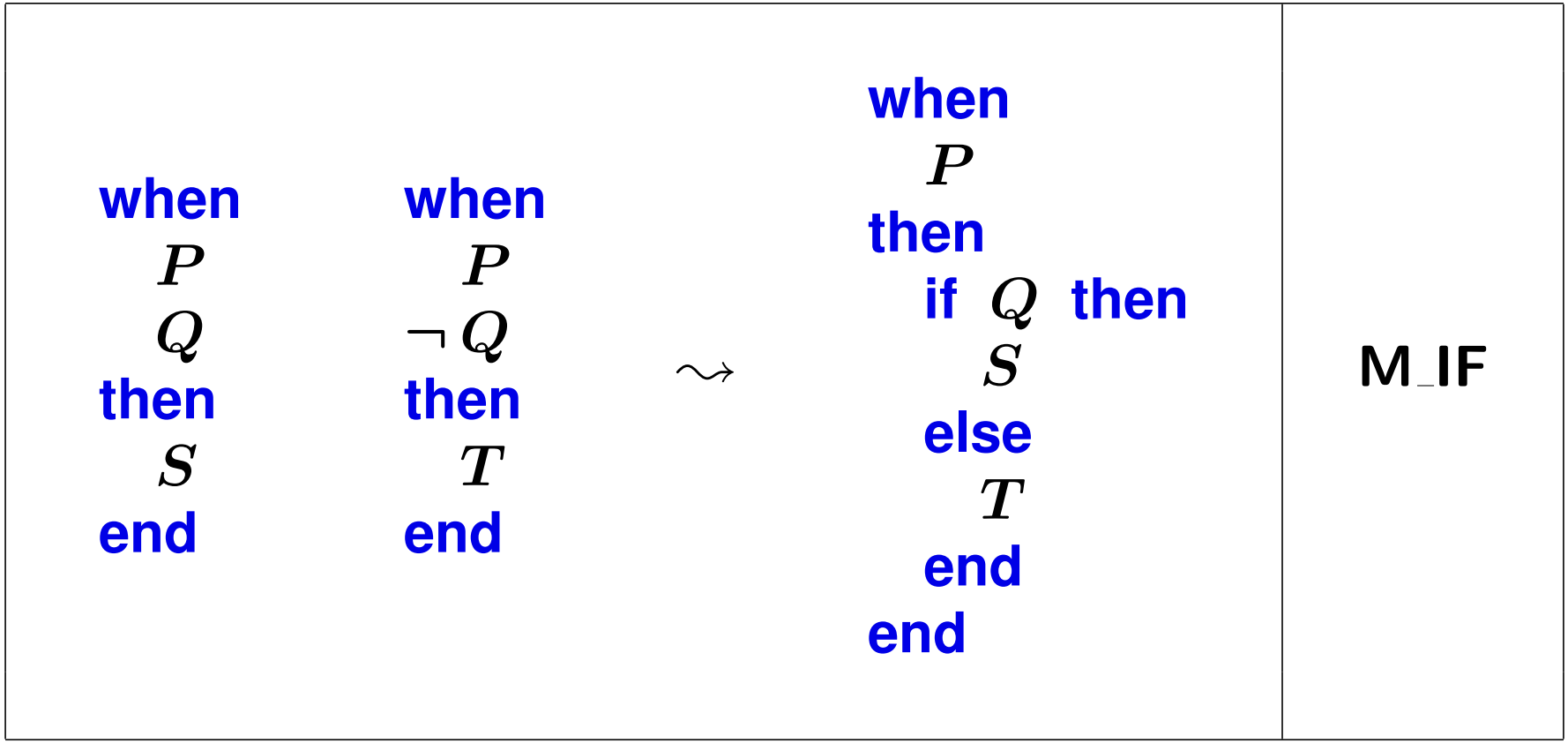
init
$p := 1$
$q := n$
$r := (1+n)/2$

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**

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

# Merging Rule M_IF



# Merging Events inc and dec by means of Rule M_IF

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

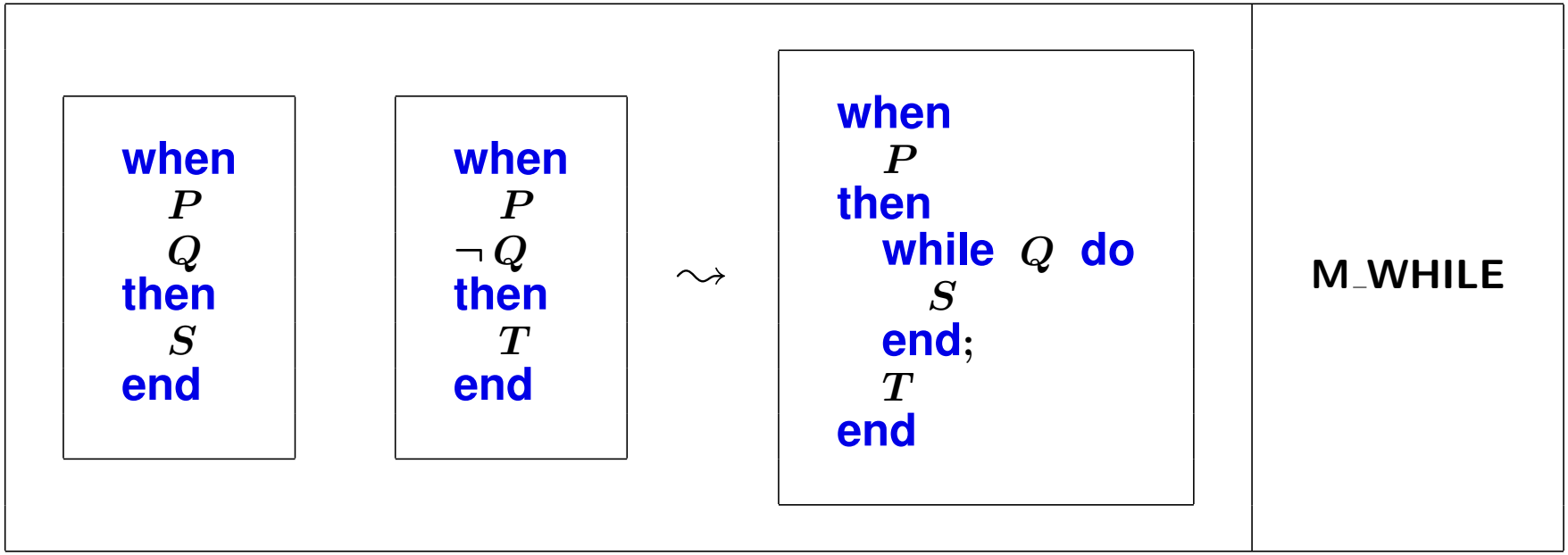
dec
**when**
$f(r) \neq v$
$v \leq f(r)$
**then**
$q := r - 1$
$s := (p + r - 1)/2$
**end**

inc_dec
**when**
$f(r) \neq v$
**then**
**if** $f(s) < v$ **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






- 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**
$f(r) \neq v$
**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**

inc_dec_final
**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**

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

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

# Merging Events inc_dec_bin_search and init with Rule M_INIT



inc_dec_final

**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**

init

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

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 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 $k+1 \,..\, n$

| | |
|---|---|
| $1 \qquad \leq x \qquad k$ | $k+1 \qquad > x \qquad n$ |

# 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:

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

- Let $x$ be equal to 10

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

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

$k$

# Initial Model



**constants:** $n$, $f$, $x$

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

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

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

**variables:** $k$, $g$

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

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

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

final
**when**
$k \in 0 \mathinner{..} n$
$g \in 1 \mathinner{..} n \rightarrowtail \mathbb{N}$
$\mathrm{ran}(g) = \mathrm{ran}(f)$
$\forall m \cdot m \in 1 \mathinner{..} k \Rightarrow g(m) \leq x$
$\forall m \cdot m \in k+1 \mathinner{..} n \Rightarrow g(m) > x$
**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 $j$

| | | |
|---|---|---|
| $1 \quad \leq x \quad k$ | $k+1 \quad > x \quad j$ | $j+1 \quad ? \quad n$ |

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:** $k, g, j$

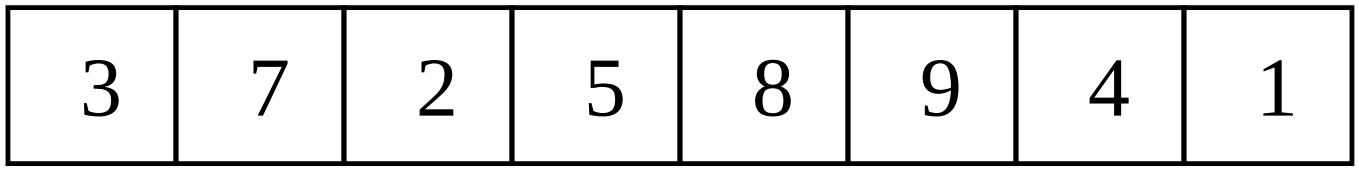
**inv1_1:** $j \in 0 \mathinner{..} n$

**inv1_2:** $k \leq j$

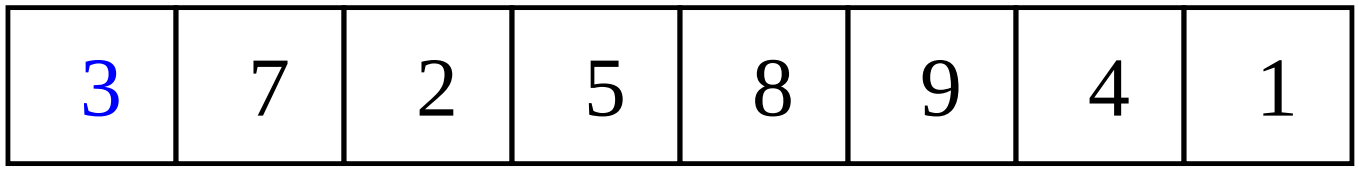
**inv1_3:** $\forall l \cdot (l \in 1 \mathinner{..} k \Rightarrow g(l) \leq x)$

**inv1_4:** $\forall l \cdot (l \in k+1 \mathinner{..} j \Rightarrow g(l) > x)$

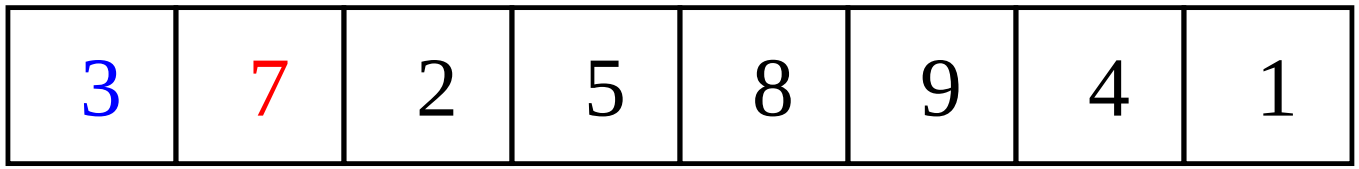
# Partitioning with 5



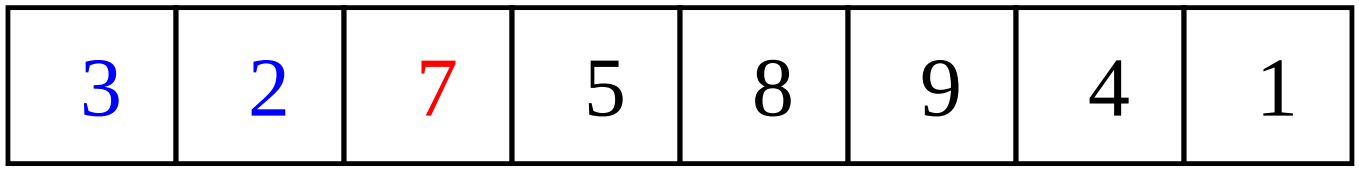
# Partitioning with 5



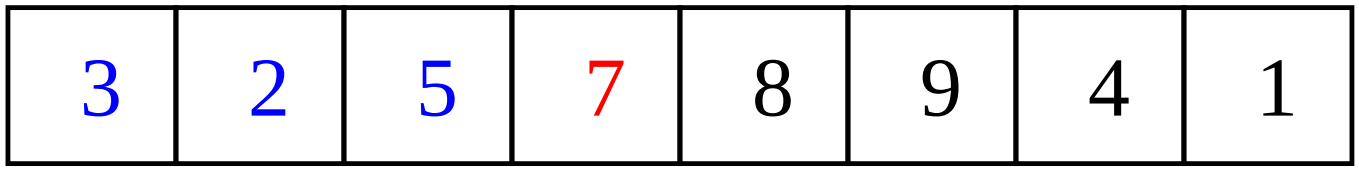
# Partitioning with 5



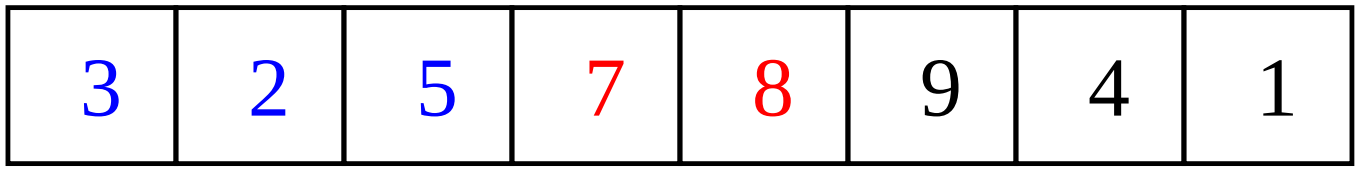
# Partitioning with 5



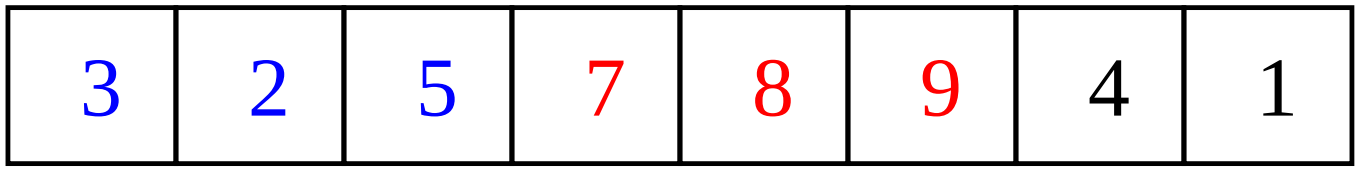
# Partitioning with 5



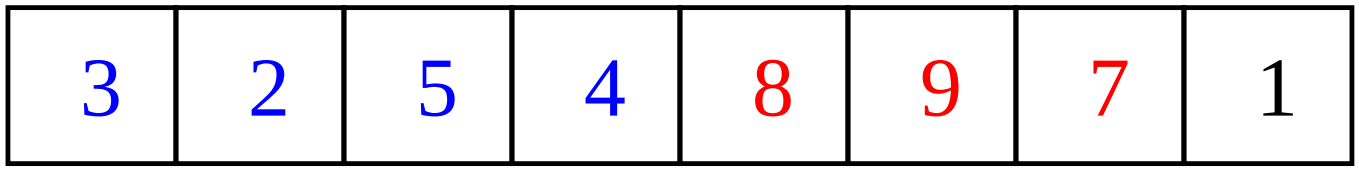
# Partitioning with 5



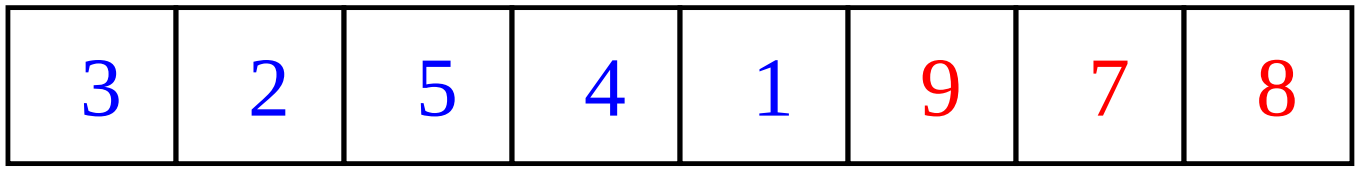
# Partitioning with 5



# Partitioning with 5



# Partitioning with 5



# Array Partitioning : Refining Existing Events (1)



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

partition
**when**
$j = n$
**then**
**skip**
**end**

# Array Partitioning : New event

| | | |
|---|---|---|
| $1 \quad \leq x \quad k$ | $k+1 \quad > x \quad j$ | $j+1 \quad ? \quad n$ |

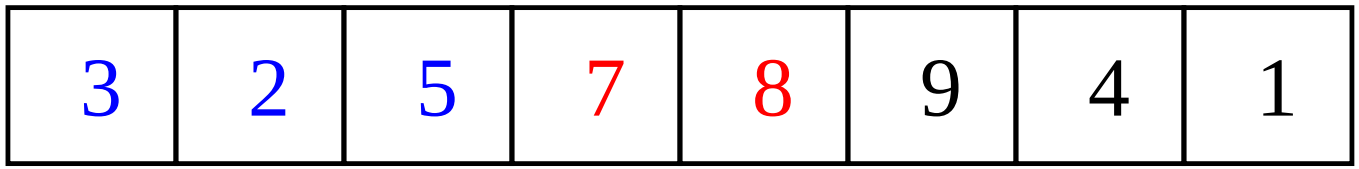
progress_1
**when**
$j \neq n$
$g(j+1) > x$
**then**
?
**end**

# Array Partitioning : New event

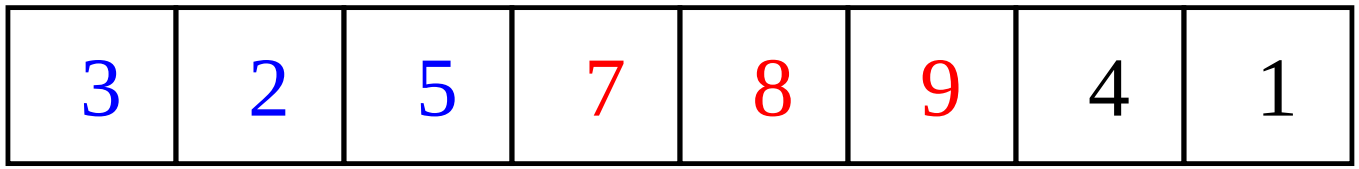
| | | |
|---|---|---|
| $1 \quad \leq x \quad k$ | $k+1 \quad > x \quad j$ | $j+1 \quad ? \quad n$ |

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

# Partitioning with 5



# Partitioning with 5



# Array Partitioning : New Event

| $1 \quad \leq x \quad k, j$ | $j+1 \quad ? \quad n$ |
|---|---|

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

# Array Partitioning : New Event



| $1 \quad \leq x \quad k, j$ | $j+1 \quad ? \quad n$ |
|---|---|

progress_2

**when**

$j \neq n$

$g(j+1) \leq x$

$k = j$

**then**

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

**end**

# Array Partitioning : New Event

| | | |
|---|---|---|
| $1 \quad \leq x \quad k$ | $k+1 \quad > x \quad j$ | $j+1 \quad ? \quad n$ |

progress_3

**when**

$j \neq n$

$g(j+1) \leq x$

$k \neq j$

**then**

**?**

**end**

# Array Partitioning : New Event

| | | |
|---|---|---|
| $1 \quad \leq x \quad k$ | $k+1 \quad > x \quad j$ | $j+1 \quad ? \quad n$ |

progress_3

**when**

$j \neq n$

$g(j+1) \leq x$

$k \neq j$

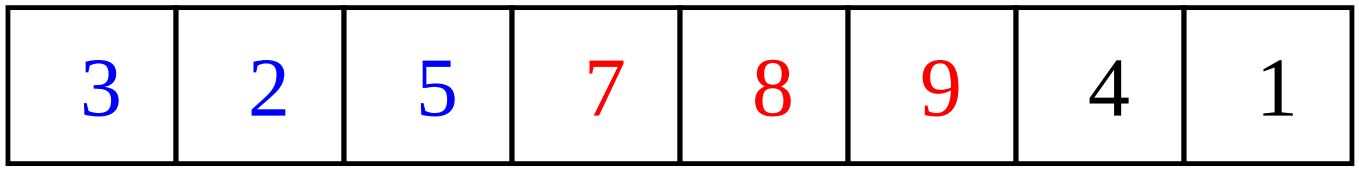
**then**

$k, j, g := k+1, j+1, \mathsf{swap}\,(g, k+1, j+1)$

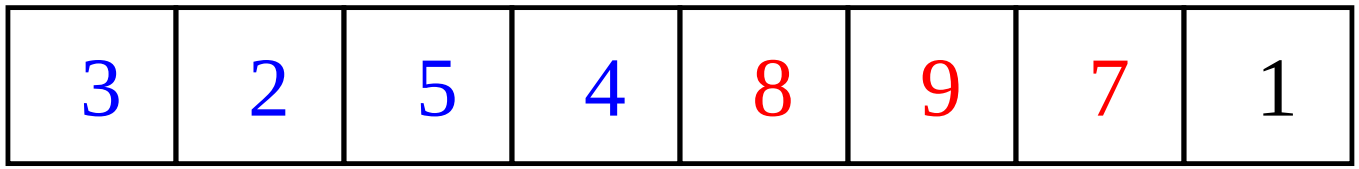
**end**

$$\mathsf{swap}\,(g,k,j) \;=\; g \Leftarrow \{k \mapsto g(j)\} \Leftarrow \{j \mapsto g(k)\}$$

# Partitioning with 5



# Partitioning with 5



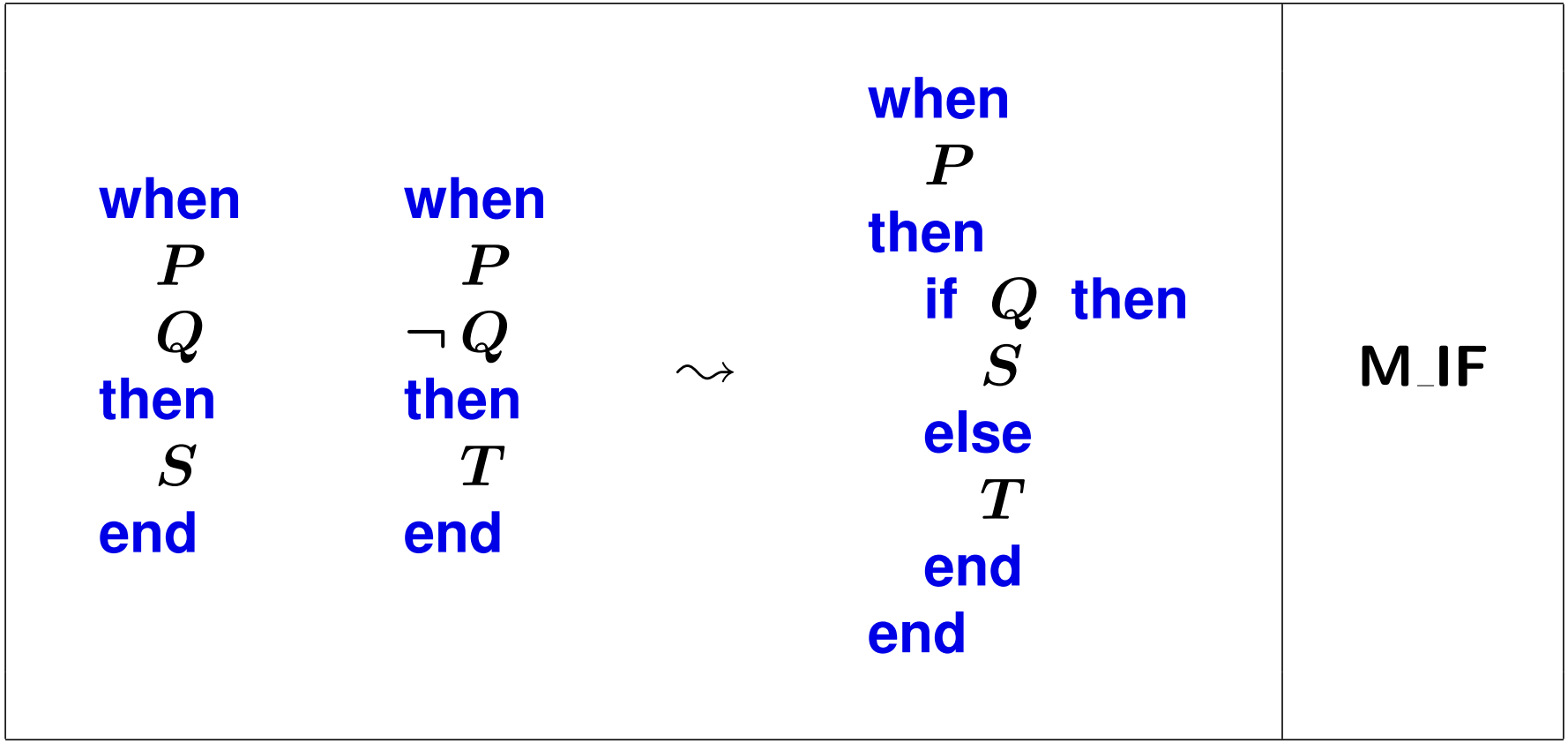
# Array Partitioning : Final Merging (1)

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)



# Array Partitioning : Final Merging (2)

Applying Rule M_IF to progress_2 and progress_3

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

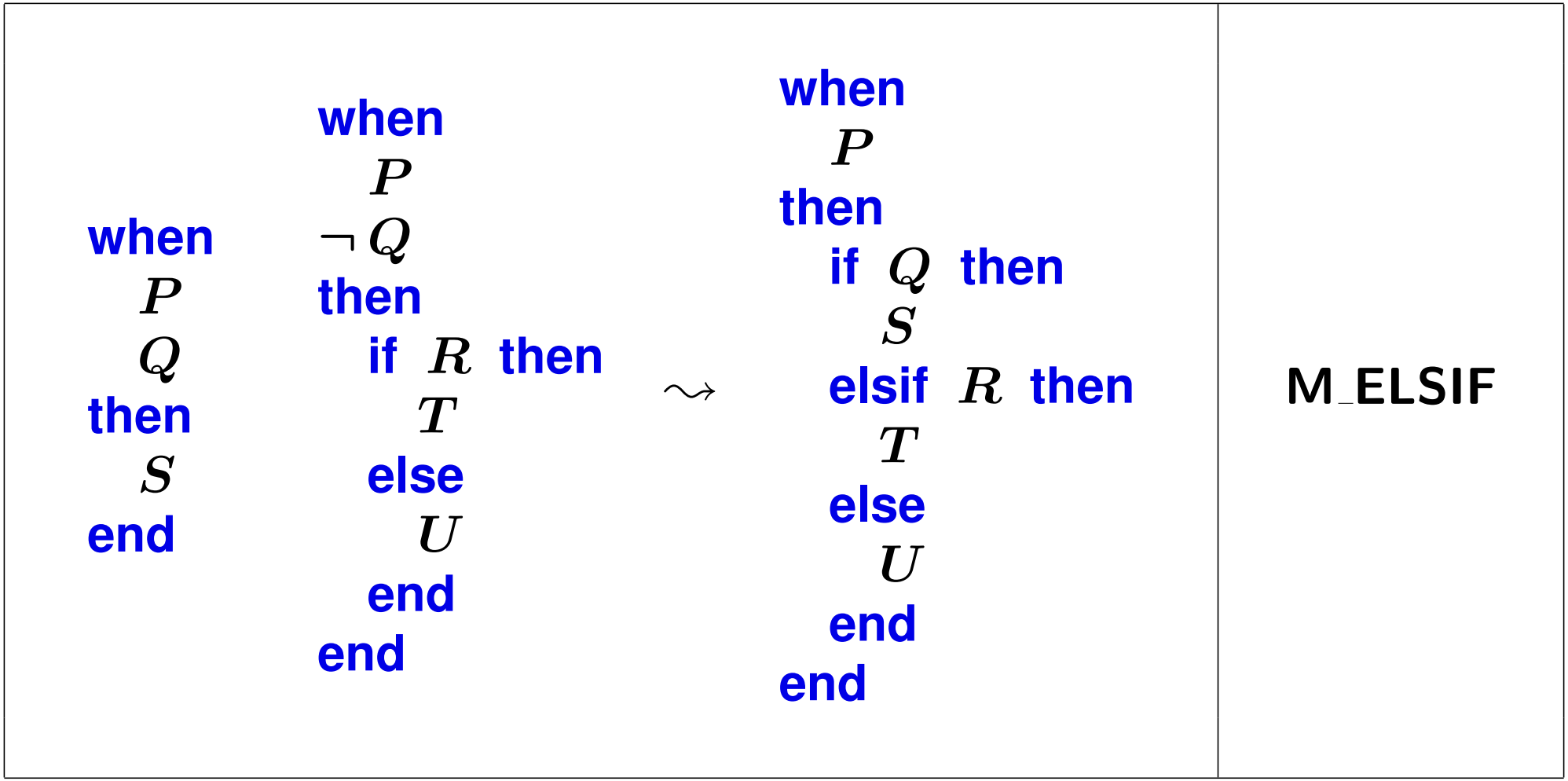
$k, j, g := k + 1, j + 1,$

**swap** $(g, k + 1, j + 1)$

**end**

**end**

# Merging Rule (3)



# Array Partitioning : Final Merging (4)

Applying M_ELSIF to progress_1 and progress_23

partition
**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**
$k, j, g := k + 1, j + 1,$ **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** | $\rightsquigarrow$ | **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_partition
**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**
    $k, j, g := k + 1, j + 1,$ **swap** $(g, k + 1, j + 1)$
  **end**
**end**

# Array Partitioning : Final Program

Applying Rule M_INIT to init and progress_123_partition 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,                         progress_3
                    swap (g, k + 1, j + 1)
      end
   end
```

# Array Partitioning: Concluding Remarks

- The complete development requires 18 proofs.

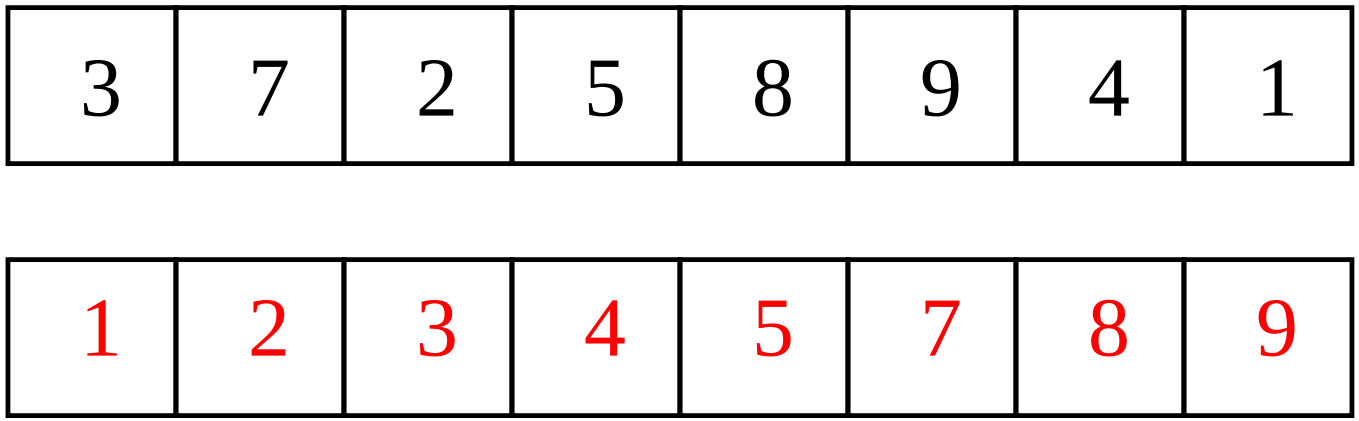
- Among which 6 were interactive

# Example 4: Array Sorting



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

# Sorting



# Sorting Initial State



**constants:** $n$ $f$

**axm0_1:** $0 < n$

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

**variables:** $g$

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

# Sorting Initial Events



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

final
**when**
$g \in 1 \mathinner{..} n \rightarrowtail \mathbb{N}$
$\mathrm{ran}(g) = \mathrm{ran}(f)$
$\forall i, j \cdot \; i \in 1 \mathinner{..} n-1$
$j \in i+1 \mathinner{..} n$
$\Rightarrow$
$g(i) < g(j)$
**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:** $g \in 1 \mathinner{..} n \rightarrowtail \mathbb{N}$

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

**inv1_3:** $k \in 1 \mathinner{..} n$

**inv1_4:** $\forall i, j \cdot \; i \in 1 \mathinner{..} k-1 \;\; j \in i+1 \mathinner{..} n \;\; \Rightarrow \;\; g(i) < g(j)$

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

# Array Sorting First Refinement: the Events



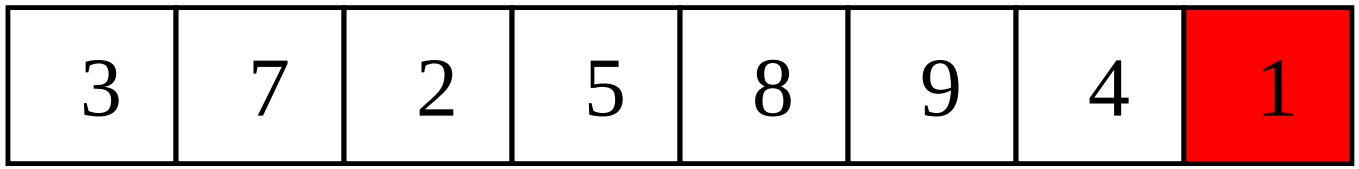
init
$g := f$
$k := 1$
$l :\in \mathbb{N}$

final
when
$k = n$
then
skip
end

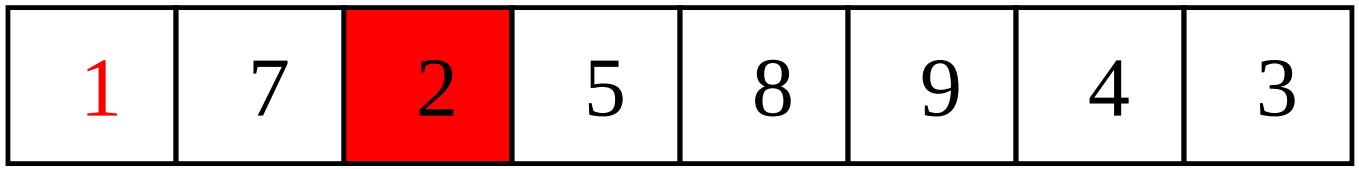
progress
status
convergent
when
$k \neq n$
$l \in k \mathinner{..} n$
$g(l) = \min(g[k..n])$
then
$g := g \lhd \{k \mapsto g(l)\} \lhd \{l \mapsto g(k)\}$
$k := k + 1$
$l :\in \mathbb{N}$
end

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

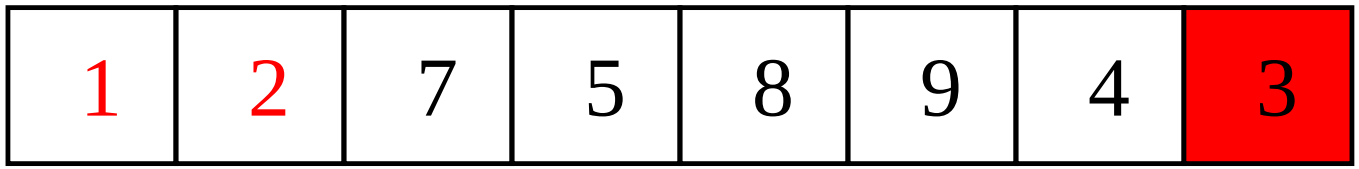
# Sorting



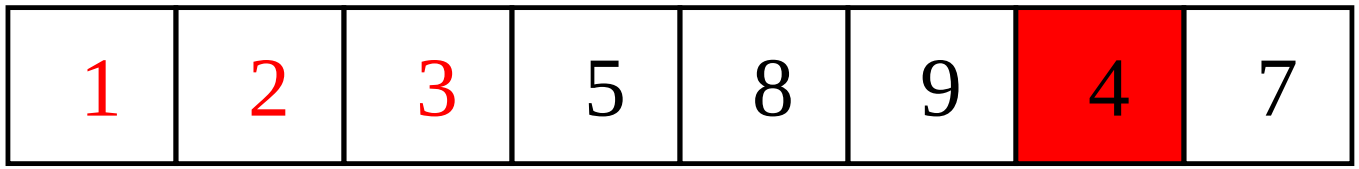
# Sorting



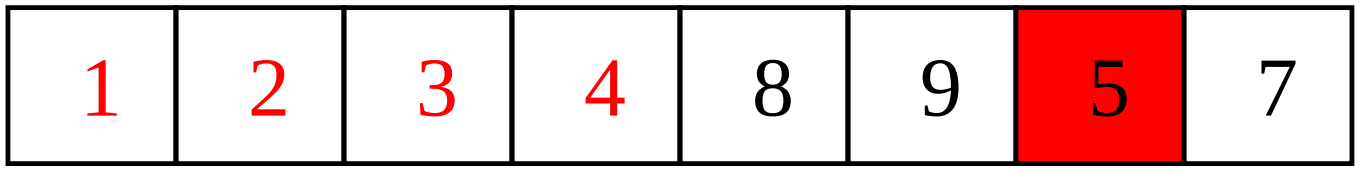
# Sorting



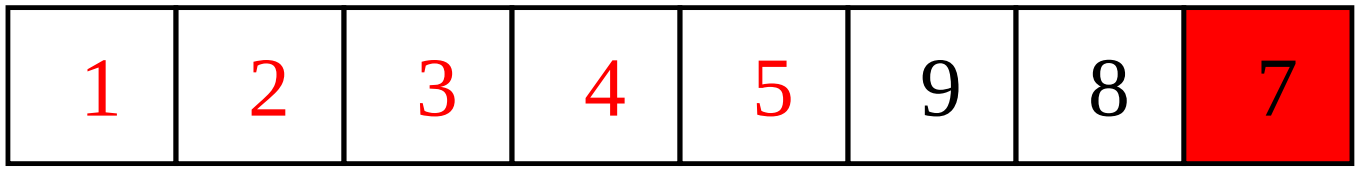
# Sorting



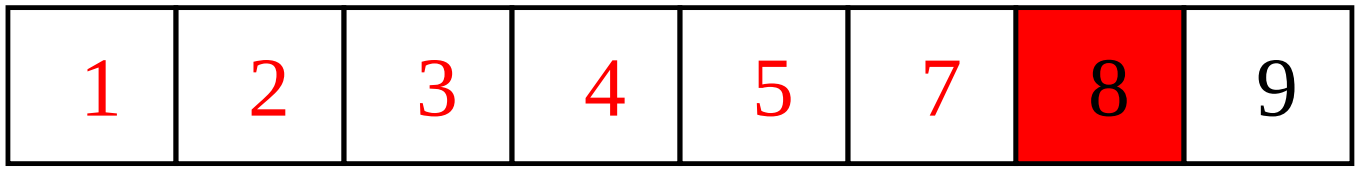
# Sorting



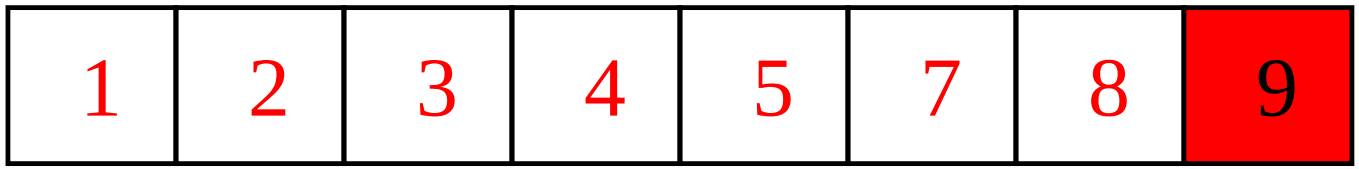
# Sorting



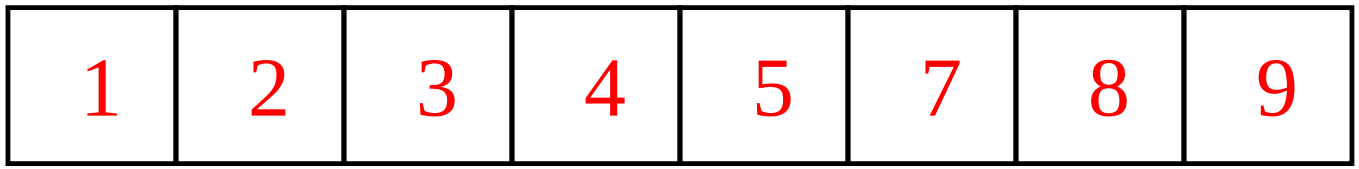
# Sorting



# Sorting



# Sorting



# Second Refinement: the state

- Introducing the variable $j$

**variables:** $g$, $k$, $l$, $j$

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

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

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

- Invariant **inv2_3** can be illustrated on the next diagram:

| | | |
|---|---|---|
| $1$ **sorted and smaller** $k-1$ | $k$ $g(l)$ **is the minimum** $j$ | $n$ |

# Second Refinement: the events (1)



- Next are the refinements of the abstract events.

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

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

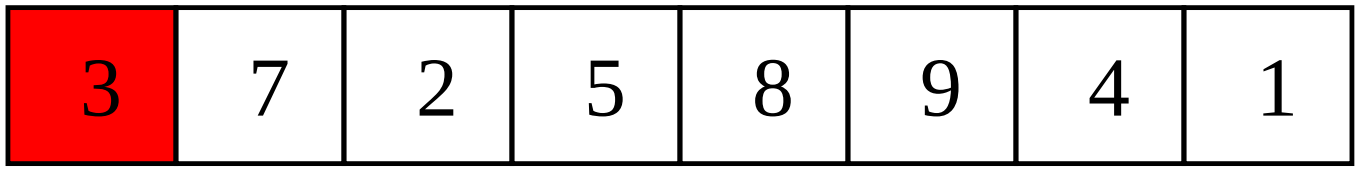
progress
**when**
$k \neq n$
$j = n$
**then**
$g := g \Leftarrow \{k \mapsto g(l)\} \Leftarrow \{l \mapsto g(k)\}$
$k := k + 1$
$j := k + 1$
$l := k + 1$
**end**

# Second Refinement: the events (2)

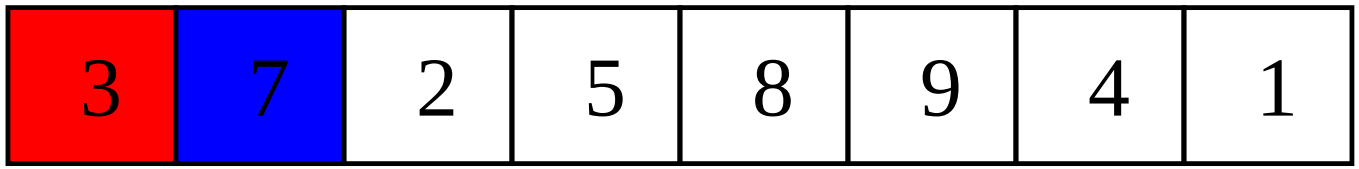
prog1
**refines**
**prog**
**status**
**convergent**
**when**
$k \neq n$
$j \neq n$
$g(l) \leq g(j+1)$
**then**
$j := j + 1$
**end**

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

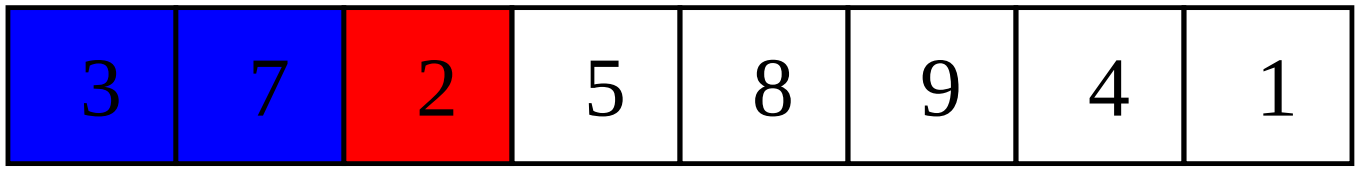
# Sorting



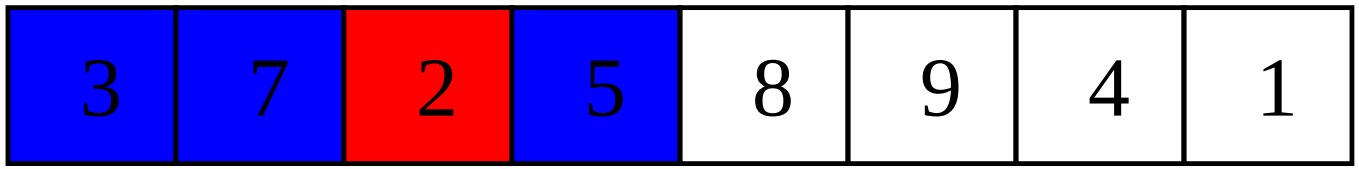
# Sorting



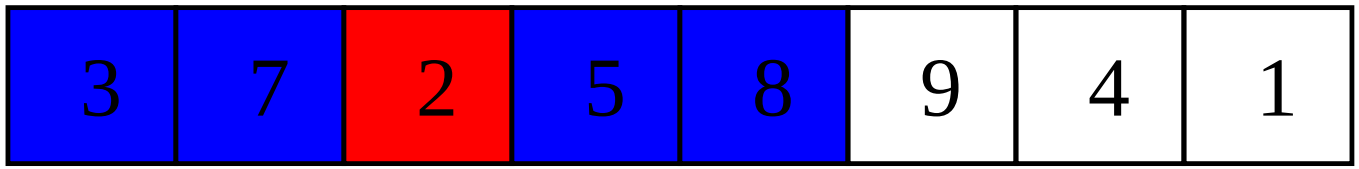
# Sorting



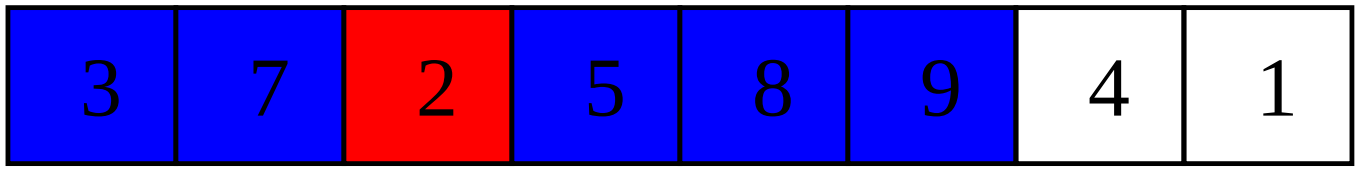
# Sorting



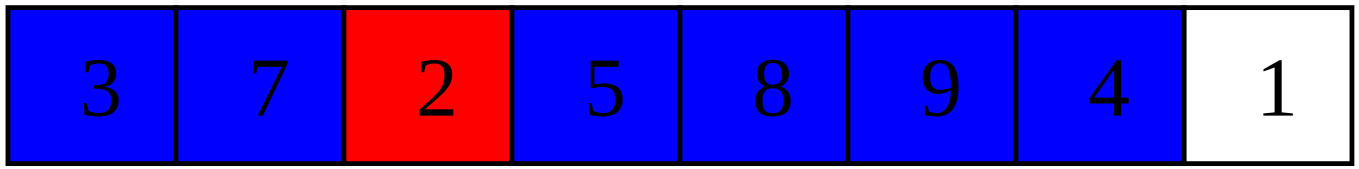
# Sorting



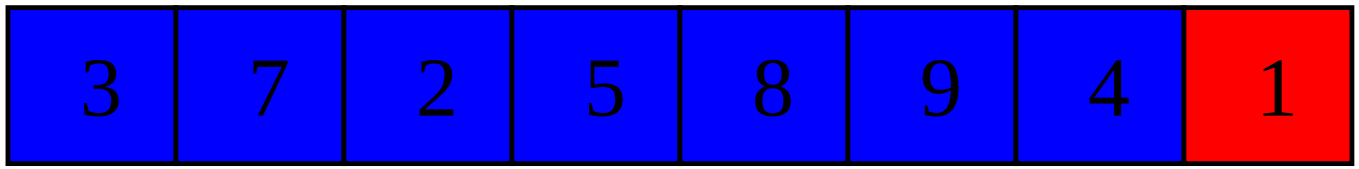
# Sorting



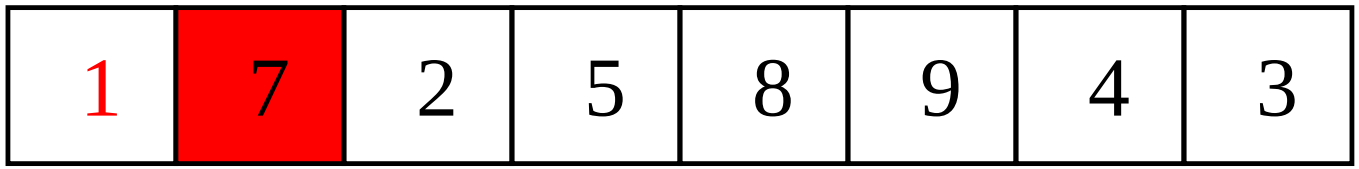
# Sorting



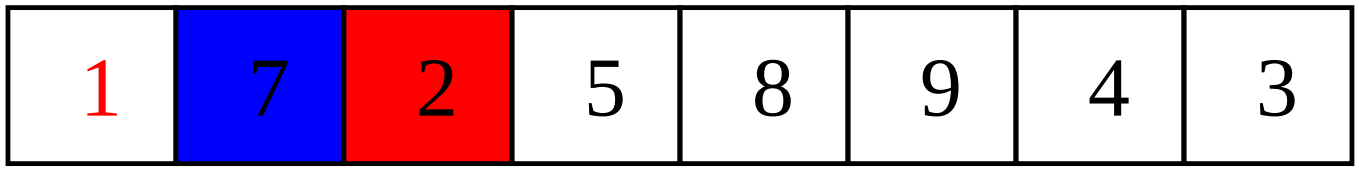
# Sorting



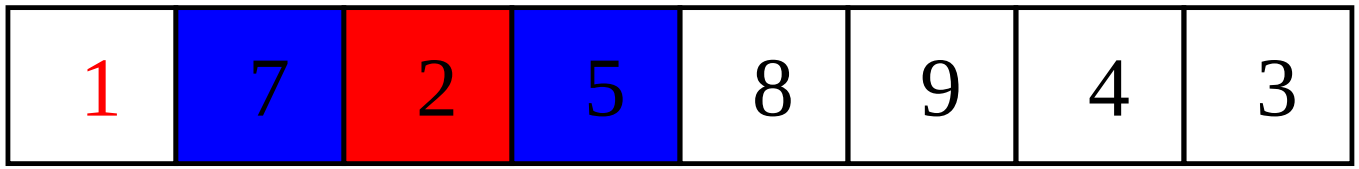
# Sorting



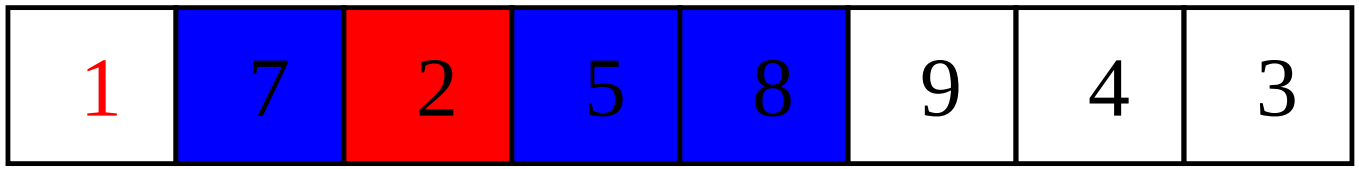
# Sorting



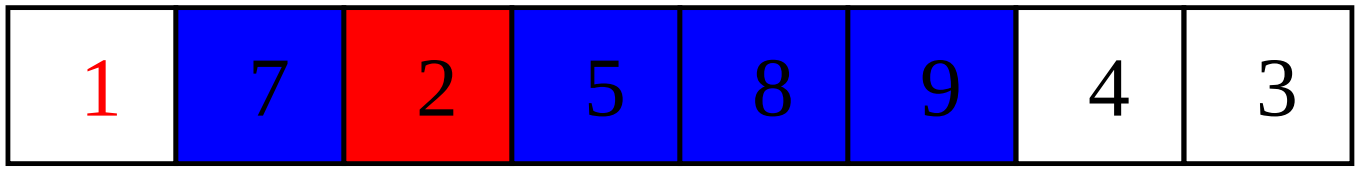
# Sorting



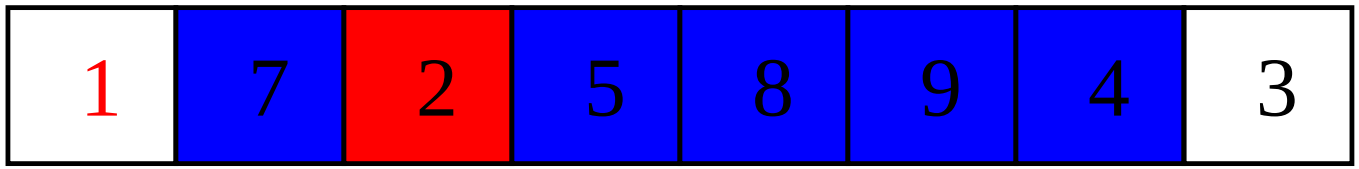
# Sorting



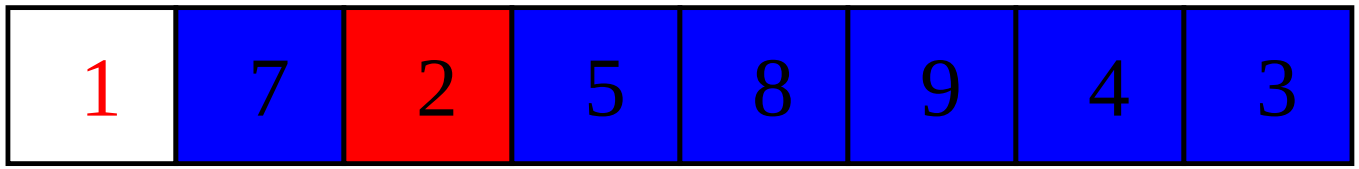
# Sorting



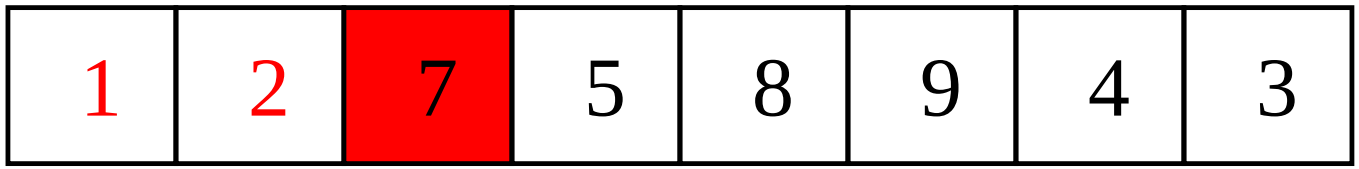
# Sorting



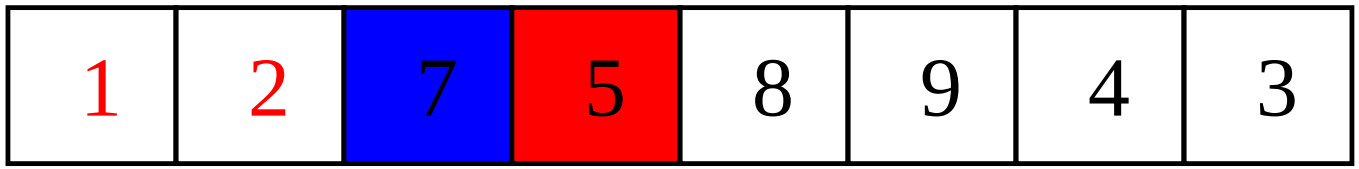
# Sorting



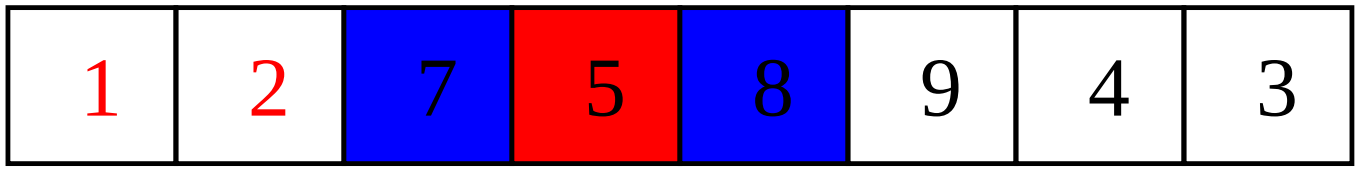
# Sorting



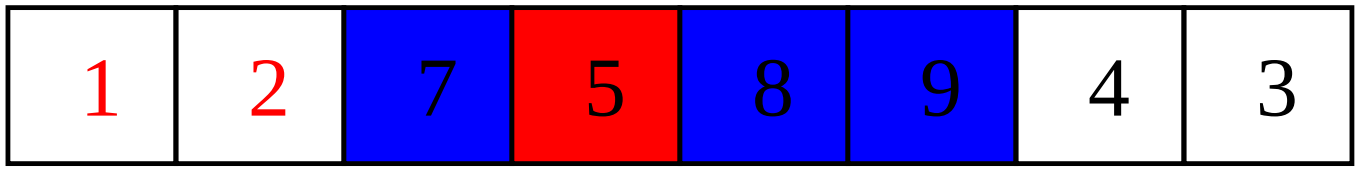
# Sorting



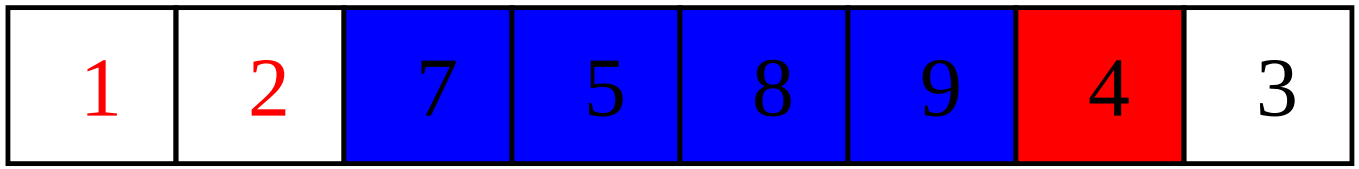
# Sorting



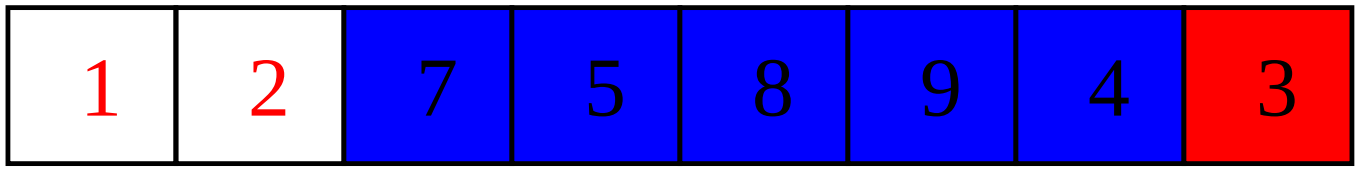
# Sorting



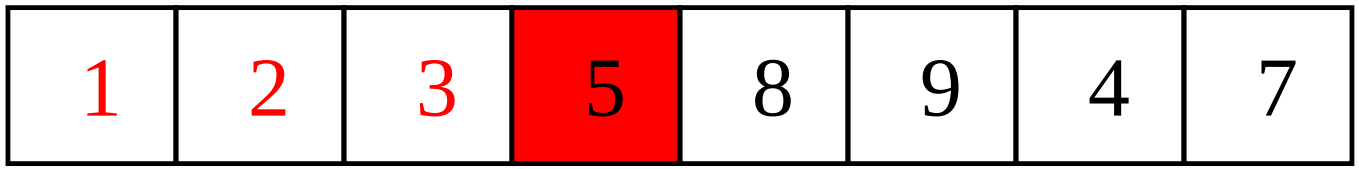
# Sorting



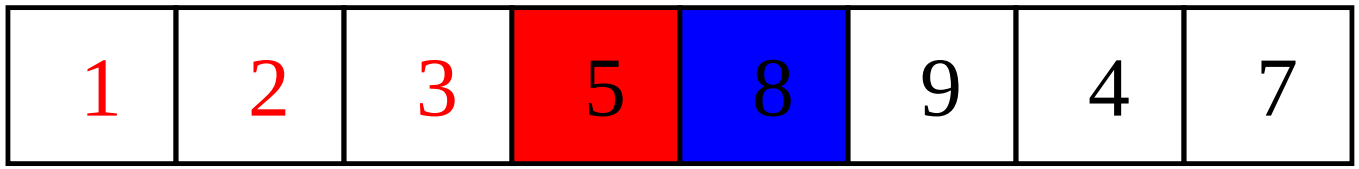
# Sorting



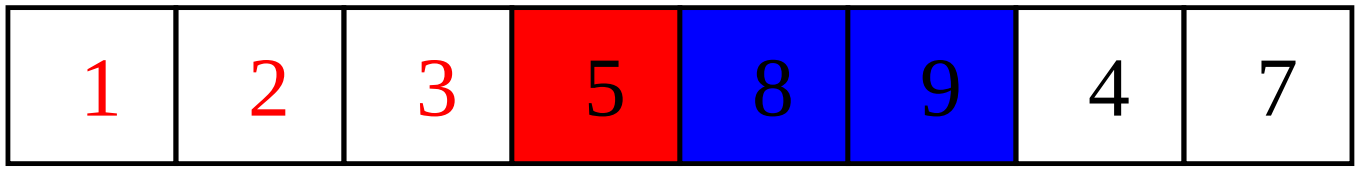
# Sorting



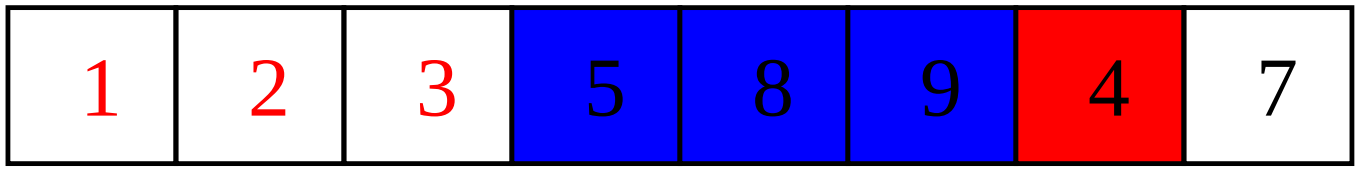
# Sorting



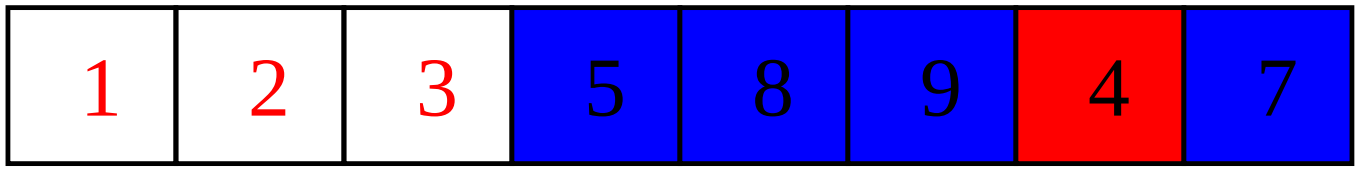
# Sorting



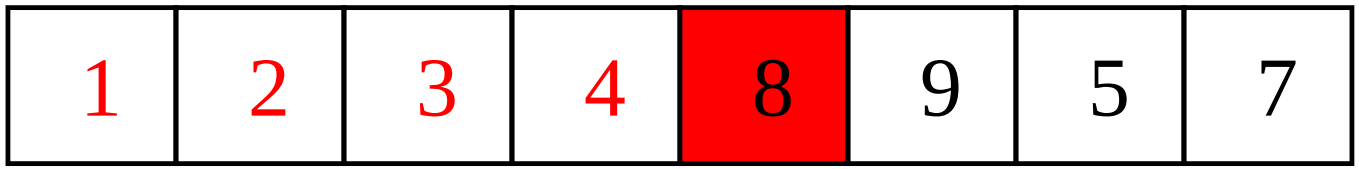
# Sorting



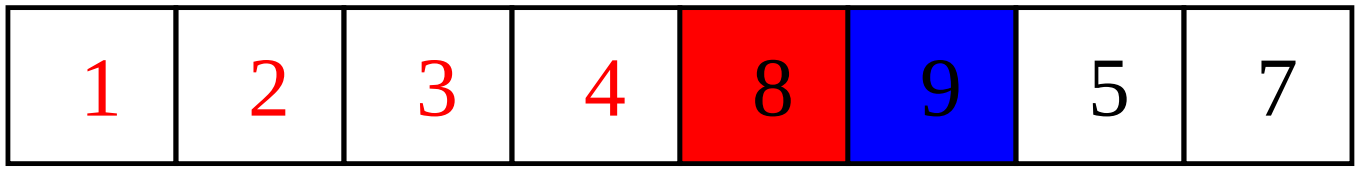
# Sorting



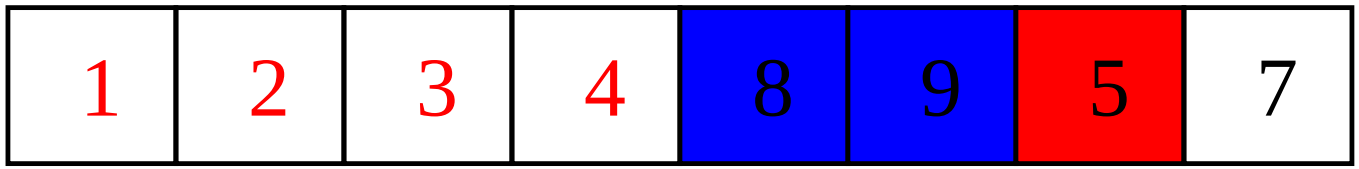
# Sorting



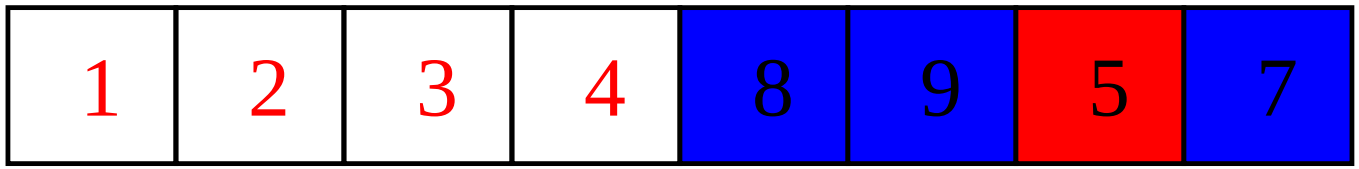
# Sorting



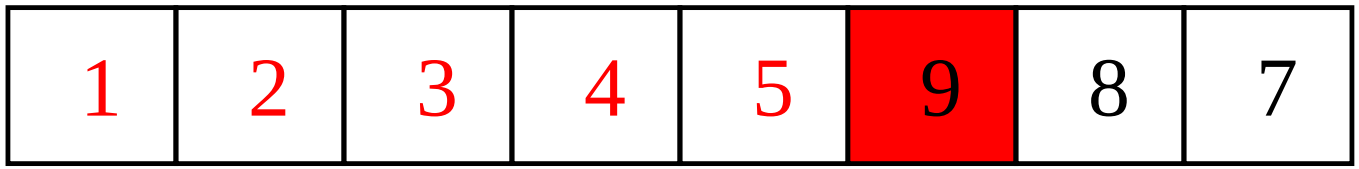
# Sorting



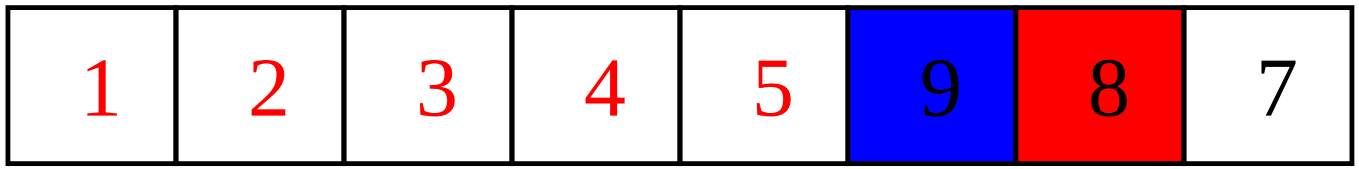
# Sorting



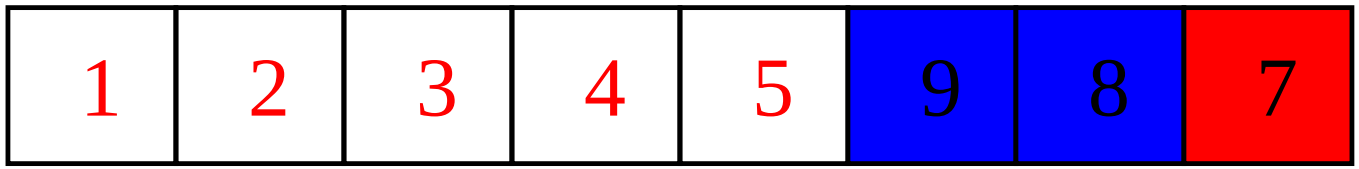
# Sorting



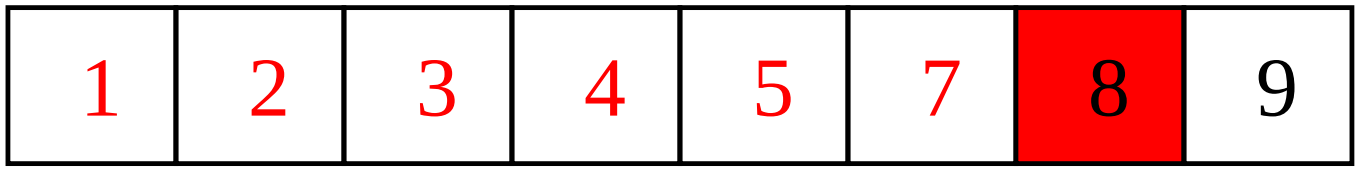
# Sorting



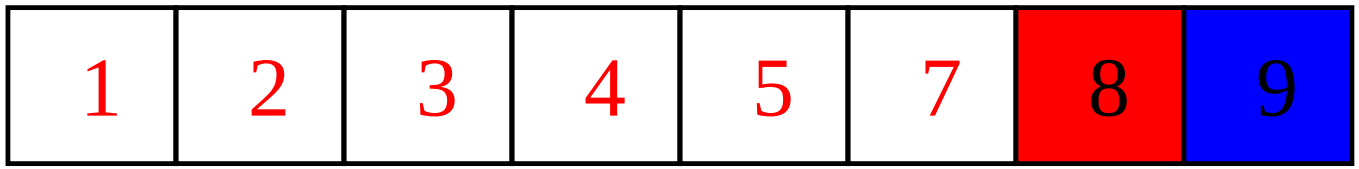
# Sorting



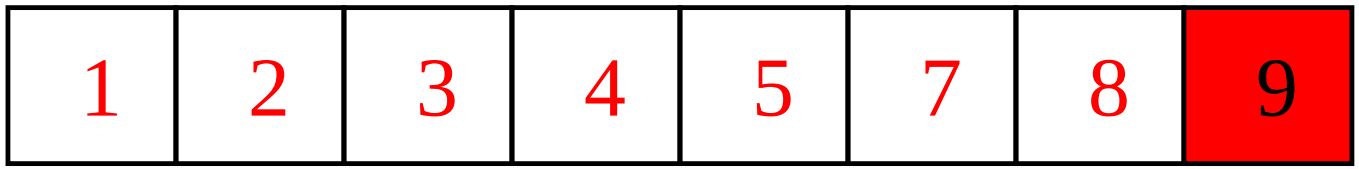
# Sorting



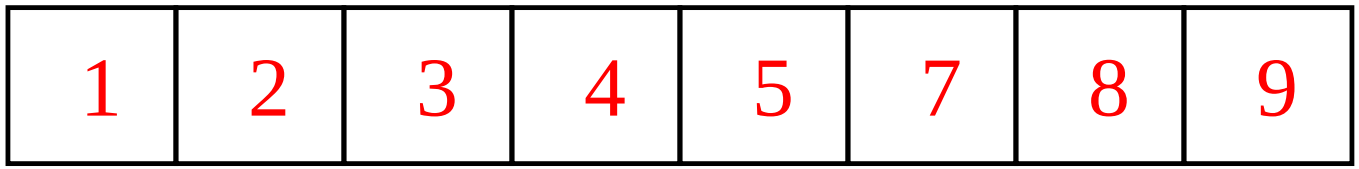
# Sorting



# Sorting



# Sorting



Applying M_IF to progr1 and progr2

progr_12
**when**
$k < n$
$j < n$
**then**
**if** $g(l) \leq g(j+1)$ **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**

$k := k + 1$

$j := k + 1$

$l := k + 1$

$g :=$ **swap** $(g, k, l)$

**end**

progr_12

**when**

$k < n$

$j < n$

**then**

**if** $g(l) \leq g(j+1)$ **then**

$j := j + 1$

**else**

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

**end**

**end**

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

Applying Rule M_WHILE to progr and progr_12

progr_progr_12
**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**;
$k, j, l, g := k + 1, k + 1, k + 1,$ **swap** $(g, k, l)$
**end**

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

progr_progr_12
**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**;
$k, j, l, g := k + 1, k + 1, k + 1,$ **swap** $(g, k, l)$
**end**

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

Applying Rule M_WHILE to sort and progr_progr_12

```
sort_progr_progr_12
  while k < n do
    while j < n do
      if h(l) ≤ h(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, swap (g, k, l)
  end
```

# Merging init and sort_progr_progr_12

init

$g := f$

$k := 1$

$j := 1$

$l := 1$

sort_progr_progr_12

**while** $k < n$ **do**

**while** $j < n$ **do**

**if** $g(l) \leq g(j+1)$ **then**

$j := j + 1$

**else**

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

**end**

**end**;

$k, j, l, g := k + 1, k + 1, k + 1,$ **swap** $(g, k, l)$

**end**

# Final Program: Applying Rule M_INIT

sort_program
**begin**
$g, k, j, l := f, 1, 1, 1$ ; init
**while** $k < n$ **do**
**while** $j < n$ **do**
**if** $g(l) \leq g(j+1)$ **then**
$j := j + 1$ progr_1
**else**
$j, l := j + 1, j + 1$ progr_2
**end**
**end**;
$k, j, l, g := k + 1, k + 1, k + 1,$ **swap** $(g, k, l)$ 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



**sets:** $S$

**constants:** $n, f$

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

**axm0_2:** $0 < n$

**axm0_3:** $f \in 1 \mathinner{..} n \to S$

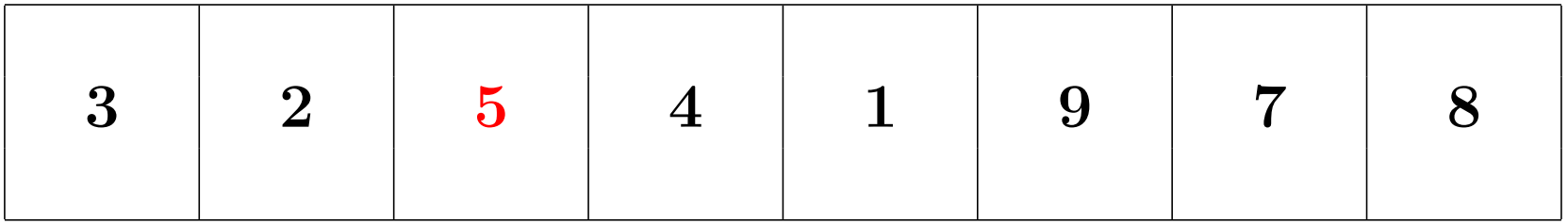
**variables:** $g$

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

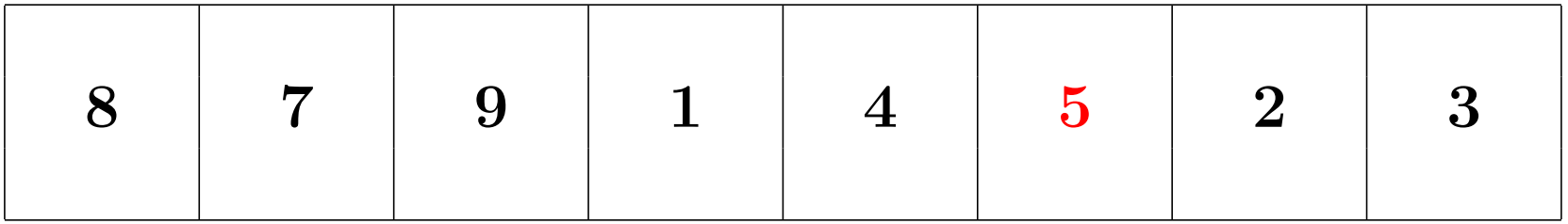
# In Place Reversing of an Array: Example



Here is an array




Here is the reverse array




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

# In Place Reversing of an Array: Events



init

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

final

**when**

$g \in 1 \mathinner{..} n \rightarrow S$

$\forall k \cdot k \in 1 \mathinner{..} n \Rightarrow g(k) = f(n - k + 1)$

**then**

skip

**end**

progress

**status**

anticipated

**then**

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

**end**

# In Place Reversing of an Array: Refinement

- 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:** $g \in 1 \mathinner{..} n \to S$

**inv1_2:** $i \in 1 \mathinner{..} n$

**inv1_3:** $j \in 1 \mathinner{..} n$

**inv1_4:** $i + j = n + 1$

**inv1_5:** $i \leq j + 1$

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

**inv1_7:** $\forall k \cdot k \in i \mathinner{..} j \Rightarrow g(k) = f(k)$

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

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

final
**when**
$j \leq i$
**then**
**skip**
**end**

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

# Final Program

- All this leads to the following final program:

reverse_program
  $i, j, g := 1, n, f;$
  **while** $i < j$ **do**
    $i, j, g := i + 1, j - 1,$ **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 a taken in a set $S$

# Formalizing the Linear Chain

**sets:** $S$

**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:** $c \in d \setminus \{l\} \rightarrowtail\!\!\!\!\!\rightarrow d \setminus \{f\}$

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

# Reversing the Chain

- Given the following initial chain


- Then the transformed chain should look like this:

# Initial Model: the Events

**inv0_1:** $r \in S \leftrightarrow S$

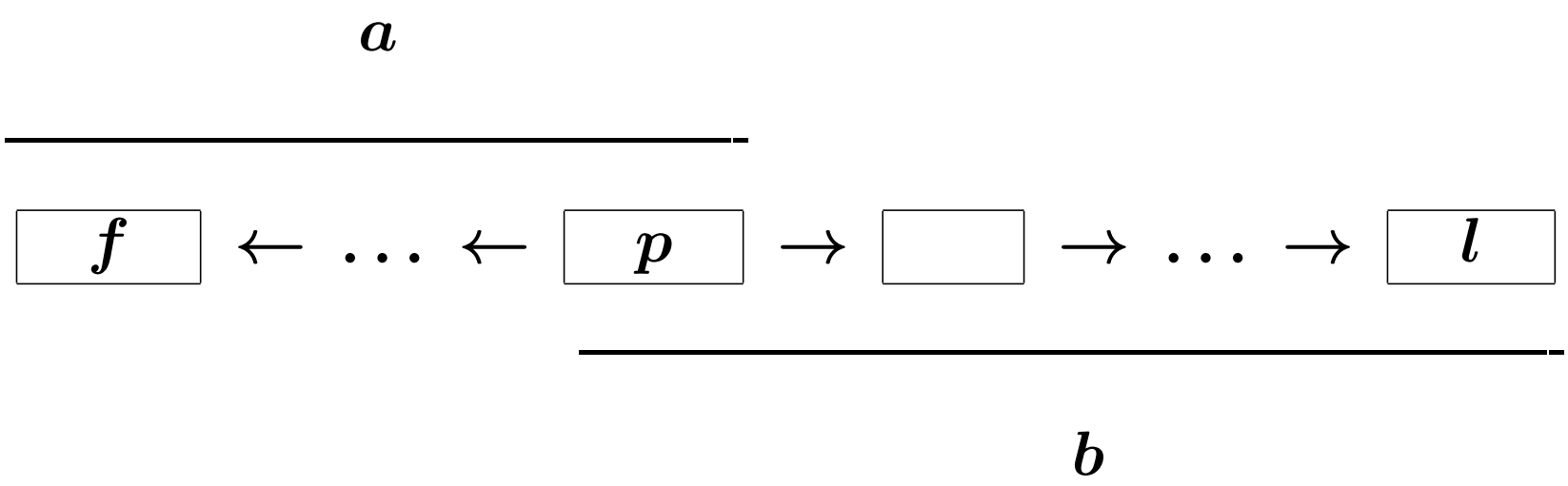
init

$r :\in S \leftrightarrow S$

reverse

$r := c^{-1}$

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



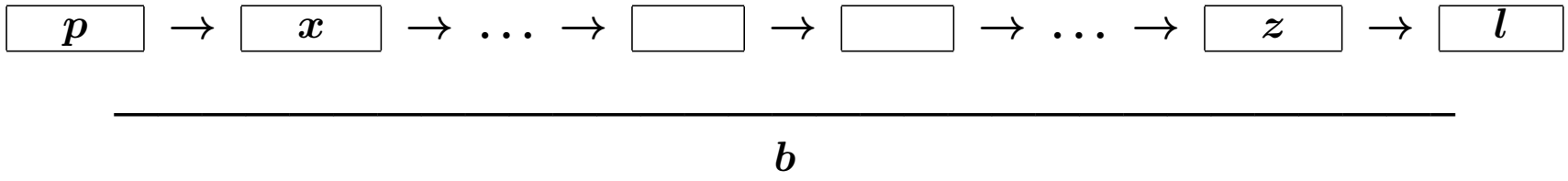

- Node $p$ starts both chains

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

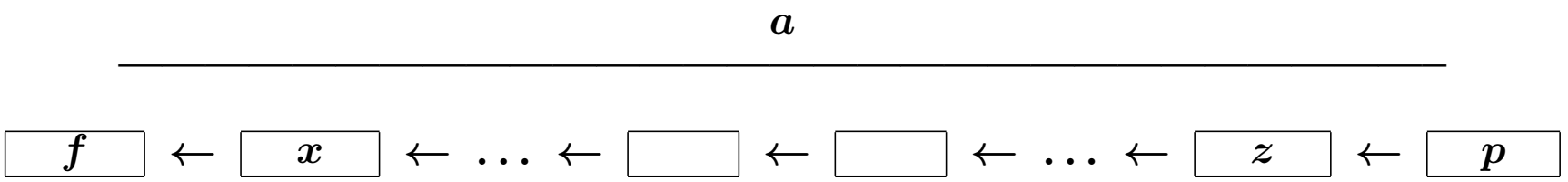
# Initial and Final Situation

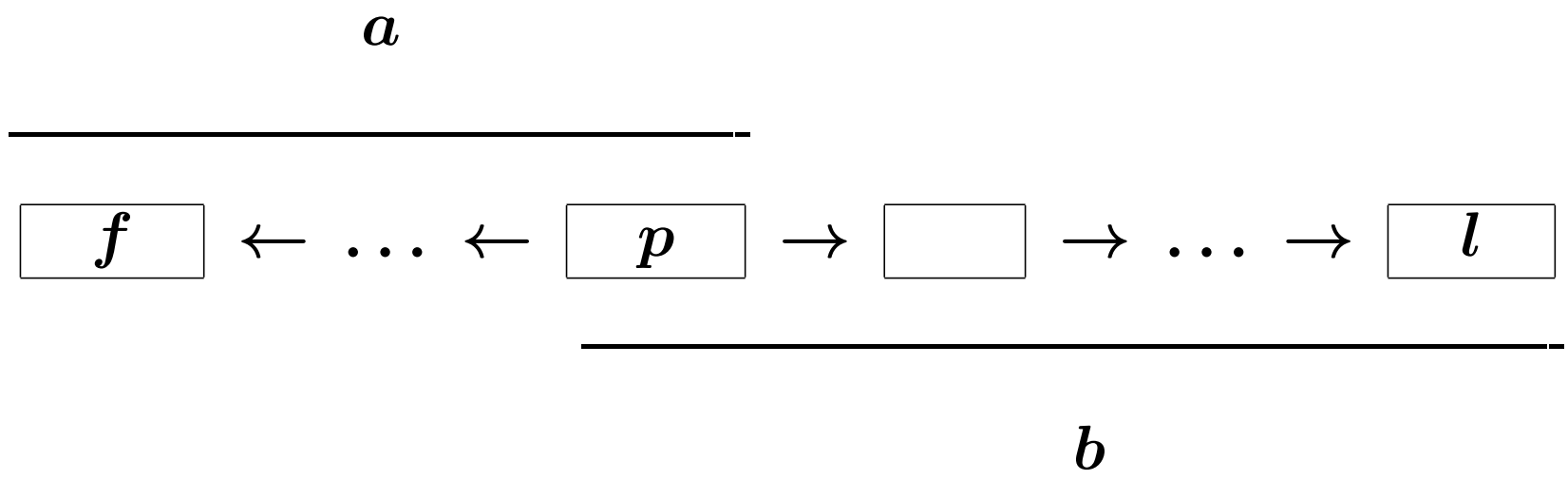


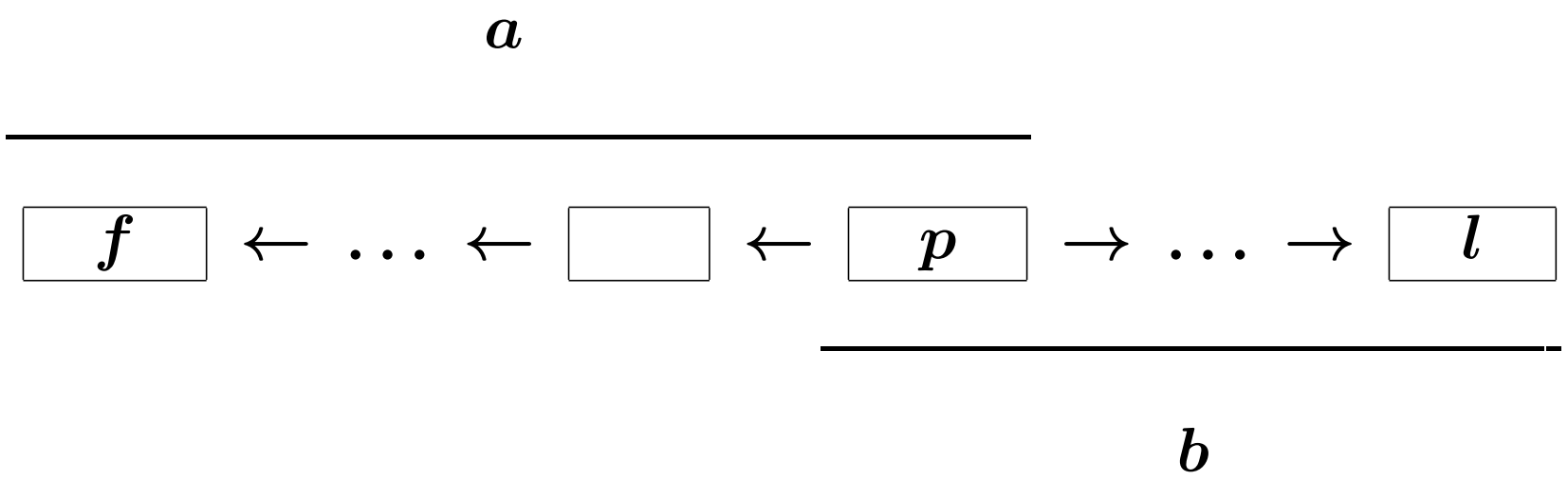
- At the beginning, $p$ is equal to $f$, $a$ is empty, and $b$ is equal to $c$:




- At the end, $p$ is equal to $l$, $a$ is the reversed chain, and $b$ is empty:




a
f
p
l
b


a
f
p
l
b

**variables:** $r$
$a$
$b$
$p$

**inv1_1:** $p \in d$

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

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

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

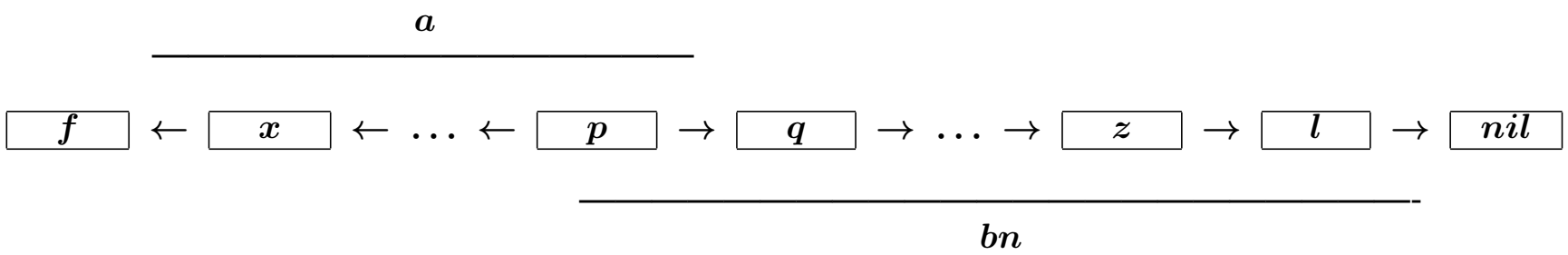
progress
**when**
$p \in \mathrm{dom}(b)$
**then**
$p := b(p)$
$a(b(p)) := p$
$b := \{p\} \lhd b$
**end**

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

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

# Second Refinement: the State (1)

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



# Second Refinement: the State (1)

- Here is the new state:

**constants:** $f, l, c, nil$

**variables:** $r, a, bn, p, q$

**axm2_1:** $nil \in S$

**axm2_2:** $nil \notin d$

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

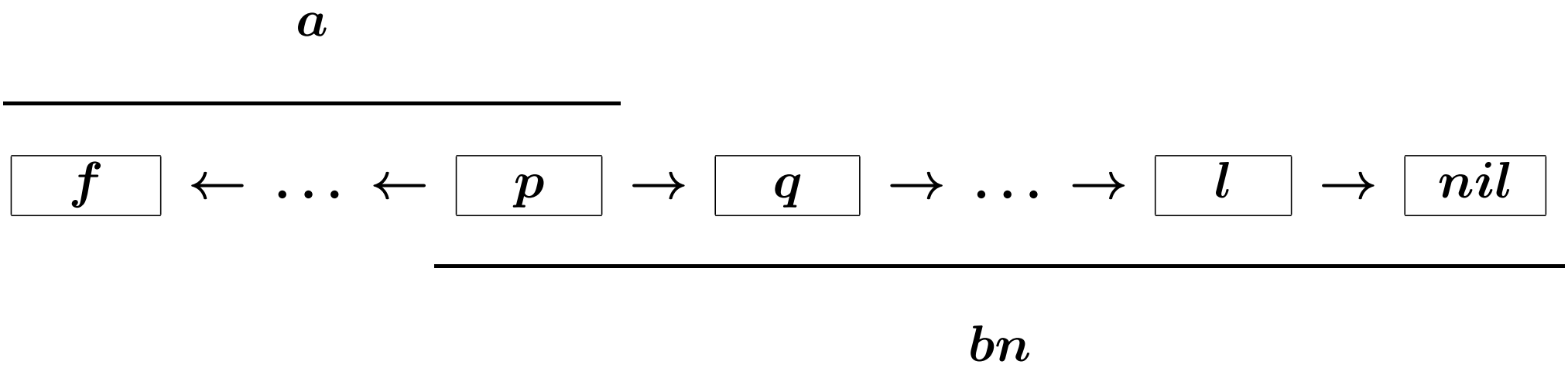
**inv2_2:** $q = bn(p)$

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

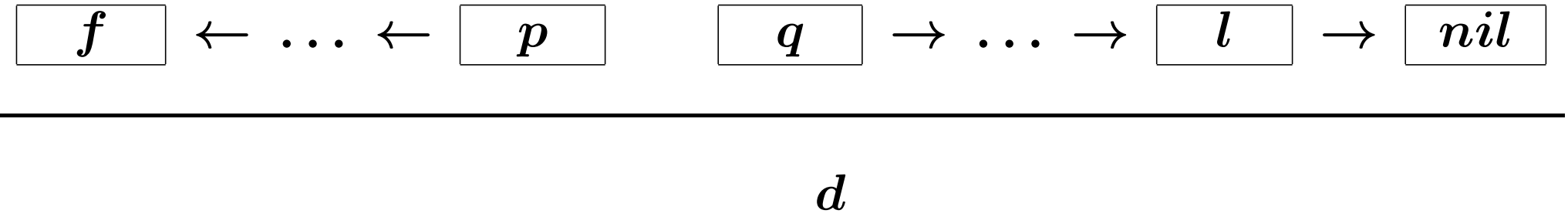
reverse
**when**
$q = nil$
**then**
$r := a$
**end**

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 $bn$




- The new situation with a single chain $d$



**carrier set:** $S$

**constants:** $f, l, c$

**variables:** $r, p, q, d$

**inv3_1:** $d \in S \pfun S$

**inv3_2:** $d = (\{f\} \domsub bn) \domminus a$

# Third Refinement: the Events

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

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

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

reverse_program

$p, q, d := f, c(f), \{f\} \lhd (c \cup \{l \mapsto nil\})$;
**while** $q \neq nil$ **do**
$p := q$
$d(q) := p$
$q := d(q)$
**end**;
$r := d \rhd \{nil\}$

# 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

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

# 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:** $n$

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

**variables:** $r$

**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**

# First Refinement

**variables:** $r$

**inv1_1:** $r^2 \leq n$

init
$r := 0$

final
**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

square_root_program

$r := 0;$

**while** $(r+1)^2 \leq n$ **do**

$r := r + 1$

**end**

# Second Refinement

- 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 ≤ 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:** $n$, $f$

**axm0_1:** $f \in \mathbb{N} \to \mathbb{N}$

**axm0_2:** $\forall i, j \cdot i \in \mathbb{N} \land j \in \mathbb{N} \land i < j \Rightarrow f(i) < f(j)$

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

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

**variables:** $r$

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

# Inverse of an Injective Numerical Function: the Events

init

$r :\in \mathbb{N}$

final

**when**

$f(r) \leq n$

$n < f(r+1)$

**then**

**skip**

**end**

progress

**status**

**anticipated**

**then**

$r :\in \mathbb{N}$

**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 $a \,..\, b$

- We try to make this interval narrower

- We introduce a constant $q$ in $a \,..\, b$ and such that

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

# First Refinement: the State

**constants:** $f, n, a, b$

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

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

**axm1_3:** $f(a) \leq n$

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

**variables:** $r, p, q$

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

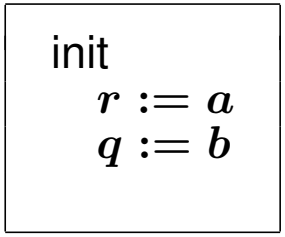
**inv1_2:** $r \leq q$

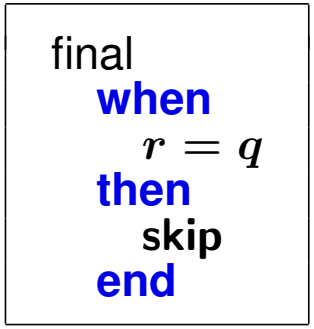
**inv1_3:** $f(r) \leq n$

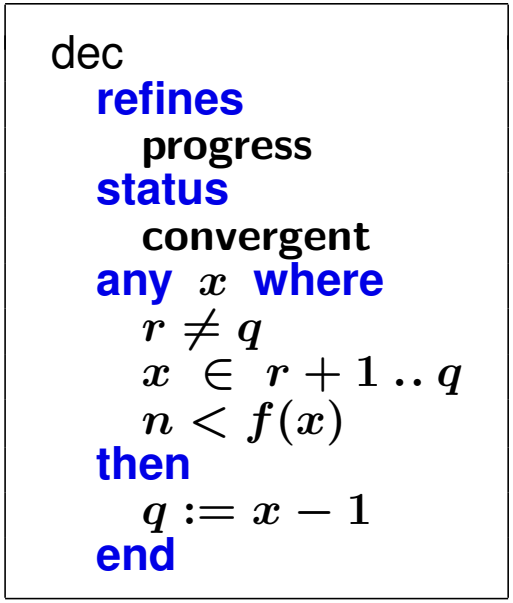
**inv1_4:** $n < f(q+1)$

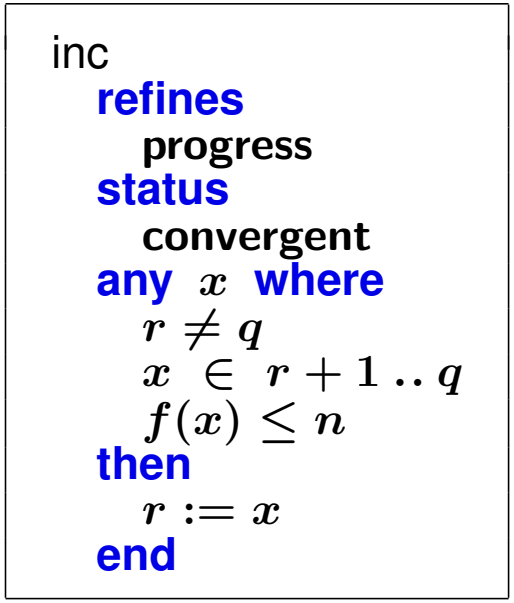
# First Refinement: the Events (1)





# 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**
$r \neq q$
$n < f((r+1+q)/2)$
**with**
$x = (r+1+q)/2$
**then**
$q := (r+1+q)/2 - 1$
**end**

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

- In order to prove this refinement the following theorem can be useful:

**thm2_1:** $$\forall x, y \cdot \; x \in \mathbb{N} \;\; y \in \mathbb{N} \;\; x \leq y \;\; \Rightarrow \;\; (x+y)/2 \in x \mathinner{..} y$$

# Final Program

inverse_program

$r, q := a, b;$

**while** $r \neq q$ **do**

**if** $n < f((r + 1 + q)/2)$ **then**

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

**else**

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

**end**

**end**

- 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

```
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;
  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

$r, q := 0, n;$

**while** $r \neq q$ **do**

**if** $n < m \times (r + 1 + q)/2)$ **then**

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

**else**

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

**end**

**end**;

$r := p$