XV. Development of Sequential Programs (July 2008)

In this Chapter, we shall see how to develop *sequential programs*. We present the approach we shall use, and then we propose a large number of examples.

Sequential programs (e.g. loops), when formally constructed, are usually developed gradually by means of a series of progressively more refined "sketches" starting with the formal specification and ending in the final program. Each such sketch is already (although often in a highly non-deterministic form) a monolithic description which resumes the final intended program in terms of a single formula. This is precisely that initial "formula", that is gradually transformed into the final program.

We are not going to use this approach here. After all, in order to prove a large formula, a logician usually breaks it down into various pieces, on which he performs some simple manipulations before putting them again together in a final proof.

1 A Systematic Approach to Sequential Program Development

1.1 Components of a Sequential Program

A sequential program is essentially made up of a number of *individual assignments* that are glued together by means of various constructs. Typical constructs are sequential composition (;), loop (**while**), and condition (**if**). Their role is to explicitly *schedule* these assignments in a proper order so that the execution of the program can achieve its intended goal. Here is an example of a sequential program where the various assignments have been emphasized:

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

Note that, to simplify matters, we use a pidgin imperative language allowing us to have multiple assignments as in:

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

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

Although it is not important for what we want to explain here, note that the expression $\mathsf{swap}\,(g, k+1, j+1)$ stands for the swapping of the values $g(k+1)$ and $g(j+1)$ in the array g. Also note that we have used a

syntax with opening (**while**, **if**), intermediate (**do**, **then**, **elsif**, **else**) and closing (**end**) keywords . We could have used another syntax which would have been more appealing to Java or C programmers. In fact, the syntax used here in not important as long as we understand what we are writing.

In summary, we shall develop programs written in a simple pidgin programming language with the following syntax for program statements:

$< variable > := < expressions >$

$< statement > ; < statement >$

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

if $< condition >$ **then** $< statement >$ **elsif** $\ldots$ **else** $< statement >$ **end**

while $< condition >$ **do** $< statement >$ **end**

Moreover, expressions will denote natural numbers, arrays, and also pointers.

1.2 Decomposing a Sequential Program into Individual Events

The approach we present here is to completely separate, during the design, these individual assignments from their scheduling. This approach is thus essentially one where we favor an initial implicit *distribution of the computation* over a centralized explicit one. At a certain stage, the "program" will just be made of a number of naked events, performing some actions under the control of certain guarding conditions. And at this point the synchronization of these events is not our concern. Thinking operationally, it is done *implicitly* by a hidden scheduler, which *may fire* an event once its guard holds. We can express as follows the various naked events corresponding to the previous example:

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
$k := k + 1$
$j := j + 1$
end

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

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

This decomposition has been done in a very systematic fashion. As can be seen, the guard of each event has been obtained by collecting the conditions that are introduced by the **while** and **if** statements. For instance, the second event dealing with assignments $k := k + 1$ and $j := j + 1$ has got the following guards:

- $j \neq m$, because this assignment is inside the loop starting with **while** $j \neq m$ **do** $\ldots$ **end**.
- $g(j+1) \leq x$, because we are *not* in the first branch of the **if** $x < g(j+1)$ **then** $\ldots$ **end** statement.
- $k = j$, because we are in the first branch of the **elsif** $k = j$ **then** $\ldots$ **end** statement.

Conversely, it seems easy to build the initial program of previous section from these four naked events. But, of course, this process will have to be made systematic later in section 3.

1.3 Sketch of the Approach

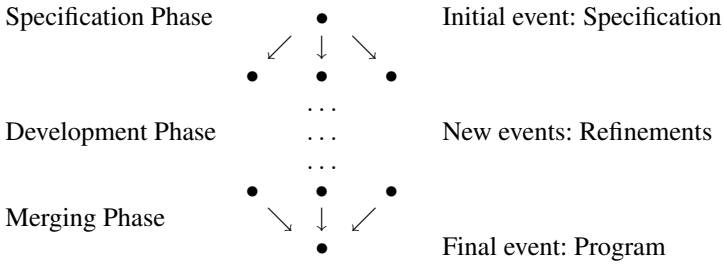
After what has been said in the previous section, the approach we are going to take can be divided up into three distinct phases:

1. At the beginning of the development process, the event system, besides an initialising event, is made of a single guarded event with no action. This event represents the specification of our future program. At this step, we might also define an anticipated event (see section 2).

2. During the development process, other events might be added or some abstract anticipated events might become convergent (see section 2).

3. When all the individual pieces are "on the table" (this is the situation shown in the previous section), and only then, we start to be interested in their *explicit* scheduling. This will have the effect of minimizing guard evaluations in the scheduling. For this, we apply certain systematic rules (section 3) whose role is to gradually *merge the events* and thus organize them into a single entity forming our final program. The application of these rules has the effect of gradually eliminating the various predicates making the guards. At the end of the development, it results in a single guardless final "event".

What is interesting about this approach is that it gives us full freedom to refine small pieces of the future program, and also to create new ones, *without being disturbed by others* : the program is developed by means of small *independent* parts that so remain until they are eventually put together systematically at the end of the process. This can be illustrated in the following diagram:



1.4 Sequential Program Specification: Pre- and Post-condition

A sequential program P with some input parameters and some results is often specified by using a so called Hoare-triple, of the following shape:

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

Here Pre denotes the *pre-condition* of the program P, while *Post* denotes its *post-condition*.The pre-condition define the condition we can assume concerning the parameters of the program, and the post-condition denotes what we can expect concerning the outcome of the program.

It is very simple to have an Hoare-triple being encoded within an event system. The parameters are constants and the pre-conditions are the axioms of these constants. The results are variables and the program is represented by an event containing the post-condition in its guard together with a skip action. We illustrate this in the next section with a very simple example.

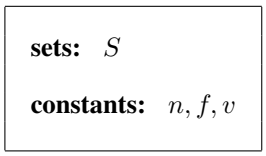
2 A Very Simple Example

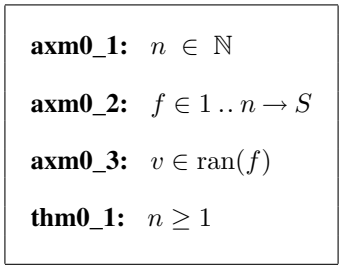
2.1 Specification

Suppose we want to specify a program, named search, with the following parameters: an array f of size n, and a certain value v guaranteed to be within the range of the array f. The result of our program search is denoted by r, which is an index of the array f such that $f(r) = v$. This informal specification can be made a little more formal by the following Hoare-triple:

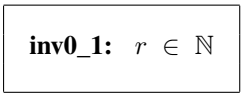
$$\left\{ \begin{array}{l} n \in \mathbb{N} \\ f \in 1 \mathinner{..} n \rightarrow S \\ v \in \mathrm{ran}(f) \end{array} \right\} \quad \mathsf{search} \quad \left\{ \begin{array}{l} r \in 1 \mathinner{..} n \\ f(r) = v \end{array} \right\}$$

The previous example can then be encoded in a straightforward fashion as follows. Here is the encoding of the pre-condition:

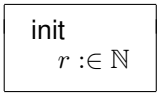


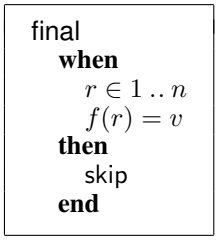


As can be seen, the three predicates making the pre-conditions have become three axioms **axm0_1** to **axm0_3**. Notice the theorem **thm0_1** we have stated. Now comes the post-condition encoding. First the definition of the result variable r:



As can be seen, the invariant is very weak, we just say in invariant **inv0_1** that the result r is a natural number. Next are the two events named init and final:





The two predicates making the post-condition have become guards of the final event, which has no action. Finally, we introduce the following anticipated event progress:

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

This event modifies variable r in a totally non-deterministic way. This is a technique we introduced in section 7 of chapter 4 and reused again in section 4.2 of chapter 6. We are going to use this technique systematically throughout this chapter.

2.2 Refinement

The development of sequential programs will exactly follow the same lines as those we have already followed in previous chapters: namely doing some refinements by looking more carefully at the state and introducing new events or (in the present case) refining an anticipated event.

In this searching example, our refinement will be extremely simple. We do not introduce any new variable: we rather add invariants on r and refine event progress. Variable r ranges over the interval $1\,..\,n$ (invariant **inv1_1**). The main invariant states that v is not within the set denoting the image of the interval $1\,..\,r-1$ under f, that is $f[1\,..\,r-1]$ (invariant **inv1_2**). In other words, interval $1\,..\,r-1$ denotes the set of indices we have already explored *unsuccessfully*.

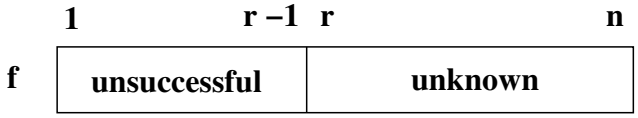
variables: r

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

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

variant1: $n - r$

This can be illustrated in the following figure:



Now we make our previous anticipated event progress, convergent. It increments r when $f(r)$ is not equal to v. Also notice the initialisation of the result variable r.

init
$r := 1$

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

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

Notice that event progress is now convergent. For this, we provide variant **variant1** which is a natural number decreased by event progress. All proofs are left to the reader.

2.3 Generalisation

Note that the previous example can be generalized to the case where the searched value v is not necessarily within the range of the array f. The program may have then two different outcomes: (1) the searched value has not been found, and (2) the searched value has been found and the result is then a corresponding index. This will be represented in the abstraction by two distinct events and an additional boolean variable $success$, which is equal to true when v is in the range of f, false otherwise. It is also possible to re-arrange the previous solution so that both solutions are very close to each other.

3 Merging Rules

At this point, our development is almost finished. It just remains for us to merge the events in order to obtain our final program. For this we shall define some *merging rules*. We essentially have two merging rules, one for defining a conditional statement (M_IF) and the other one for defining a loop statement (M-WHILE). Here are these rules:

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

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

These rules can be read as follows: if we have an event system where two events have forms corresponding to the ones shown on the left of the $\rightsquigarrow$ symbol in the rule, they can be merged into a single *pseudo-event* corresponding to the right hand side of the rule. Notice that both rules have the same antecedent events, so that the application of one or the other might be problematic. There is no confusion, however, as the rules have some *incompatible side conditions*, which are the following:

- The second rule (that introducing **while**) requires that the first antecedent event (that giving rise to the "body" S of the loop) appears as *new or non-anticipated, thus convergent* at *one refinement level below that of the second one*. In this way, we are certain that there exists a variant ensuring that the loop terminates. Moreover, *the first event must keep the common condition P invariant*. The merged event is considered to "appear" at the same level as the second antecedent event.
- The first rule (that introducing **if**) is applicable when both events have been introduced at the same level. The merged event is considered to bear the same "level" as the component events.

The first rule may take a special form when one of the antecedent events has an **if** form. It goes as follows:

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

Note that in the three rules, the common guard P is optional. When P is missing then the pseudo-event on the right hand side of the rule reduces to a non-guarded event. Also note that in the merging rule M_WHILE, the action T can be reduced to skip. In these case the rule simplifies accordingly.

The rules are applied systematically until a single pseudo-event with no guard is left. It then remains for us to apply a last merging rule, called M_INIT, consisting in prepending the unique pseudo-event with the initialization event. In our case, it leads to the final program construction:

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

4 Example: Binary Search in a Sorted Array

4.1 Initial Model

This problem is exactly the same as the previous one: searching a value in an array. So, the formal specification is (almost) identical to the one of the previous example. The only difference is that, this time, we have more information on the array: it is an array of natural numbers which is sorted in a non-decreasing way as indicated in **axm0_4**. Here is the pre-condition:

constants: n f v

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

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

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

thm0_1: $n \geq 1$

axm0_4: $\forall i, j \cdot\ i \in 1 \,..\, n \Rightarrow j \in 1 \,..\, n \Rightarrow i \leq j \Rightarrow f(i) \leq f(j)$

Here is now 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

4.2 First Refinement

The State. We introduce two new variables p and q. Variables p and q are supposed to be two indices in the array f (**inv1_1** and **inv1_2**). Variable r is within the interval $p \mathinner{..} q$ (**inv1_3**). Moreover, the value v is supposed to be a member of the set denoting the image of the interval $p \mathinner{..} q$ under f, that is $f[p \mathinner{..} q]$ (**inv1_4**). Here is the state of this refinement:

variables: r, p, q

inv1_1: $p \in 1 \mathinner{..} n$

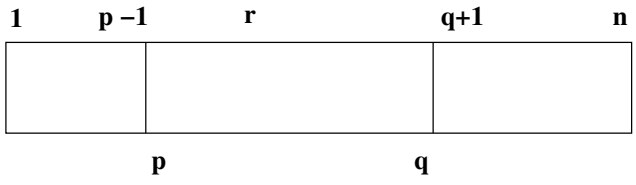
inv1_2: $q \in 1 \mathinner{..} n$

inv1_3: $r \in p \mathinner{..} q$

inv1_4: $v \in f[p \mathinner{..} q]$

variant1: $q - p$

The current situation is illustrated in the following figure:



Now, we introduce two events called inc and dec which split abstract anticipated event progress. This events are convergent (see **variant1** above). They increment or decrement p or q when $f(r)$ is smaller or greater than v. They also move r non-deterministically within the new interval $p \mathinner{..} q$.

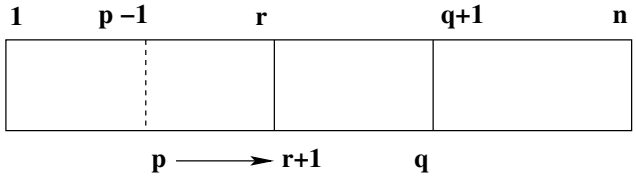
init
$p := 1$
$q := 1$
$r :\in 1 \,..\, n$

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

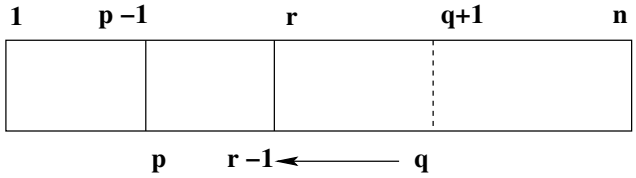
dec
refines
progress
status
convergent
when
$v < f(r)$
then
$q := r - 1$
$r :\in p \,..\, r - 1$
end

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

The following figure illustrates the situation encountered by event inc (the arrow indicates the new value of index p), which is guarded by the predicate $f(r) < v$:



The following figure illustrates the situation encountered by event dec (the arrow indicates the new value of index q), which is guarded by the predicate $v < f(r)$:



The proofs are simple. We encourage the reader to do them with the Rodin Platform

4.3 Second Refinement

The second refinement is very simple. The state is the same as in the abstraction. We only reduce the non-determinism of events inc and dec by choosing r in the "middle" of the intervals $r + 1 \,..\, q$ or $p \,..\, r - 1$. This leads to the following refinements of these events.

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

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

The main proofs concern the implication of the common actions on r in events inc and dec. For event dec this amounts to proving $(r + 1 + q)/2 \in r + 1 \,..\, q$, which is obvious since we have already proved in the abstraction that the interval $r + 1 \,..\, q$ was not empty (feasibility of abstract event dec). The proof of event inc is similar.

4.4 Merging

We are now ready to merge events inc and dec. For this we can use merging rule M_IF thus obtaining (on the left) the following pseudo-event inc_dec. After that, we can merge this pseudo-event with event final. For this, we can use merging rule M_WHILE, thus obtaining the pseudo-event situated on the right:

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

The final program is obtained by pre-pending the initialization (rule M-INIT):

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

5 Example: Minimum of an Array of Natural Numbers

5.1 Initial Model

Our next elementary example consists in looking for the minimum of the range of a non-empty array of natural numbers. Let n and f be two constants, and m a variable. Here is our initial model:

constants: n f

axm0_1: $0 < n$
axm0_2: $f \in 1 \mathinner{..} n \rightarrow \mathbb{N}$
thm0_1: $\mathrm{ran}(f) \neq \varnothing$

variables: m

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

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

minimum
$m := \min(\mathrm{ran}(f))$

5.2 First Refinement

Our first refinement consists in introducing, as in the previous example, two indices p and q where p is not greater than q as indicated in invariant **inv1_3**. Moreover, it is shown in invariant **inv1_4** that the minimum of the array is in the set $f[p \,..\, q]$:

constants: n, f

variables: m, p, q

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

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

inv1_3: $p \leq q$

inv1_4: $\min(\mathrm{ran}(f)) \in f[p \,..\, q]$

We also introduce two new events inc and dec. When p is smaller that q and $f(p)$ is greater than $f(q)$, we can reduce the interval $p \,..\, q$ to $p + 1 \,..\, q$ since $f(p)$ is certainly not the minimum we are looking for. We have a similar effect with invariant dec. The minimum is then found when p is equal to q according to invariant **inv1_4**.

init
$p, q := 1, n$
$m :\in \mathbb{N}$

inc
when
$p < q$
$f(p) > f(q)$
then
$p := p + 1$
end

dec
when
$p < q$
$f(p) \leq f(q)$
then
$q := q - 1$
end

minimum
when
$p = q$
then
$m := f(p)$
end

We leave it as an exercise for the reader to prove this refinement (don't forget to prove the convergence of events inc and dec) and generate the corresponding final program by applying some merging rules.

6 Example: Array Partitioning

In this example all proofs are left to the reader.

6.1 Initial Model

The problem we study now is a variant of the well known partitioning problem used in Quicksort. Let f be an array of n natural numbers (supposed to be distinct for simplification). Let x be a natural number. We would like to transform f in another array g with exactly the same elements as the initial array f, in such a way that there exists an index k of the interval $0 \,..\, n$ such that all elements in $g[1 \,..\, k]$ are smaller than or equal to x while all elements in $g[k + 1 \,..\, n]$ are strictly greater than x. The final result is that shown in the following diagram:

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

For example, let the array f be the following:

3	7	2	5	8	9	4	1

If we like to partition it with 5, then the transformed array g can be the following with k being set to 5:

3	2	5	4	1	9	7	8

k

Note that in case all elements of f are greater than x, then k should be equal to 0. And in case all elements are smaller than or equal to x, then k should be equal to n. We now have enough elements to introduce our initial model as follows:

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

6.2 First Refinement

Our next step is to introduce one new variable j. Variables j and k are indices in $0 \mathinner{..} n$. Variable k is supposed to be smaller than or equal to j. We have also two invariants saying that k and j partition the array g as indicated by the following diagram:

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

As can be seen, the array g is partitioned in the interval $1 \,..\, j$ with k being the intermediate partitioning point. The idea is then to possibly increment j alone or both k and j while maintaining the corresponding invariant. The process is completed when j is equal to n. More formally, this yields the following new state:

variables: k, g, j

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

inv1_2: $k \in 0 \,..\, j$

inv1_3: $g \in 1 \,..\, n \rightarrowtail \mathbb{N}$

inv1_4: $\mathrm{ran}(g) = \mathrm{ran}(f)$

inv1_5: $\forall m \cdot m \in 1 \,..\, k \Rightarrow g(m) \leq x$

inv1_6: $\forall m \cdot m \in k+1 \,..\, j \Rightarrow x < g(m)$

Here are the refinements of the events init and final, and the introduction of three convergent events progress_1, progress_2, and progress_3 all refining abstract anticipated event progress (guess the variant):

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

final
when
$j = n$
then
skip
end

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

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

progress_3
refines
progress
status
convergent
when
$j \neq n$
$g(j+1) \leq x$
$k \neq j$
then
$k := k+1$
$j := j+1$
$g := g \lhd\!\!+ \{k+1 \mapsto g(j+1)\} \lhd\!\!+ \{j+1 \mapsto g(k+1)\}$
end

6.3 Merging

By merging these events, we obtain the following final program:

partition_program
$j, k, g := 0, 0, f;$
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
$k, j, g := k + 1, j + 1, g \Leftarrow \{k + 1 \mapsto g(j + 1)\} \Leftarrow \{j + 1 \mapsto g(l + 1)\}$
end
end

7 Example: Simple Sorting

In this example, all proofs are left to the reader.

7.1 Initial Model

In this section, we are not going to develop a very smart sorting algorithm. Our intention is only to use sorting as an opportunity to develop a little program containing an *embedded loop*. We have two constants: n, which is a positive natural number, and f, which is a total injective function from $1 \,..\, n$ to the natural numbers. We have a result variable g which must be sorted and have the same elements as f. Here is our initial state:

constants: n f

axm0_1: $0 < n$

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

variables: g

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

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

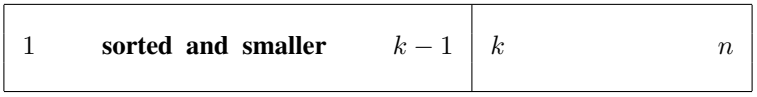
final
when
$g \in 1 \,..\, n \rightarrowtail \mathbb{N}$
$\mathrm{ran}\,(g) = \mathrm{ran}\,(f)$
$\forall i, j \cdot \; i \in 1 \,..\, n-1$
$j \in i+1 \,..\, n$
$\Rightarrow$
$g(i) < g(j)$
then
skip
end

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

The guards in event final stipulates that g has exactly the same elements as the original f, and that it is sorted in an ascending way.

7.2 First Refinement

In our first refinement, we introduce a new index k supposed to be in the interval $1 \,..\, n$. Moreover, the elements of the sub-part of g ranging from 1 to $k-1$ are all sorted and also smaller than the elements lying in the other sub-part, namely those ranging from k to n. This can be illustrated in the following diagram:



We also introduce a new variable l and a new anticipated event prog. In the guard of the convergent event progress (guess the variant), we require that $g(l)$ is the minimum of the set $g[k \,..\, n]$. Our new state and events are as follows. Notice that events init, progress, and prog all modify l non-deterministically:

variables: g, k, l

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

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

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

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

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

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

final
when
$k = n$
then
skip
end

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

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

7.3 Second Refinement

Our next step consists in determining the minimum chosen arbitrarily in the previous section. For this, we introduce an additional index j. The index j ranges from k to n, whereas l ranges from k to j. The value of g at index l is supposed to be the minimum of g on the sub-part of it ranging from k to j. Here is our new state:

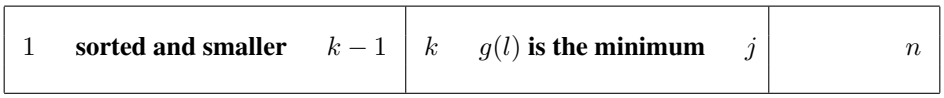
variables: g, k, l, j

inv2_1: $j \in k \mathinner{..} n$

inv2_2: $l \in k \mathinner{..} j$

inv2_3: $g(l) = \min(g[k \mathinner{..} j])$

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



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
$g := g \lhd\!\!\!\!- \{k \mapsto g(l)\} \lhd\!\!\!\!- \{l \mapsto g(k)\}$
$k := k + 1$
$j := k + 1$
$l := k + 1$
end

In the concrete event progress, the strengthening of the guard (with condition $j = n$) implies that the value of the variable l corresponds exactly to the minimum chosen arbitrarily in the abstraction. Here are the new convergent events prog1 and prog2 (guess the variant) both refining abstract anticipated event prog:

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

7.4 Merging

After applying the merging rule we obtain the following final program:

```
sort_program
  g, k, j, l := f, 1, 1, 1
  while k ≠ n do
    while j ≠ n do
      if g(l) ≤ 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, g ⩤ {k ↦ g(l)} ⩤ {l ↦ g(k)}
  end
```

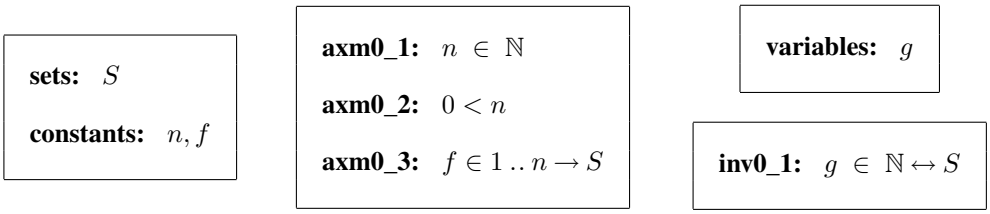
Note that the initialization of the inner loop variables, namely j and l, is made in two different places: either in the proper initialization at the beginning of the program, or in the trailing statement after the inner loop itself.

8 Example: Array Reversing

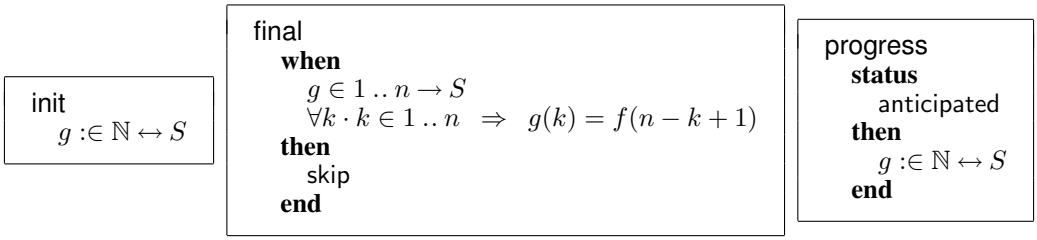
In this example, all proofs are left to the reader.

8.1 Initial Model

Our next example is the classical array reversing. We are given a carrier set S, and two constants n and f, and a variable g. Here is the state:



Here are the events:



8.2 First Refinement

Our first refinement consists in introducing two indices i, starting at 1, and j, starting at n. The indices i and j move towards each other. The array g is gradually reversed by swapping elements $g(i)$ and $g(j)$ while, of course, i is strictly smaller than j. This is done in the event progress. In this way, the sub-arrays of g ranging from 1 to $i-1$ and from $j+1$ to n respectively have all their elements reversed with regard to the original array f. And the middle part is still unchanged with regards to f. This can be illustrated in the following diagram:

Notice that the quantity $i+j$ is always equal to $n+1$. At the end of the process, either i is equal to j when n is odd, or i is equal to $j+1$ when n is even. But, in both cases, we have $i \geq j$. Here is the new state:

variables: g, i, j

inv1_1: $g \in 1 \mathinner{..} n \rightarrow S$

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

Here are the refined events (guess the variant for event progress):

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

Now, we can apply the merging rules and obtain the following final program:

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

9 Example: Reversing a Linked List

So far, all our examples were dealing with arrays and corresponding indices. As a consequence some of the proofs relied on elementary arithmetic properties. In this example, we experiment with a data structure that deals with pointers. The problem we shall tackle is very classical and simple: we just want to reverse a linear chain. Notice that to simplify matters the chain is made of pointers only. In other words, a node of the chain has no information field.

9.1 Initial Model

Each node in the chain points to its immediate successor (if any). The chain starts with a node called f (for "first") and ends with a node called l (for "last"). All this can be represented in the following figure:

$$\boxed{f} \rightarrow \boxed{x} \rightarrow \ldots \rightarrow \boxed{z} \rightarrow \boxed{l}$$

Before engaging in our problem, we first have to formalize what we have just introduced. After renaming its constants, we simply copy the axioms which have been presented in section 7.4 of chapter 9:

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

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

We would like to reverse this chain. So, if the initial chain is:

$$\boxed{f} \rightarrow \boxed{x} \rightarrow \ldots \rightarrow \boxed{z} \rightarrow \boxed{l}$$

then the transformed chain r should look like this:

$$\boxed{f} \leftarrow \boxed{x} \leftarrow \ldots \leftarrow \boxed{z} \leftarrow \boxed{l}$$

Here is the definition of the result r together with the event reverse doing the job in one shot: r is exactly the converse of c.

variables: r

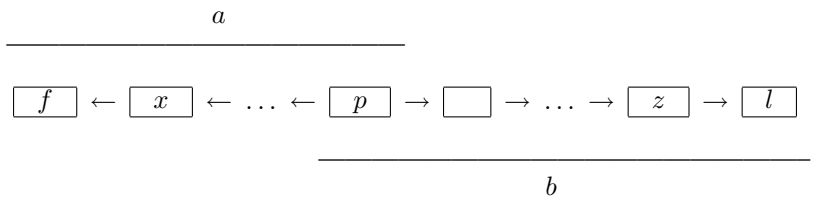
inv0_1: $r \in S \leftrightarrow S$

init
$r :\in S \leftrightarrow S$

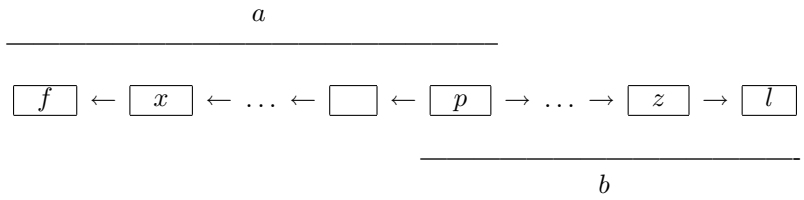
reverse
$r := c^{-1}$

9.2 First Refinement

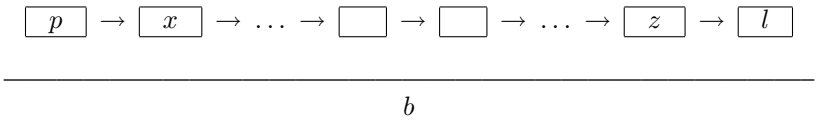
In this first refinement, we introduce two additional chains a and b and a pointer p. Chain a corresponds to the part of chain c that has already been reversed, whereas chain b corresponds to the part of chain c that has not yet been reversed. Node p is the starting node of both chains. Here is the situation:



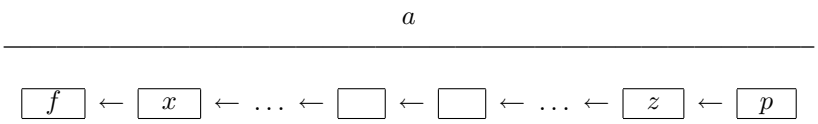
Progressing is obtained by moving p one step to the right and reversing the first pointer of chain b. This is indicated in the following figure:



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:



Formalizing what we have just informally presented is simple: we define both chains a and b and their relationship with c. Notice that we use $\text{cl}(c)$ and $\text{cl}(c^{-1})$, which are the irreflexive transitive closures of c and c^{-1} (cl is defined in section 7.1 of chapter 9).

variables: r, a, b, p

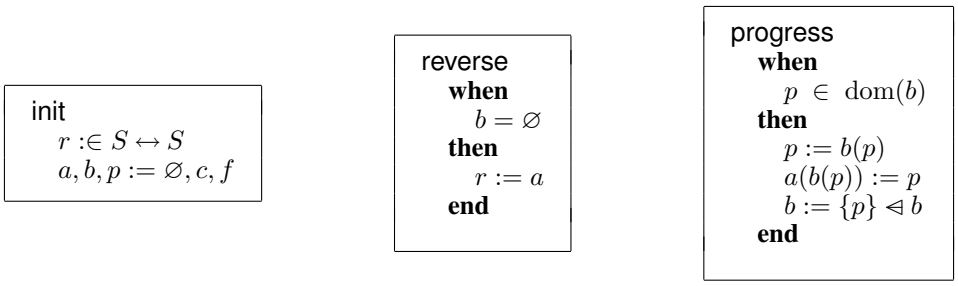
inv1_1: $p \in d$

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

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

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

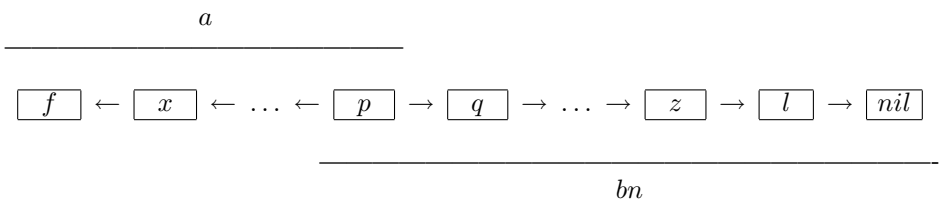
Here are the refinements of the previous events and also the introduction of the new event progress:



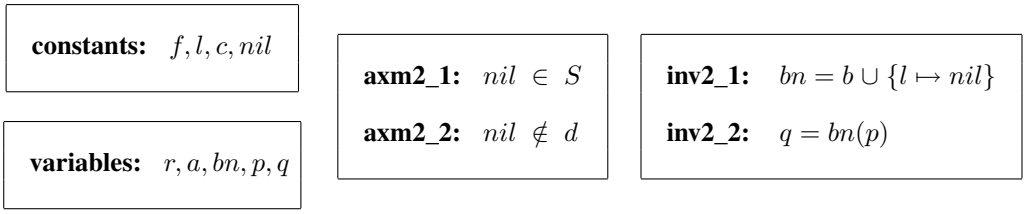
As can be seen in event progress, p is moved to the right (that is $p := b(p)$), the pair $b(p) \mapsto p$ is added to the chain a (that is $a(b(p)) := p$), and finally node p is removed from chain b (that is $b := \{p\} \lhd b$).

9.3 Second Refinement

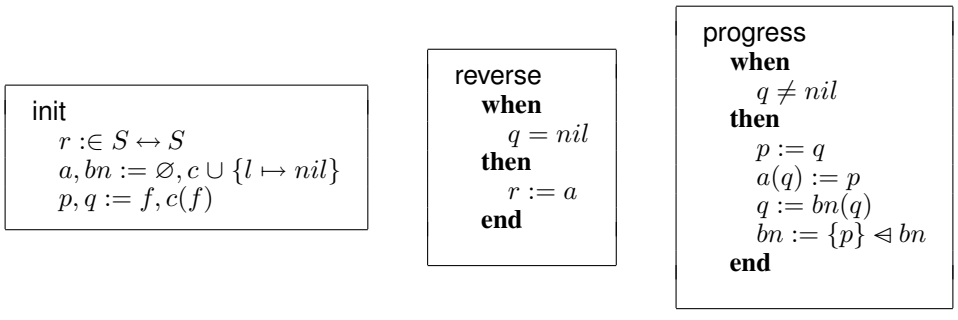
In this refinement, we introduce a special constant node named nil (**axm2_1**), which is supposed to be outside the set d. We also replace the chain b by the chain bn which is equal to $b \cup \{l \mapsto nil\}$ (**inv2_1**). Finally, we introduce a second pointer, q, which is equal to $bn(p)$. This is represented as follows:



Here is the new state:

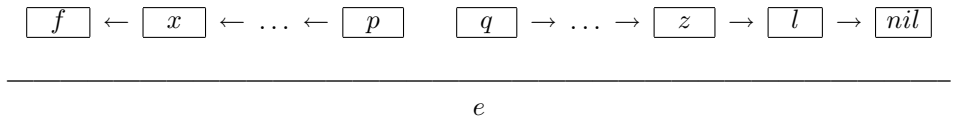


Here are the refinements of the events. Notice that the guards have been made independent of the chain b:



9.4 Third Refinement

We now remove chains a and bn and replace it by a unique chain, e, containing both chains a and bn. Here is the new situation:



Next is the refined state with the definition of the new variable e in terms of the abstract variables a and bn:

variables: r, p, q, e

inv3_1: $e = (\{f\} \lhd bn) \lhd\!\!+ a$

The events correspond to straightforward transformations of the previous one:

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

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

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

9.5 Merging

The last refinement leads to the following final program:

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

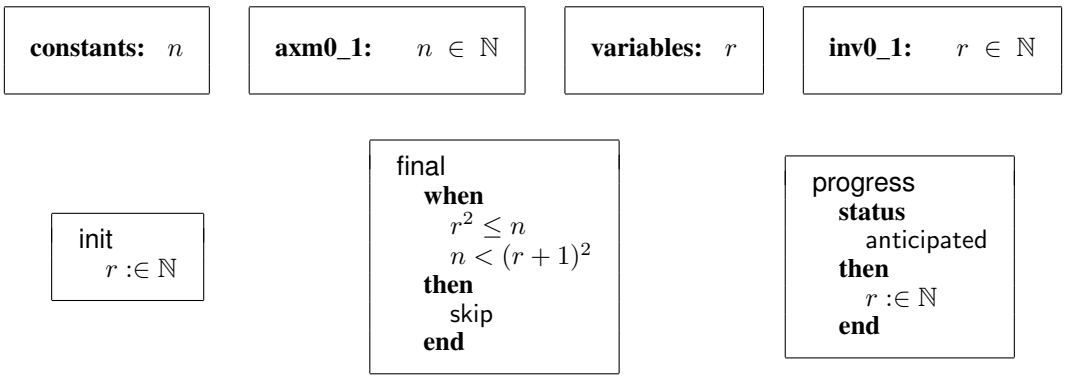
10 Example: Simple Numerical Program Computing the Square Root

We have not yet tried our approach on a numerical example. This is the purpose of this section. Given a natural number n, we want to compute its natural number square root by defect, that is a number r such that the following holds:

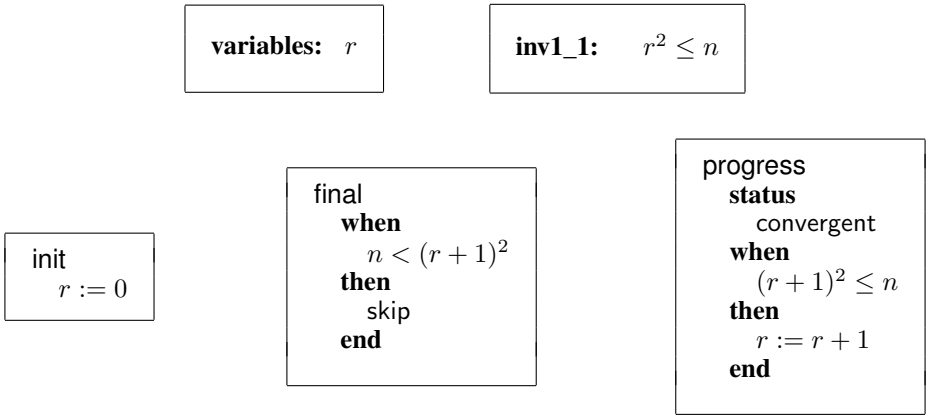
$$r^2 \leq n < (r+1)^2$$

10.1 Initial Model

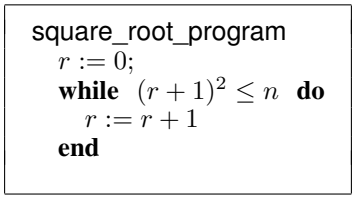
Our first model is simply the following:



10.2 First Refinement



The proof of this refinement is straightforward but don't forget to prove the convergence of event progress. We obtain the following program:



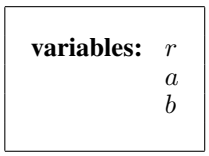
10.3 Second Refinement

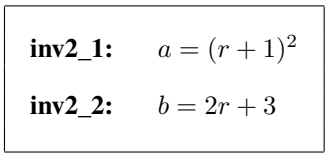
The previous solution, although correct, is a bit heavy because we have to compute $(r+1)^2$ at each step of the computation. We would like to investigate whether it would be possible to refine the previous solution by computing this quantity in a less expensive way. The idea relies on the following equalities:

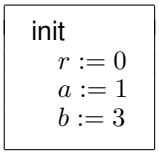
$$((r+1)+1)^2 \;=\; (r+1)^2 + (2r+3)$$

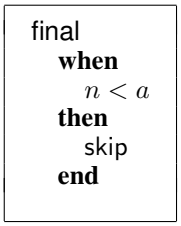
$$2(r+1)+3 \;=\; (2r+3)+2$$

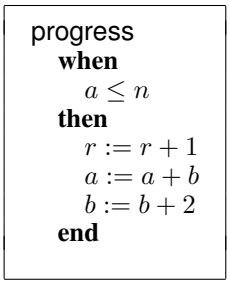
We are thus extending our state with two more variables a and b recording in advance respectively $(r+1)^2$ and $2r+3$. Here is the new state and the new program:



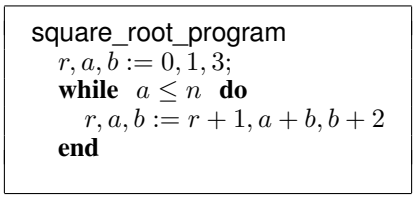








We obtain the following program:

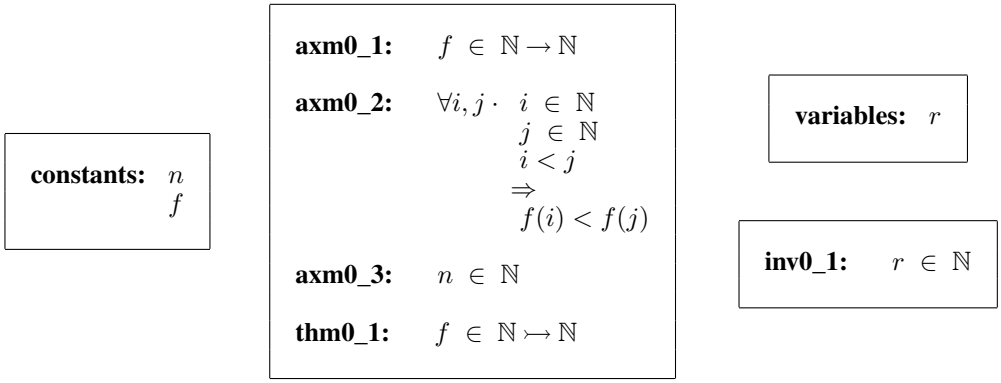


11 Example: The Inverse of an Injective Numerical Function

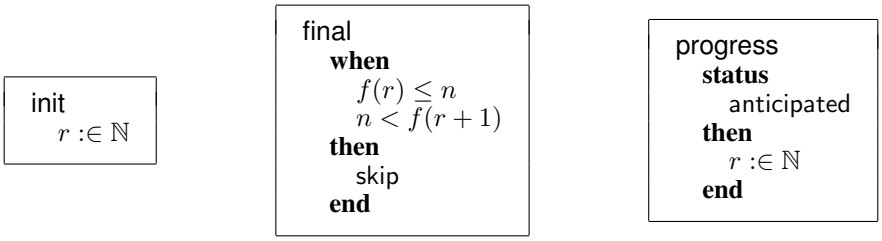
In this example we are trying to borrow some ideas coming from the binary search example of section 4. We want to compute the inverse by defect of an injective numerical function defined on all natural numbers.

11.1 Initial Model

This is a generalization of the previous example where the function in question was the squaring function. In this model we are a bit more specific than was announced: our function f is not stated to be injective to begin with. It is only defined to be a total function in **axm0_1**. But it is said in **axm0_2** that this numerical function f is strictly increasing. As a consequence, it can be proved to be injective, namely that its inverse is also a function: this is stated in **thm0_1**.



Event final calculates the inverse by defect of f at n in one shot is just a generalization of what was defined in the previous example.



11.2 First Refinement

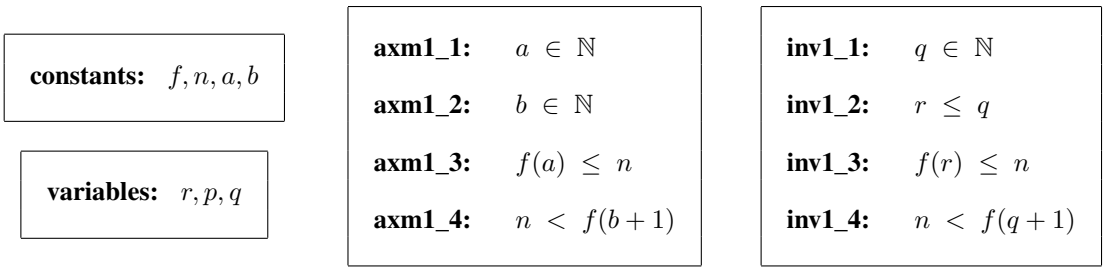
The idea of this refinement is to suppose first that we can exhibit two numerical constants a and b, which are such that the following conditions hold:

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

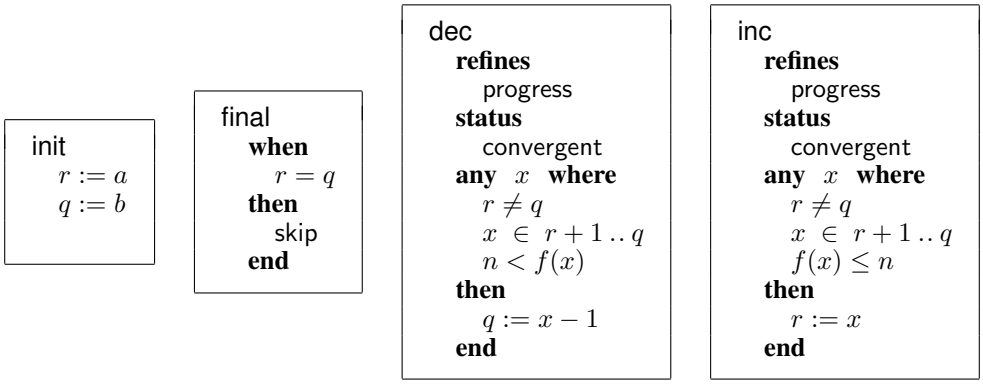
We are certain then that our result r is within the interval $a..b$ since f is defined everywhere and increasing. The idea of the refinement is then to narrow this initial interval. For this, we introduce a new variable q initially set to b, whereas variable r is initially set to a. These two variables will have the following invariant property:

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

When r and q are equal, we are done. When r and q are distinct we are left to perform a search in the interval $r..q$. For this we shall use a technique very close to the one we used in section 4 for binary search. Here is the state of this refinement:



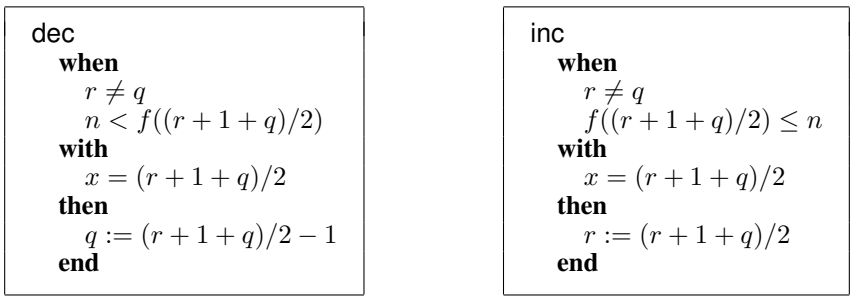
We introduce two new events inc and dec. As can be seen, a number x is chosen non-deterministically in the interval $p+1\,..\,q$. Then n is compared with $f(x)$ and an action is done accordingly on q or on p.



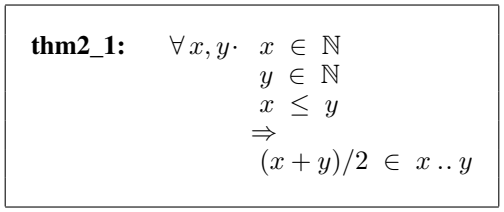
The proof of this refinement is not difficult. One has to exhibit a variant and prove that it is decreased by the new events. This variant is not difficult to guess.

11.3 Second Refinement

In this second refinement, we are going to remove the non-determinacy in the events inc and dec. This will be done by choosing for the local variable x the "middle" of the interval $r+1\,..\,q$. In this refinement, we do not change the state, only the events dec and inc as follows:



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



As a result, we obtain the following program, by using some of the merging rules:

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

11.4 Instantiation

The development we have done in this example is interesting because it is *generic*. By this, it is meant that is can be *instantiated*. For this, it is sufficient to provide some values to the constants and to provide proofs that the proposed values are obeying the properties that were given for these constants.

In our case, the constants to be instantiated are f, a, and b. The constant n will remain as it is since it corresponds to the quantity for which we want to compute the inverse function value. And the *properties we have to prove* for the proposed instantiations are **axm0_1** (f is a total function defined on $\mathbb{N}$), **axm0_2** (f is an increasing function), **axm1_3** ($f(a) \leq n$), and **axm1_4** ($n < f(b+1)$).

11.5 First Instantiation

If we take for f the squaring function, then the computation will provide the square root of n by defect. More precisely, we shall compute a quantity r such that the following holds:

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

We have thus to prove that the squaring function is total and increasing, which is trivial. Now, given a value n we have to find two numbers a and b such that the following holds:

$$a^2 \ \leq \ n \ < \ (b+1)^2$$

It is easy to see that a can be instantiated to 0 and b to n. As a result, we have *for free* the following program calculating the square root of n:

```
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
```

11.6 Second Instantiation

If we take for f the function "multiply by m", *where m is a positive natural number*, then the computation will provide the integer division of n by m. More precisely, we shall compute a quantity r such that the following holds:

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

We have thus to prove that this function is total and increasing, which is trivial. Now, given a value n we have to find two numbers a and b such that the following holds:

$$m \times a \ \leq \ n \ < \ m \times (b+1)$$

It is easy to see that a can be instantiated to 0 and b to n (remember, m is a *positive* natural number). As a result, we have *for free* the following program calculating the integer division of n by m.

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