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 (**;**)

**Introduction to Sequential Program Development**

---

2
while \( j \neq m \) do
  if \( g(j + 1) > x \) then
    \( j := j + 1 \)
  elseif \( k = j \) then
    \( k, j := k + 1, j + 1 \)
  else
    \( k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1) \)
  end
end

\( p := k \)
while condition do statement end

if condition then statement else statement end

if condition then statement elsif ... else statement end

statement ; statement

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

- This approach favors:
  - the **distribution** of computation
  - over its **centralization**
- Each individual assignment is formalized by a **guarded event** made of:
  - A **firing condition**: the guard,
  - An **action**: the multiple assignment.

- These events are scheduled **implicitly**.
while $j \neq m$ do
  if $g(j + 1) > x$ then
    $j := j + 1$
  elseif $k = j$ then
    $k, j := k + 1, j + 1$
  else
    $k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1)$
  end
end;

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

when
  \( j \neq m \)
  \( g(j + 1) \leq x \)
  \( k = j \)
then
  \( k, j := k + 1, j + 1 \)
end
while $j \neq m$ do
  if $g(j + 1) > x$ then
    $j := j + 1$
  elseif $k = j$ then
    $k, j := k + 1, j + 1$
  else
    $k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1)$
  end
end

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

when $j = m$ then
    $p := k$
end
The Various Events of our Program

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

when
\[ j \neq m \]
\[ g(j + 1) \leq x \]
\[ k \neq j \]
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
- We have just decomposed a program into separate events

- Our approach will consists in doing the reverse operation

- We shall construct the events first

- And then compose our program from these events
Principles of the Event Approach

- **Specification Phase**
  - Initial event: Specification

- **Design Phase**
  - New events: Refinements

- **Merging Phase**
  - Final event: Program
Sequential Programs are usually specified by means of:

- A pre-condition
- and a post-condition

It is represented with a Hoare-triple

$\{Pre\} \mathbf{P} \{Post\}$
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: \(0 < n\)
- 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: \( v \in \text{ran}(f) \)
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: \( v \in \text{ran}(f) \)

- We are looking for (Post-condition)
- We are given (Pre-condition)
  - a natural number \( n \): \( n \in \mathbb{N} \)
  - \( n \) is positive: \( 0 < n \)
  - an array \( f \) of \( n \) elements built on a set \( S \): \( f \in 1..n \rightarrow S \)
  - a value \( v \) known to be in the array: \( v \in \text{ran}(f) \)

- We are looking for (Post-condition)
  - an index \( r \) in the domain of the array: \( r \in \text{dom}(f) \)
Example 1: The search Program

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

- We are looking for (Post-condition)
  - an index $r$ in the domain of the array: $r \in \text{dom}(f)$
  - such that $f(r) = v$
Example 1: The search Program

- We are given (Pre-condition)
  - a natural number \( n: n \in \mathbb{N} \)
  - \( n \) 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: \( v \in \text{ran}(f) \)

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

\[
\begin{align*}
\{ n \in \mathbb{N} \\
0 < n \\
f \in 1..n \rightarrow S \\
v \in \text{ran}(f) \}
\end{align*}
\text{search}\quad\begin{align*}
\{ r \in \text{dom}(f) \\
f(r) = v \}
\end{align*}
- 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

\[
\left\{ \begin{array}{l}
n \in \mathbb{N} \\
0 < n \\
f \in 1 \ldots n \rightarrow S \\
v \in \text{ran}(f)
\end{array} \right. \quad \text{search} \quad \left\{ \begin{array}{l}
r \in \text{dom}(f) \\
f(r) = v
\end{array} \right.
\]
Encoding a Hoare-triple in an Event System

\[
\begin{aligned}
\begin{cases}
  n \in \mathbb{N} \\
  0 < n \\
  f \in 1 \ldots n \rightarrow S \\
  v \in \text{ran}(f)
\end{cases}
\end{aligned}
\]
\[
\begin{aligned}
\text{search} \quad \begin{cases}
  r \in \text{dom}(f) \\
  f(r) = v
\end{cases}
\end{aligned}
\]

carrier sets: \( S \)

constants: \( n, f, v \)

variables: \( r \)

\text{axm0}\_1: \quad n \in \mathbb{N}
\text{axm0}\_2: \quad 0 < n
\text{axm0}\_3: \quad f \in 1 \ldots n \rightarrow S
\text{axm0}\_4: \quad v \in \text{ran}(f)

\text{inv0}\_1: \quad r \in \mathbb{N}
Encoding a Hoare-triple in an Event System

\[
\begin{align*}
\{ n \in \mathbb{N} \\
0 < n \\
f \in 1 \ldots n \rightarrow S \\
v \in \text{ran}(f)
\} \quad \text{search} \quad \{ r \in \text{dom}(f) \\
f(r) = v \}
\end{align*}
\]

**carrier sets:** \( S \)

**constants:** \( n, f, v \)

**variables:** \( r \)

**axm0.1:** \( n \in \mathbb{N} \)

**axm0.2:** \( 0 < n \)

**axm0.3:** \( f \in 1 \ldots n \rightarrow S \)

**axm0.4:** \( v \in \text{ran}(f) \)

**inv0.1:** \( r \in \mathbb{N} \)

**init**

\[
\begin{align*}
r & :\in \mathbb{N}
\end{align*}
\]

**final**

\[
\begin{align*}
\text{when} & \quad r \in \text{dom}(f) \\
f(r) = v
\end{align*}
\]

**then**

\[
\begin{align*}
\text{skip}
\end{align*}
\]

**end**

**progress**

**status**

**anticipated**

**then**

\[
\begin{align*}
r & :\in \mathbb{N}
\end{align*}
\]

**end**
Result variable $r$ is set to 1 initially
Development of the search Program: Refinement

\[
\text{inv1}_1: \quad r \in 1..n \\
\text{inv1}_2: \quad v \notin f[1..r-1]
\]

\[
\text{variant1:} \quad n - r
\]

\[
\text{init} \quad r := 1
\]

\[
\text{progress status} \quad \text{convergent} \\
\text{when} \quad f(r) \neq v \\
\text{then} \quad r := r + 1 \\
\text{end}
\]

\[
\text{final} \quad \text{when} f(r) = v \\
\text{then} \quad \text{skip} \\
\text{end}
\]
- 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.

\[
\text{init} \\
\quad r := 1
\]

\[
\text{progress} \\
\quad \begin{aligned}
\text{when} & \quad f(r) \neq v \\
\text{then} & \quad r := r + 1 \\
\text{end}
\end{aligned}
\]

\[
\text{final} \\
\quad \begin{aligned}
\text{when} & \quad f(r) = v \\
\text{then} & \quad \text{skip} \\
\text{end}
\end{aligned}
\]
Merging Rule (1)

\[
\begin{align*}
\text{when } & P \\
\text{then } & Q \\
\text{then } & S \\
\text{end} \\
\text{when } & P \\
\text{then } & \neg Q \\
\text{then } & T \\
\text{end} \\
\end{align*}
\sim
\begin{align*}
\text{when } & P \\
\text{then } & \text{while } Q \text{ do } \\
& S \\
& \text{end; } \\
& T \\
\text{end}
\end{align*}
\]

- 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)

- **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
\[ f(r) \neq v \]
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
- 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

\[
\begin{align*}
\text{init} & \quad r := 1 \\
\text{progress_final} & \quad \text{while } f(r) \neq v \text{ do} \\
& \quad \quad r := r + 1 \\
& \quad \text{end} \\
\text{search_program} & \quad \{ r \in \text{dom}(f) \} \\
& \quad \text{while } f(r) \neq v \text{ do} \\
& \quad \quad r := r + 1 \\
& \quad \text{end} \\
& \quad \{ r \in \text{dom}(f) \} \\
& \quad \quad f(r) = v
\end{align*}
\]
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: $v \in \text{ran}(f)$
- 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: \( v \in \text{ran}(f) \)

- 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: \( v \in \text{ran}(f) \)

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

- We are given (Pre-condition)
  - a natural number \( n \): \( n \in \mathbb{N} \)
  - \( n \) 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: \( v \in \text{ran}(f) \)

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

constants: \( n, f, v \)

variables: \( r \)

\textbf{inv0.1:} \( r \in \mathbb{N} \)

\textbf{axm0.1:} \( n \in \mathbb{N} \)

\textbf{axm0.2:} \( 0 < n \)

\textbf{axm0.3:} \( f \in 1 \ldots n \rightarrow \mathbb{N} \)

\textbf{axm0.4:} \( \forall i, j \cdot \begin{cases} \begin{aligned} &i \in 1 \ldots n \quad j \in 1 \ldots n \\ &i \leq j \end{aligned} \\ \Rightarrow f(i) \leq f(j) \end{cases} \)

\textbf{axm0.5:} \( v \in \text{ran}(f) \)

\textbf{init} \quad r :\in \mathbb{N}

\textbf{final}

\textbf{when} \quad r \in \text{dom}(f)

\quad f(r) = v

\textbf{then} \quad \text{skip}

\textbf{end}

\textbf{progress}

\textbf{status} \quad \text{anticipated}

\textbf{then} \quad r :\in \mathbb{N}

\textbf{end}
First Refinement: the State

constants: \( n, f, v \)

variables: \( r, p, q \)

- Current situation

\[
\begin{array}{cccccc}
1 & p -1 & r & q+1 & n \\
\hline
p & & & & v : f[p..q] & q \\
\end{array}
\]
- Situation encountered by event inc

\begin{align*}
\text{inc} & \\
\text{status} & \text{convergent} \\
\text{when} & \quad f(r) < v \\
\text{then} & \quad p := r + 1 \\
\text{end} & \quad r \in r + 1 .. q
\end{align*}

\textbf{variant1:} \quad q - p

\begin{array}{cccc}
1 & p - 1 & r & q + 1 & n \\
\hline
& & v:f[r+1..q] & \\
\hline
p & \rightarrow r + 1 & q
\end{array}
Event dec refining progress

- Situation encountered by event dec

\[
\begin{array}{c}
\text{dec} \\
\text{status} \\
\text{convergent} \\
\text{when} \\
v < f(r) \\
\text{then} \\
q := r - 1 \\
r : \in p .. r - 1 \\
\text{end}
\end{array}
\]

variant1: \( q - p \)
First Refinement: the Events

init
\[ p := 1 \]
\[ q := n \]
\[ r \in 1 .. n \]

final

when \[ f(r) = v \]
then
skip
end

inc

when \[ f(r) < v \]
then
\[ p := r + 1 \]
\[ r \in r + 1 .. q \]
end

dec

when \[ v < f(r) \]
then
\[ q := r - 1 \]
\[ r \in p .. r - 1 \]
end
- At the previous stage, \texttt{inc} and \texttt{dec} were non-deterministic.

- \( r \) was chosen \textit{arbitrarily} within the interval \( p .. q \).

- We now remove the non-determinacy in \texttt{inc} and \texttt{dec}.

- \( r \) is chosen to be the \textit{middle of the interval} \( p .. q \).
## Reducing Non-determinacy

<table>
<thead>
<tr>
<th>Abstract Inc</th>
<th>Concrete Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>when</strong></td>
<td><strong>when</strong></td>
</tr>
<tr>
<td>( f(r) &lt; v )</td>
<td>( f(r) &lt; v )</td>
</tr>
<tr>
<td><strong>then</strong></td>
<td><strong>then</strong></td>
</tr>
<tr>
<td>( p := r + 1 )</td>
<td>( p := r + 1 )</td>
</tr>
<tr>
<td>( r := { r + 1 \ldots q } )</td>
<td>( r := (r + 1 + q)/2 )</td>
</tr>
<tr>
<td><strong>end</strong></td>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract Dec</th>
<th>Concrete Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>when</strong></td>
<td><strong>when</strong></td>
</tr>
<tr>
<td>( f(r) &lt; v )</td>
<td>( f(r) &lt; v )</td>
</tr>
<tr>
<td><strong>then</strong></td>
<td><strong>then</strong></td>
</tr>
<tr>
<td>( q := r - 1 )</td>
<td>( q := r - 1 )</td>
</tr>
<tr>
<td>( r := { p \ldots r - 1 } )</td>
<td>( r := (p + r - 1)/2 )</td>
</tr>
<tr>
<td><strong>end</strong></td>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>
Second Refinement: the Events

init

\[ p, q := 1, n \]
\[ r := \frac{1 + n}{2} \]

bin_search

when \( f(r) = v \)
then
skip
end

ing

when \( f(r) < v \)
then
\[ p := r + 1 \]
\[ r := \frac{r + 1 + q}{2} \]
end

dec

when \( v < f(r) \)
then
\[ q := r - 1 \]
\[ r := \frac{p + r - 1}{2} \]
end
<table>
<thead>
<tr>
<th>when ( P )</th>
<th>when ( P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q )</td>
<td>( \neg Q )</td>
</tr>
<tr>
<td>then ( S )</td>
<td>then ( T )</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

\[
\text{when } P \text{ then if } Q \text{ then } S \text{ else } T \text{ end end}
\]

\[
\text{when } P \text{ then } T \text{ end}
\]

M_IF
Merging Events \texttt{inc} and \texttt{dec} by means of Rule M\_IF

\begin{align*}
\text{inc} \\
\text{when} & \quad f(r) \neq v \\
& \quad f(r) < v \\
\text{then} & \quad p := r + 1 \\
& \quad r := (r + 1 + q)/2 \\
\text{end} \\
\text{dec} \\
\text{when} & \quad f(r) \neq v \\
& \quad v \leq f(r) \\
\text{then} & \quad q := r - 1 \\
& \quad r := (p + r - 1)/2 \\
\text{end} \\
\text{inc\_dec} \\
\text{when} & \quad f(r) \neq v \\
\text{then} & \quad \text{if } f(r) < v \text{ then} \\
& \quad p, r := r + 1, (r + 1 + q)/2 \\
& \quad \text{else} \\
& \quad q, r := r - 1, (p + r - 1)/2 \\
\text{end} \\
\text{final} \\
\text{when} & \quad f(r) = v \\
\text{then} & \quad \text{skip} \\
\text{end}
\end{align*}
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 \texttt{inc\_dec} and \texttt{bin\_search} with Rule M\_WHILE

\texttt{inc\_dec}
\begin{align*}
&\text{\texttt{when}} \\
&\quad \quad f(r) \neq v \\
&\text{\texttt{then}} \\
&\quad \quad \text{if } f(r) < v \text{ then} \\
&\quad \quad \quad p, r := r + 1, (r + 1 + q)/2 \\
&\quad \quad \quad \text{else} \\
&\quad \quad \quad q, r := r - 1, (p + r - 1)/2 \\
&\text{\texttt{end}} \\
&\text{\texttt{end}}
\end{align*}

\texttt{inc\_dec\_final}
\begin{align*}
&\text{\texttt{while}} \quad f(r) \neq v \quad \texttt{do} \\
&\quad \quad \text{if } f(r) < v \text{ then} \\
&\quad \quad \quad p, r := r + 1, (r + 1 + q)/2 \\
&\quad \quad \quad \text{else} \\
&\quad \quad \quad q, r := r - 1, (p + r - 1)/2 \\
&\quad \quad \texttt{end} \\
&\text{\texttt{end}}
\end{align*}

\texttt{final}
\begin{align*}
&\text{\texttt{when}} \\
&\quad \quad f(r) = v \\
&\text{\texttt{then}} \\
&\quad \quad \texttt{skip} \\
&\text{\texttt{end}}
\end{align*}

\texttt{init}
\begin{align*}
&\quad p, q := 1, n \\
&\quad r := (1 + n)/2
\end{align*}
Merging Events \texttt{inc\_dec\_bin\_search} and \texttt{init} with Rule M\_INIT

\begin{verbatim}
imc_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

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

init
p, q := 1, n
r := (1 + n)/2
\end{verbatim}
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.
- $g$ 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:

\[
\begin{array}{cccccccc}
3 & 7 & 2 & 5 & 8 & 9 & 4 & 1 \\
\end{array}
\]

- Let \( x \) be equal to 5

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

\[
\begin{array}{cccccccc}
3 & 2 & 5 & 4 & 1 & 9 & 7 & 8 \\
\end{array}
\]
- Let the array $f$ be the following:

$$
\begin{array}{ccccccccc}
3 & 7 & 2 & 5 & 8 & 9 & 4 & 1 \\
\end{array}
$$

- Let $x$ be equal to 0

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

$$
\begin{array}{ccccccccc}
3 & 7 & 2 & 5 & 8 & 9 & 4 & 1 \\
\end{array}
$$
- Let the array $f$ be the following:

$$
\begin{array}{cccccccc}
3 & 7 & 2 & 5 & 8 & 9 & 4 & 1 \\
\end{array}
$$

- Let $x$ be equal to 10

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

$$
\begin{array}{cccccccc}
3 & 7 & 2 & 5 & 8 & 9 & 4 & 1 \\
\end{array}
$$
Array Partitioning: the Initial State

constants:  \( n, f, x \)

variables:  \( k, g \)

\[
\begin{align*}
\text{axm0\_1: } & \quad n \in \mathbb{N} \\
\text{axm0\_2: } & \quad f \in 1..n \mapsto \mathbb{N} \\
\text{axm0\_3: } & \quad x \in \mathbb{N}
\end{align*}
\]

\[
\begin{align*}
\text{inv0\_1: } & \quad k \in \mathbb{N} \\
\text{inv0\_2: } & \quad g \in \mathbb{N} \leftrightarrow \mathbb{N}
\end{align*}
\]
Array Partitioning: the Initial Events

init
\[ k \in \mathbb{N} \]
\[ g :\in \mathbb{N} \leftrightarrow \mathbb{N} \]

final

when
\[ k \in 0 \ldots n \]
\[ g \in 1 \ldots n \rightarrow \mathbb{N} \]
\[ \text{ran} \,(g) = \text{ran} \,(f) \]
\[ \forall l \cdot l \in 1 \ldots k \Rightarrow g(l) \leq x \]
\[ \forall l \cdot l \in k + 1 \ldots n \Rightarrow g(l) > x \]

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 $j$

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

Invariant

$$k \leq j$$

$$\forall l \cdot l \in 1 \ldots k \Rightarrow g(l) \leq x$$

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

\[
\begin{array}{cccccccc}
3 & 7 & 2 & 5 & 8 & 9 & 4 & 1 \\
\end{array}
\]
Partitioning with 5

\[
\begin{array}{cccccccc}
3 & 7 & 2 & 5 & 8 & 9 & 4 & 1 \\
\end{array}
\]
Partitioning with 5

\[
\begin{array}{ccccccccc}
3 & 7 & 2 & 5 & 8 & 9 & 4 & 1 \\
\end{array}
\]
Partitioning with 5

\[
\begin{array}{cccccccc}
3 & 2 & 7 & 5 & 8 & 9 & 4 & 1 \\
\end{array}
\]
Partitioning with 5

| 3 | 2 | 5 | 7 | 8 | 9 | 4 | 1 |
Partitioning with 5

\[
\begin{array}{cccccccc}
3 & 2 & 5 & 7 & 8 & 9 & 4 & 1 \\
\end{array}
\]
Partitioning with 5

\[ \begin{array}{ccccccc}
3 & 2 & 5 & 7 & 8 & 9 & 4 & 1
\end{array} \]
Partitioning with 5

\[
\begin{array}{cccccccc}
3 & 2 & 5 & 4 & 8 & 9 & 7 & 1 \\
\end{array}
\]
Partitioning with 5

\[
\begin{array}{cccccc}
3 & 2 & 5 & 4 & 1 & 9 & 7 & 8 \\
\end{array}
\]
Init
\[ g, j, k := f, 0, 0 \]

Final
\begin{align*}
&\text{when } j = n \\
&\text{then } \text{skip} \\
&\text{end}
\end{align*}
| 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

variant 1: \( n - j \)
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: \( n - j \)
\[
\begin{array}{cccccc}
1 & \leq x & k & k + 1 & > x & j & j + 1 & \text{?} & n \\
\end{array}
\]

\[
\text{progress}_3
\]

\[
\text{progress}
\]

\[
\text{sattus}
\]

\[
\text{convergent}
\]

\[
\text{when}
\]

\[
j \neq n
\]

\[
g(j + 1) \leq x
\]

\[
k \neq j
\]

\[
\text{then}
\]

\[
k, j, g := k + 1, j + 1, \]

\[
\text{swap} (g, k + 1, j + 1)
\]

\[
\text{end}
\]

\[
\text{variant1: } n - j
\]

\[
\text{swap} (g, k, j) = g \Leftrightarrow \{k \mapsto g(j)\} \Leftrightarrow \{j \mapsto g(k)\}
\]
Partitioning with 5

\[ \begin{array}{cccccccc}
3 & 2 & 5 & 7 & 8 & 9 & 4 & 1 \\
\end{array} \]
Partitioning with 5

3 2 5 4 8 9 7 1
Putting together progress_2 and progress_3

progress_2
  when
      j ≠ n
      g(j + 1) ≤ x
      k = j
  then
      k, j := k + 1, j + 1
  end

progress_3
  when
      j ≠ n
      g(j + 1) ≤ x
      k ≠ j
  then
      k, j, g := k + 1, j + 1, swap (g, k + 1, j + 1)
  end
Merging Rule (2)

<table>
<thead>
<tr>
<th>when</th>
<th>when</th>
<th>when</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>$P$</td>
<td>$P$</td>
</tr>
<tr>
<td>$Q$</td>
<td>$¬Q$</td>
<td>if $Q$ then</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
<td>then</td>
</tr>
<tr>
<td>$S$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
<td>else end</td>
</tr>
</tbody>
</table>

$\sim \rightarrow$  

$\text{M}_{\text{IF}}$
Applying Rule M\_IF to progress\_2 and progress\_3

\[
\text{progress}_23
\]

\[
\begin{align*}
\text{when} & \quad j \neq n \\
\text{then} & \quad g(j + 1) \leq x \\
\text{if} & \quad k = j \\
\text{then} & \quad k, j := k + 1, j + 1 \\
\text{else} & \quad k, j, g := k + 1, j + 1, \text{swap}(g, k + 1, j + 1) \\
\end{align*}
\]
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)

<table>
<thead>
<tr>
<th>when $P$ then $S$</th>
<th>when $P\land\neg Q$ then (if $R$ then $T$ else $U$) end end</th>
</tr>
</thead>
<tbody>
<tr>
<td>when $P$ then $Q$ then (if $R$ then $T$ else $U$) end end</td>
<td>when $P$ then (if $Q$ then $S$ elsif $R$ then $T$ else $U$) end end</td>
</tr>
</tbody>
</table>

$\sim$ M\_ELSIF
Applying **M_ELSIF** to progress_1 and progress_23

```
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
```

final
when
  j = n
then
  skip
end
Merging Rule M\_WHILE (special case)

| when $Q$ then $S$ end | when $\neg Q$ then skip end | while $Q$ do $S$ end | M\_WHILE |
Applying \texttt{M\_WHILE4} to partition and progress 123

\begin{center}
\begin{verbatim}
progress_123_final
while \(j \neq n\) do
    if \(g(j + 1) > x\) then
        \(j := j + 1\)
    elseif \(k = j\) then
        \(k, j := k + 1, j + 1\)
    else
        \(k, j, g := k + 1, j + 1,\ \text{swap} (g, k + 1, j + 1)\)
    end
end
\end{verbatim}
\end{center}

init
\(g := f\)
\(j := 0\)
\(k := 0\)
Applying Rule M_INIT to init and progress_123_final yields

```
partition_program

\[
g, k, j := f, 0, 0 \quad ;
\]

**init**

while \( j \neq m \) 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
```
- 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 2 3 4 5 7 8 9
constants: $n, f$

variables: $g$

**axm0\_1:** $n \in \mathbb{N}$

**axm0\_2:** $0 < n$

**axm0\_3:** $f \in 1..n \mapsto \mathbb{N}$

**inv0\_1:** $g \in \mathbb{N} \leftrightarrow \mathbb{N}$
Sorting Initial Events

\[ \text{init} \quad g : \in \mathbb{N} \leftrightarrow \mathbb{N} \]

final

when

\[ g \in 1 .. n \rightarrow \mathbb{N} \]
\[ \text{ran} (g) = \text{ran} (f) \]
\[ \forall i, j \cdot \left( \begin{array}{c}
  i \in 1 .. n - 1 \\
  j \in i + 1 .. n \\
  g(i) < g(j)
\end{array} \right) \]
then skip
end

progress

status anticipated
then
\[ g : \in \mathbb{N} \leftrightarrow \mathbb{N} \]
end
- Introducing a new variable $k$ ranging from 1 to $n$

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

| 1    | sorted and $\leq$ | $k - 1$ | $k$ | ? | $n$ |
variables: $g, k, l$

inv1.1: $g \in 1..n \Rightarrow \mathbb{N}$

inv1.2: $\text{ran}(g) = \text{ran}(f)$

inv1.3: $k \in 1..n$

inv1.4: $\forall i, j \cdot \left( i \in 1..k-1 \Rightarrow \left( j \in i+1..n \Rightarrow g(i) < g(j) \right) \right)$

inv1.5: $l \in \mathbb{N}$

- We introduce an anticipated variable $l$
Sorting

3 7 2 5 8 9 4 1
Sorting
Sorting

1 2 7 5 8 9 4 3
Sorting

1 2 3 5 8 9 4 7
Sorting

1 2 3 4 8 9 5 7
Sorting

1 2 3 4 5 9 8 7
Sorting

1 2 3 4 5 7 8 9
Sorting

1 2 3 4 5 7 8 9
Sorting

| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
First Refinement: Events

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

final
\[ \text{when } k = n \text{ then skip end} \]

progress
\[ \text{any } l \text{ where} \]
\[ k < n \]
\[ l \in k .. n \]
\[ g(l) = \min(g[k .. n]) \]
\[ \text{then} \]
\[ g := g \leftarrow \{k \mapsto g(l)\} \leftarrow \{l \mapsto g(k)\} \]
\[ k := k + 1 \]
\[ l \in \mathbb{N} \]
\[ \text{end} \]

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 $\leq$ | $k - 1$ | $k$ | $? j$ | $j + 1$ | $? n$ |
variables: $g, k, j, l$

inv2_1: $j \in k \ldots n$

inv2_2: $l \in k \ldots j$

inv2_3: $g(l) = \min(g[k \ldots j])$
Sorting

3 7 2 5 8 9 4 1
Sorting

3 7 2 5 8 9 4 1
Sorting

3 7 2 5 8 9 4 1
Sorting

3 7 2 5 8 9 4 1
Sorting

\begin{center}
\begin{tabular}{cccccc}
3 & 7 & 2 & 5 & 8 & 9 & 4 & 1 \\
\end{tabular}
\end{center}
Sorting

3 7 2 5 8 9 4 1
Sorting

3 7 2 5 8 9 4 1
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

1 7 2 5 8 9 4 3
Sorting

1 7 2 5 8 9 4 3
Sorting

1 7 2 5 8 9 4 3
Sorting

1  7  2  5  8  9  4  3
Sorting

1 2 7 5 8 9 4 3
Sorting

1 2 7 5 8 9 4 3

128
Sorting

1 2 7 5 8 9 4 3
Sorting

1 2 7 5 8 9 4 3
Sorting

1 2 7 5 8 9 4 3
Sorting

1 2 7 5 8 9 4 3
Sorting

1 2 3 5 8 9 4 7
Sorting

1 2 3 5 8 9 4 7
Sorting
Sorting

1 2 3 5 8 9 4 7
Sorting

1 2 3 5 8 9 4 7
Sorting

1 2 3 4 8 9 5 7
Sorting
Sorting

1 2 3 4 8 9 5 7
Sorting

1 2 3 4 8 9 5 7
Sorting

1 2 3 4 5 9 8 7
Sorting

1 2 3 4 5 9 8 7
Sorting

1 2 3 4 5 9 8 7
Sorting
Sorting

1 2 3 4 5 7 8 9
Sorting
Sorting
Sorting 2nd Refinement: Refining Existing Events

\[
\begin{aligned}
\text{init} & \quad g, k := f, 1 \\
& \quad j, l := 1, 1 \\
\text{final} & \quad \text{when } k = n \\
& \quad \text{then} \\
& \quad \text{skip} \\
& \quad \text{end}
\end{aligned}
\]

\[
\begin{aligned}
\text{progress} & \quad \text{when } k < n, j = n \\
& \quad \text{then} \\
& \quad g := g \leftarrow \{k \mapsto g(l)\} \leftarrow \{l \mapsto g(k)\} \\
& \quad k, j, l := k + 1, k + 1, k + 1 \\
& \quad \text{end}
\end{aligned}
\]
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
  
  \[ g, k, j, l := f, 1, 1, 1; \] \hspace{1cm} \text{init}

  while \( k < n \) do
    \text{while } j < n \text{ do}
      \text{if } g(l) \leq g(j + 1) \text{ then}
        \[ j := j + 1 \]
      \text{else}
        \[ j, l := j + 1, j + 1 \]
  \endwhile

end

end

\[ k, j, l, g := k + 1, k + 1, k + 1, \text{swap } (g, k, l) \] \hspace{1cm} \text{progress}
- 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: \( g \)

\[
\begin{align*}
\text{axm0}_1: & \quad n \in \mathbb{N} \\
\text{axm0}_2: & \quad 0 < n \\
\text{axm0}_3: & \quad f \in 1 \ldots n \rightarrow \mathbb{N}
\end{align*}
\]

inv0_1: \( g \in \mathbb{N} \leftrightarrow S \)
Here is an array

\[ \begin{array}{cccccccc}
3 & 2 & 5 & 4 & 1 & 9 & 7 & 8 \\
\end{array} \]

Here is the reverse array

\[ \begin{array}{cccccccc}
8 & 7 & 9 & 1 & 4 & 5 & 2 & 3 \\
\end{array} \]

An element which was at index \( i \) is now at index \( 8 - i + 1 \).
**In Place Reversing of an Array: Events**

**init**

\[
g : \mathbb{N} \leftrightarrow S
\]

**final**

\[
\text{when} \quad g \in 1 \ldots n \rightarrow S \\
\quad \forall k \cdot \left( k \in 1 \ldots n \implies g(k) = f(n - k + 1) \right)
\]

\[
\text{then} \quad \text{skip}
\]

\[
\text{end}
\]

**progress**

\[
\text{status} \quad \text{anticipated}
\]

\[
\text{then} \quad g : \mathbb{N} \leftrightarrow S
\]

\[
\text{end}
\]
- We introduce two additional variables $i$ and $j$, both in $1 \ldots n$

- Initially $i$ is equal to 1 and $j$ is equal to $n$

- Here is the current situation:

<table>
<thead>
<tr>
<th></th>
<th>reversed</th>
<th>$i$</th>
<th>unchanged</th>
<th>$j$</th>
<th>reversed</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- A new event is going to exchange elements in $i$ and $j$. 
variables: \( g, i, j \)

\begin{align*}
\text{inv1}_1: & \quad g \in 1 .. n \to S \\
\text{inv1}_2: & \quad i \in 1 .. n \\
\text{inv1}_3: & \quad j \in 1 .. n \\
\text{inv1}_4: & \quad i + j = n + 1 \\
\text{inv1}_5: & \quad i \leq j + 1
\end{align*}
Refinement: the Main Invariants

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: \[ \forall k \cdot k \in j + 1 .. n \Rightarrow g(k) = f(n - k + 1) \]
Refinement: the Events

init
\[ i := 1 \]
\[ j := n \]
\[ g := f \]

final
\[ \text{when} \]
\[ j \leq i \]
\[ \text{then} \]
\[ \text{skip} \]
\[ \text{end} \]

\text{variant1: } j - i

progress
\[ \text{status} \]
\[ \text{convergent} \]
\[ \text{when} \]
\[ i < j \]
\[ \text{then} \]
\[ g := g \Leftrightarrow \{i \mapsto g(j)\} \Leftrightarrow \{j \mapsto g(i)\} \]
\[ i, j := i + 1, j - 1 \]
\[ \text{end} \]
reverse_program

\[ i, j, g \leftarrow 1, n, f; \]

\textbf{while} \( i < j \) \textbf{do}

\[ i, j, g \leftarrow i + 1, j - 1, \text{swap}(g, i, j) \]

\textbf{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.
- Here is a linear chain:

\[ f \rightarrow \ldots \rightarrow l \]

- 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:** $d, f, l, c$

**axioms:**

- **axm0.1:** $d \subseteq S$
- **axm0.2:** $f \in d$
- **axm0.3:** $l \in d$
- **axm0.4:** $f \neq l$
- **axm0.5:** $c \in d \setminus \{l\} \rightarrow d \setminus \{f\}$
- **axm0.6:** $\forall T \cdot T \subseteq c[T] \Rightarrow T = \emptyset$
- Given the following initial chain

\[
\begin{array}{c}
\text{f} \\
\rightarrow \\
x \\
\rightarrow \\
\ldots \\
\rightarrow \\
z \\
\rightarrow \\
l
\end{array}
\]

- Then the transformed chain should look like this:

\[
\begin{array}{c}
\text{f} \\
\leftarrow \\
x \\
\leftarrow \\
\ldots \\
\leftarrow \\
z \\
\leftarrow \\
l
\end{array}
\]
Initial Model: the Events

constants: \(d, f, l, c\)

\[\text{init} \quad r \in S \leftrightarrow S\]

\[\text{reverse} \quad r := c^{-1}\]

inv0_1: \(r \in S \leftrightarrow S\)
We introduce two additional chains \( a \) and \( b \) and a pointer \( p \)

- Node \( p \) starts both chains

- Main invariant:  \( a \cup b^{-1} = c^{-1} \)
Progressing

\[ a \]

\[ f \leftarrow \ldots \leftarrow p \rightarrow \ldots \rightarrow l \]

\[ b \]

\[ a \]

\[ f \leftarrow \ldots \leftarrow l \leftarrow p \rightarrow \ldots \rightarrow l \]

\[ b \]
variables:  \( r, a, b, p \)

"cl" is the irreflexive transitive closure operator

**inv1.1:**  \( p \in d \)

**inv1.2:**  \( a \in (\text{cl}(c^{-1})[p] \cup p) \setminus \{f\} \mapsto \text{cl}(c^{-1})[p] \)

**inv1.3:**  \( b \in (\text{cl}(c)[p] \cup p) \setminus \{l\} \mapsto \text{cl}(c)[p] \)

**inv1.4:**  \( c = a^{-1} \cup b \)
First Refinement: the Events

init
\[ r : \in S \leftrightarrow S \]
\[ a, b, p := \emptyset, c, f \]

progress
\[ \text{when} \quad p \in \text{dom}(b) \]
\[ \text{then} \]
\[ p := b(p) \]
\[ a(b(p)) := p \]
\[ b := \{p\} \sqsubset b \]
\[ \text{end} \]

reverse
\[ \text{when} \quad b = \emptyset \]
\[ \text{then} \]
\[ r := a \]
\[ \text{end} \]
- We introduce a new constant \( \textit{nil} \)
- We replace the chain \( b \) by the chain \( bn \)
- And we introduce a new pointer \( q \)

### Constants:
- \( f, l, c, \textit{nil} \)

### Variables:
- \( r, a, bn, p, q \)

#### Axioms:

**axm2.1:** \( \textit{nil} \in S \)

**axm2.2:** \( \textit{nil} \notin d \)

#### Invariants:

**inv2.1:** \( bn = b \cup \{ l \mapsto \textit{nil} \} \)

**inv2.2:** \( q = bn(p) \)
Second Refinement: the Events

progress

\[
\begin{align*}
\textbf{when} & \quad q \neq \text{nil} \\
\textbf{then} & \\
& \quad p := q \\
& \quad a(q) := p \\
& \quad q := bn(q) \\
& \quad bn := \{p\} \leftarrow bn
\end{align*}
\]

end

reverse

\[
\begin{align*}
\textbf{when} & \quad q = \text{nil} \\
\textbf{then} & \\
& \quad r := a \\
\textbf{end}
\end{align*}
\]

init

\[
\begin{align*}
& \quad r \in S \leftarrow S \\
& \quad a, bn := \emptyset, c \cup \{l \mapsto \text{nil}\} \\
& \quad p, q := f, c(f)
\end{align*}
\]
- The previous situation with two chains \( a \) and \( bn \)

\[
\begin{array}{ccccccccc}
 f & \leftarrow & \ldots & \leftarrow & p & \rightarrow & q & \rightarrow & \ldots & \rightarrow & l & \rightarrow & \text{nil} \\
\hline
 a
\end{array}
\]

- The new situation with a single chain \( d \)

\[
\begin{array}{ccccccccc}
 f & \leftarrow & \ldots & \leftarrow & p & \rightarrow & q & \rightarrow & \ldots & \rightarrow & l & \rightarrow & \text{nil} \\
\hline
 d
\end{array}
\]
Third Refinement: the State

variables: \( r, p, q, d \)

inv3_1: \( d \in S \rightarrow S \)

inv3_2: \( d = (\{f\} \leftarrow bn) \leftarrow a \)
Third Refinement: the Events

progress

when
q ≠ nil
then
p := q
d(q) := p
q := d(q)
end

reverse

when
q = nil
then
r := d ⊳ {nil}
end

init

r :∈ S ←→ S
d := {f} ⊆ (c ∪ {l ← nil})
p, q := f, c(f)
reverse_program

\[ p, q, d := f, c(f), \{ f \} \triangleq (c \cup \{ l \mapsto \text{nil} \}); \]

\[ \textbf{while } q \neq \text{nil} \textbf{ do} \]

\[ p := q \]

\[ d(q) := p \]

\[ q := d(q) \]

\[ \textbf{end;} \]

\[ r := d \bowrightarrow \{ \text{nil} \} \]
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

\[ 4^2 \leq 17 < 5^2 \]

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

\[ 4^2 \leq 16 < 5^2 \]

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

\[ 3^2 \leq 15 < 4^2 \]
Integer Square Root: Initial State and Events

constants: $n$

variables: $r$

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

final
  when $r^2 \leq n$
  then
  skip
  end

when $n < (r + 1)^2$

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

axm0.1: $n \in \mathbb{N}$

inv0.1: $r \in \mathbb{N}$
inv1_1: \[ r^2 \leq n \]

variant1: \[ n - r^2 \]

init
\[
\begin{align*}
    r &:= 0 \\
\end{align*}
\]

square_root
\[
\begin{align*}
    &\text{when} \\
    &n < (r + 1)^2 \\
    &\text{then} \\
    &\text{skip} \\
    &\text{end}
\end{align*}
\]

progress
\[
\begin{align*}
    &\text{status} \\
    &\text{convergent} \\
    &\text{when} \\
    & (r + 1)^2 \leq n \\
    &\text{then} \\
    &r := r + 1 \\
    &\text{end}
\end{align*}
\]
We obtain the following program:

```plaintext
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: \( r, a, b \)

\[
\begin{align*}
\text{init} & \\
\quad r & := 0 \\
\quad a & := 1 \\
\quad b & := 3 \\
\end{align*}
\]

\[
\begin{align*}
\text{final} & \\
\quad \text{when} & \quad n < a \\
\quad \text{then} & \\
\quad \text{skip} & \\
\text{end} & \\
\end{align*}
\]

\[
\begin{align*}
\text{when} & \quad a \leq n \\
\text{then} & \\
\quad r & := r + 1 \\
\quad a & := a + b \\
\quad b & := b + 2 \\
\text{end} & \\
\end{align*}
\]

\[
\begin{align*}
\text{inv2.1:} & \quad a = (r + 1)^2 \\
\text{inv2.2:} & \quad b = 2r + 3 \\
\end{align*}
\]
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
```
- 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: | $\forall i, j \cdot \left\{ \begin{array}{l} i \in \mathbb{N} \\
 j \in \mathbb{N} \\
 i < j \\
 \Rightarrow \\
 f(i) < f(j) \end{array} \right\}$ |

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

| thm0_1: | $f \in \mathbb{N} \mapsto \mathbb{N}$ |
Inverse of an Injective Numerical Function: the Events

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

final
when $f(r) \leq n < f(r + 1)$
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 $q$ such that:

$$\ f(r) \leq n < f(q + 1) \$$
First Refinement: the State (1)

constants: \( f, n, a, b \)

variables: \( r, q \)

\[
\begin{align*}
\text{axm1}_1: & \quad a \in \mathbb{N} \\
\text{axm1}_2: & \quad b \in \mathbb{N} \\
\text{axm1}_3: & \quad f(a) \leq n \\
\text{axm1}_4: & \quad n < f(b + 1)
\end{align*}
\]
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)

\[
\text{init} \\
\quad r, q := a, b
\]

\[
\text{final} \\
\quad \text{when} \\
\quad \quad r = q \\
\quad \text{then} \\
\quad \quad \text{skip} \\
\quad \text{end}
\]
dec
refines progress status convergent
any \( x \) where
\[ r \neq q \]
\[ x \in r + 1 \ldots q \]
\[ n < f(x) \]
then
\[ q := x - 1 \]
end

inc
refines progress status convergent
any \( x \) where
\[ r \neq q \]
\[ x \in r + 1 \ldots q \]
\[ f(x) \leq n \]
then
\[ r := x \]
end

variant1: \( q - r \)
- We reduce the non-determinacy

<table>
<thead>
<tr>
<th>dec</th>
<th>inc</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>when</strong></td>
<td><strong>when</strong></td>
</tr>
<tr>
<td>( r \neq q )</td>
<td>( r \neq q )</td>
</tr>
<tr>
<td>( n &lt; f((r + 1 + q)/2) )</td>
<td>( f((r + 1 + q)/2) \leq n )</td>
</tr>
<tr>
<td>then</td>
<td>then</td>
</tr>
<tr>
<td>( q := (r + 1 + q)/2 - 1 )</td>
<td>( r := (r + 1 + q)/2 )</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>
inverse\_program

\[ r, q := a, b; \]
\begin{align*}
\text{while } r \neq q \text{ do} & \\
\text{if } n < f((r + 1 + q)/2) \text{ then} & \\
q & := (r + 1 + q)/2 - 1 \\
\text{else} & \\
r & := (r + 1 + q)/2
\end{align*}
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**: $f(a) \leq n < f(b + 1)$
- $f$ 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

\[ r, q := 0, n; \]
\[ \textbf{while } r \neq q \textbf{ do} \]
\[ \textbf{if } n < \left( \frac{r + 1 + q}{2} \right)^2 \textbf{ then} \]
\[ q := \frac{r + 1 + q}{2} - 1 \]
\[ \textbf{else} \]
\[ r := \frac{r + 1 + q}{2} \]
\[ \textbf{end} \]
\[ \textbf{end} \]
- $f$ 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: $n/m$
integer_division_program

\[ r, q := 0, n; \]
\[ \text{while } p \neq q \text{ do} \]
\[ \quad \text{if } n < m \times (r + 1 + q)/2 \text{ then} \]
\[ \quad \quad q := (r + 1 + q)/2 - 1 \]
\[ \quad \text{else} \]
\[ \quad \quad r := (r + 1 + q)/2 \]
\[ \quad \text{end} \]
\[ \text{end} \]