Event Driven Sequential Program Construction

by J.-R. Abrial

Version 12 (March 2000)

Event Driven Sequential Program Construction

J.-R. Abrial

Consultant

Introduction

Sequential programs (i.e. loops), when formally constructed, are usually developped gradually by means of a series of more and more refined "sketches" starting with the formal specification and ending up in the final program. Each such sketch is already (although often in a highly non-deterministic form) a unique piece concentrating the final intended program on a single formula. This is precisely that initial formula that is gradually transformed into the final program.

It is argued here that this might *not be the right approach*. After all, in order to prove a large formula, a logician usually breaks it down into various pieces on which he performs some little work before putting them again together in a final proof. We would like to experiment with such a paradigm by looking whether it is applicable to construct programs as well.

A sequential program is essentially made up of a number of individual assignments that are glued together by means of various constructs: conditional statements, sequential compositions, loops. The rôle of such constructs is to explicitly *schedule* these assignments in a proper order so that the execution of the program can achieve its intended goal.

The idea we want to explore here is to completely separate, during the design, these individual assignments from their scheduling. This approach is thus essentially one by which we favor an initial implicit *distribution of computation* over a centralized explicit one. At a certain stage, the "program" is just made of a number of "naked" guarded commands (which we call here "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.

In the course of the development process, such events might be (data) refined (guards might be strengthened) and other events might be added (which have to refine skip). As we shall see below, this has to be done in a certain disciplined manner. 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 developped by means of little *independant* parts that so remain until they are eventually put together systematically at the end of the process.

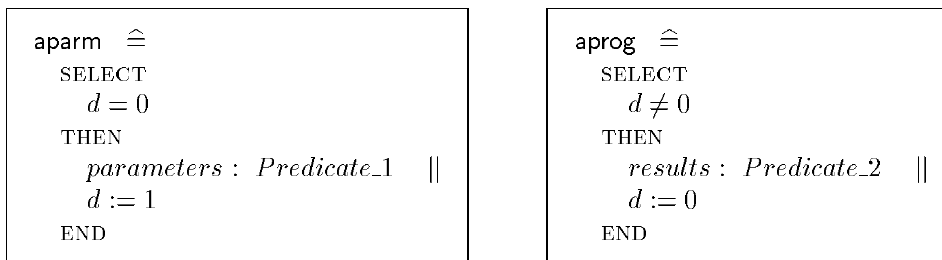
When all the individual pieces are "on the table", and only then, we start to be interested in their *explicit* scheduling. For this, we apply certain systematic rules whose rôle is to gradually organize these pieces into a single entity forming our final program. The application of these rules has the effect of making the explicit guards pointless. As a matter of fact, at the end of the development, they will have completely disappeared.

The paper is essentially made up of nine illustrating examples. Before engaging in the examples however, we formally define the properties of our event systems and we then propose a number of transformation rules that we are going to apply systematically in the sequel. In the examples, we shall only make brief mentions of the formal proofs since they have been done mechanically with Atelier B.

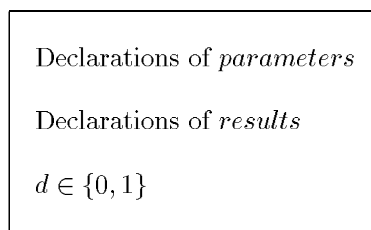
Properties of the Event System[1]

In this section, we shall express the general properties that an event system used for program development should satisfy. We shall also fix the *style* we shall adopt in our future program development.

(1) *At the beginning of a development*, our event system only contains two events that can be fired alternatively: the first one sets the parameters of the future program while the second one sets the corresponding results. The two events together form an implicit loop that "runs" for ever. Here is the general shape of these events:



Here *parameters* and *results* denote some variables of the event system, $Predicate_1$ denotes the initial condition, which the parameters of the program should satisfy, and $Predicate_2$ denotes the final condition, which the results of the program should satisfy[2]. These predicates together constitute the *specification* of the future program. The variables (and also possibly some constants) are together typed in an invariant containing at least the following



(2) *During the development*, the event system may contain more than two guarded events. These events are together constrained by the following laws:

[1] An event system is a potentially infinite loop on a finite choice of different events.

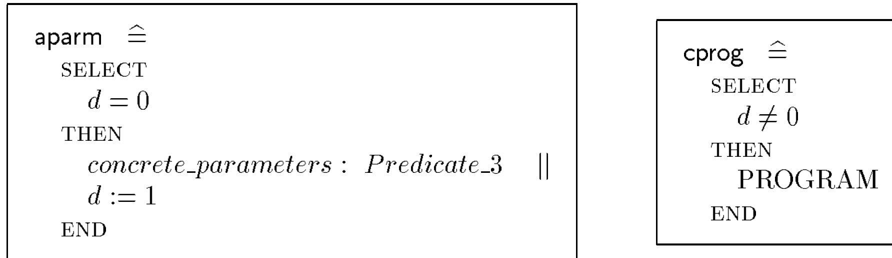
[2] We remind the reader that the notation $x : P(x, x_0)$ corresponds to the non-deterministic assignment of variable x to any value satisfying the predicate $P(x, x_0)$, where the optional x_0 denotes, by convention, the initial value of x

– LAW 1: The disjunction of the guards of the events must always be true (under the current invariant). This is so because we want that our system remains "live": in no way should the system be able to deadlock.

– LAW 2: No event can ever "take control" for ever: this means that the new events that are introduced at some point in the development must always decrease some natural number quantity (or some lexicographically ordered sequence).

Notice that these two laws trivially apply to the initial event system.

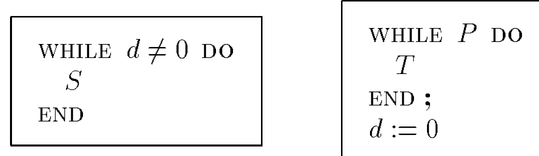
(3) *At the end of the development*, one should obtain again two events of the following shape



```
aparm ≙
  SELECT
    d = 0
  THEN
    concrete_parameters : Predicate_3   ||
    d := 1
  END
```

```
cprog ≙
  SELECT
    d ≠ 0
  THEN
    PROGRAM
  END
```

where PROGRAM can take one of the following two shapes



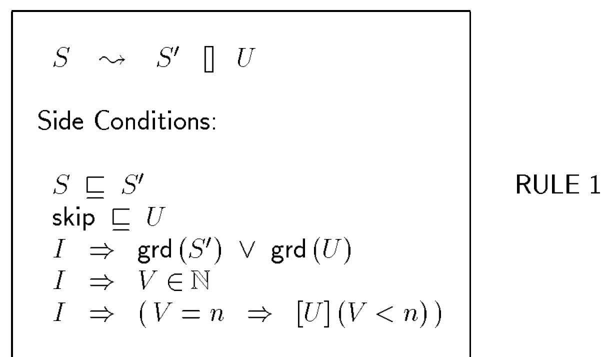
```
WHILE d ≠ 0 DO
  S
END
```

```
WHILE P DO
  T
END ;
d := 0
```

where S is a piece of code assigning d to 0, and where P is a predicate and T a piece of code that both do not involve d. In any case, it can be seen that d is equal to 0 after the "execution" of the concrete event cprog as is the case with the abstract event aprog.

Some Event System Transformation (Refinement) Rules

Let S be a choice of events (guarded actions). Let $\Longrightarrow$ be the guarding operator[3]. Let $[]$ be the choice operator. We consider the following transformation rules:



$S \leadsto S' \; [] \; U$

Side Conditions:

$S \sqsubseteq S'$
$\text{skip} \sqsubseteq U$
$I \Rightarrow \text{grd}(S') \lor \text{grd}(U)$
$I \Rightarrow V \in \mathbb{N}$
$I \Rightarrow (V = n \Rightarrow [U](V < n))$

RULE 1

[3] The construct $P \Longrightarrow S$ is just a shorthand for the more verbose SELECT P THEN S END

where $\sqsubseteq$ denotes the data refinement operator. As can be seen, the new event U refines skip and decreases some natural number quantity (under the current invariant I). Note that the last three side conditions essentially verify that the above LAW 1 and LAW 2 are satisfied. The next law put together two events to form an IF statement in an obvious manner:

$$
\begin{array}{l}
(P \wedge Q \implies S) \; [\!] \; (P \wedge \neg Q \implies T) \; [\!] \; U \\
\leadsto \\
(P \implies \text{IF } Q \text{ THEN } S \text{ ELSE } T \text{ END}) \; [\!] \; U
\end{array}
\qquad \text{RULE 2}
$$

Our third law transforms a single event into a WHILE statement. The idea is to force the firing of that event as much as possible by excluding that of other enabled events.

$$
\begin{array}{l}
(P \implies S) \; [\!] \; T \\
\leadsto \\
(P \implies \text{WHILE } P \text{ DO } S \text{ END}) \; [\!] \; T
\end{array}
\qquad \text{RULE 3}
$$

Side Condition:

S is not itself guarded

RULE 1 is essentially used in the first half of the development (putting the events "on the table"), while RULE 2 and RULE 3 are used in the second half (putting the events together in order to built a sequential program).

Clearly, other rules could be invented (another one will be added below however), but for simplicity, and because we shall not use other ones in this paper, we have limited ourselves to these rules.

Example 1: Searching in an Array

Our intention is to build a simple search program able to find the index of a sequence whose corresponding value is given (this value being known in advance to belong to the sequence). Let s be a certain set, and let n and f be two constants such that the following property holds:

$$
\begin{array}{l}
n \in \mathbb{N}_1 \\
f \in 1 \,..\, n \rightarrow s
\end{array}
$$

Let i, x, and d be three variables, such that the following invariant holds:

$$
\begin{array}{l}
i \in \mathrm{dom}\,(f) \\
x \in \mathrm{ran}\,(f) \\
d \in \{0, 1\}
\end{array}
$$

Let, finally, aparm and aprog be the following events, which, alternatively, sets x to any member of the range of f[4], and sets i to any "index" of the domain of f, in such a way that $f(i)$ is exactly x.

aparm $\widehat{=}$ SELECT $d = 0$ THEN $x :\in \mathsf{ran}\,(f) \quad ||$ $d := 1$ END	aprog $\widehat{=}$ SELECT $d \neq 0$ THEN $i : (\, i \in \mathsf{dom}\,(f) \;\wedge\; f(i) = x\,) \quad ||$ $d := 0$ END

Our goal is to eventually "implement" the second event by means of a search loop. We want to do this by decomposing aprog into several events, then refining them, and finally re-composing them into a single event again (all this being done by applying systematically the above rules). We now introduce the variable k together with the following (gluing) invariant[5]:

$$k \in 0\,..\,n-1$$

$$x \notin f\,[1\,..\,k]$$

This invariant is quite intuitive: it simply expresses that the first k elements of the sequence f have already been explored unsuccessfully. The success will thus be certainly obtained eventually in the remaining part of the sequence since we know that x belongs to the range of f. We apply now RULE 1 together with the following new event progress:

aparm $\widehat{=}$ SELECT $d = 0$ THEN $x :\in \mathsf{ran}\,(f) \quad ||$ $k, d := 0, 1$ END	aprog $\widehat{=}$ SELECT $d \neq 0 \quad \wedge$ $f(k+1) = x$ THEN $i, d := k+1, 0$ END	progress $\widehat{=}$ SELECT $d \neq 0 \quad \wedge$ $f(k+1) \neq x$ THEN $k := k+1$ END

The side conditions are not difficult to prove: they concern the correct data refinements of events aparm and aprog relative to their respective abstractions, the correct data refinement of event progress relative to skip, and the decreasing by event progress of a certain natural number quantity, namely $n - k$. By applying now RULE 2, we group together events aprog and progress to form event body. Event aparm is left unchanged.

[4] The construct $x :\in s$ used below is a shorthand for $x : (x \in s)$

[5] We remind the reader that the construct $r[s]$ where r is a binary relation, and s a subset of the source of r, denotes the image of s under r

```
aparm  ≙
  SELECT
    d = 0
  THEN
    x :∈ ran (f)   ||
    k, d := 0, 1
  END
```

```
body  ≙
  SELECT
    d ≠ 0
  THEN
    IF f(k + 1) = x THEN
      i := k + 1   ||
      d := 0
    ELSE
      k := k + 1
    END
  END
```

By applying now RULE 3, we transform event body into event cprog. Event aparm is left unchanged.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    x :∈ ran (f)   ||
    k, d := 0, 1
  END
```

```
cprog  ≙
  SELECT
    d ≠ 0
  THEN
    WHILE d ≠ 0 DO
      IF f(k + 1) = x THEN
        i, d := k + 1, 0
      ELSE
        k := k + 1
      END
    END
  END
```

One More Rule

In the event cprog above, there is obviously something that is inefficient, namely the guarding of the loop by the condition $d \neq 0$. We have the feeling that it could be replaced by the condition $f(k+1) \neq x$ since the body of the loop sets d to 0 exactly when the condition $f(k+1) = x$ holds. The purpose of the next rule is to overcome this difficulty:

RULE 4

$$(P \wedge Q \implies S) \;[\,]\; (P \wedge \neg Q \implies T) \;[\,]\; U$$

$\rightsquigarrow$

$$(P \implies \text{WHILE } Q \text{ DO } S \text{ END} \,;\, T) \;[\,]\; U$$

Side Condition:

S and T are not guarded
$I \wedge P \wedge Q \;\Rightarrow\; [S]\,P$

Notice that the loop body S maintains P invariant (under the guard Q).

Example 1 Revisited

At some point in **Example 1** we had the following events:

```
aparm  ≙
  SELECT
    d = 0
  THEN
    x :∈ ran (f)   ||
    k, d := 0, 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0   ∧
    f(k + 1) = x
  THEN
    i, d := k + 1, 0
  END
```

```
progress  ≙
  SELECT
    d ≠ 0   ∧
    f(k + 1) ≠ x
  THEN
    k := k + 1
  END
```

We can use RULE 4 on events progress and aprog to form event new_cprog that looks more "natural":

```
new_cprog  ≙
  SELECT
    d ≠ 0
  THEN
    WHILE f(k + 1) ≠ x DO k := k + 1 END ;
    i, d := k + 1, 0
  END
```

Example 2: Looking for the Maximum of an Array of Numbers

Our next elementary example consists in looking for the maximum of a non empty sequence of natural numbers. Let n be a constant and f, m, and d be three variables such that the following holds:

$$n \in \mathbb{N}_1$$

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

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

$$d \in \{0, 1\}$$

Let, finally, aparm and aprog be the following events, which, alternatively, set f to any natural number sequence of size n, and set m to the maximum of the range of f.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    f :∈ 1 .. n → ℕ   ||
    d := 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0
  THEN
    m, d := max (ran (f)), 0
  END
```

Our goal is to "implement" the second event by means of an exhaustive analysis of the sequence f. For this, we introduce the variables k and m' (replacing m), together with the following (gluing) invariant.

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

$$m' = \max(f[1\,..\,k])$$

$$d = 0 \;\Rightarrow\; m' = m$$

This invariant expresses that m' is indeed the maximum of the first k elements of the sequence f. It also says that the concrete variable m' and abstract variable m are the same when the condition $d = 0$ holds (that is, when the analysis is completed). We now apply RULE 1 thus defining the new events test_1 and test_2. This leads to the following:

```
aparm  ≙
  SELECT
    d = 0
  THEN
    f, m' : (f ∈ (1 .. n) → ℕ ∧ m' = f(1))   ||
    k, d := 1, 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0   ∧
    k = n
  THEN
    d := 0
  END
```

```
test_1  ≙
  SELECT
    d ≠ 0   ∧
    k ≠ n   ∧
    f(k + 1) ≤ m'
  THEN
    k := k + 1
  END
```

```
test_2  ≙
  SELECT
    d ≠ 0   ∧
    k ≠ n   ∧
    f(k + 1) > m'
  THEN
    k, m' := k + 1, f(k + 1)
  END
```

The side conditions are not difficult to prove: clearly both new events decrease the natural number quantity $n-k$. Notice that the event aprog does not calculate any more; as a matter of fact, it just "observes" that the calculation has been completed. Our next step is to put together events test_1 and test_2 by applying RULE 2, thus forming event test_12 while events aparm and aprog are left unchanged.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    f, m' : (f ∈ (1 .. n) → ℕ ∧ m' = f(1))   ||
    k, d := 1, 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0   ∧
    k = n
  THEN
    d := 0
  END
```

```
test_12  ≙
  SELECT
    d ≠ 0    ∧
    k ≠ n
  THEN
    IF  f(k+1) ≤ m′  THEN
      k := k+1
    ELSE
      k, m′ := k+1, f(k+1)
    END
  END
```

By applying now RULE 4, we group together events aprog and test_12 thus forming event cprog. Events aparm is left unchanged.

```
cprog  ≙
  SELECT
    d ≠ 0
  THEN
    WHILE  k ≠ n  DO
      IF  f(k+1) ≤ m′  THEN
        k := k+1
      ELSE
        k, m′ := k+1, f(k+1)
      END
    END ;
    d := 0
  END
```

Example 3: Searching in a Matrix

This example is very close to **Example 1**. Rather than searching in a sequence, our intention is now to search in a matrix. We would like to see whether this will significantly modify the structure of our final program. Given a set S, let m, n and f be three constants such that the following property holds:

$$m \in \mathbb{N}_1$$

$$n \in \mathbb{N}_1$$

$$f \in (1\,..\,m) \times (1\,..\,n) \to S$$

Let i, j, x, and d be four variables such that the following invariant holds:

$$
\begin{array}{l}
(i,j) \in \mathsf{dom}\,(f) \\
x \in \mathsf{ran}\,(f) \\
d \in \{0,1\}
\end{array}
$$

Let, finally, aparm and aprog be the following events, which, alternatively, set x to any member of the range of f, and set i and j to any "indices" of the domain of f, in such a way that $f(i,j)$ is exactly x.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    x :∈ ran (f)   ||
    d := 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0
  THEN
    (i, j) : ( (i, j) ∈ dom (f)  ∧ ∈ f(i, j) = x )   ||
    d := 0
  END
```

We now introduce the variables k and l together with the following (gluing) invariant:

$$
\begin{array}{l}
k \in 0\,..\,m-1 \\
l \in 0\,..\,n \\
x \notin f\,[(1\,..\,k) \times (1\,..\,n)] \;\;\wedge\;\; x \notin f\,[\{k+1\} \times (1\,..\,l)]
\end{array}
$$

This invariant is also quite intuitive: it simply expresses that the first k rows of the matrix f have already been explored unsuccessfully as well as the first l column of row $k+1$. We apply now RULE 1 together with the two new events progress_l and progress_k:

```
aparm  ≙
  SELECT
    d = 0
  THEN
    x :∈ ran (f)   ||
    k, l, d := 0, 0, 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0   ∧
    l ≠ n   ∧
    f(k + 1, l + 1) = x
  THEN
    i, j, d := k + 1, l + 1, 0
  END
```

```
progress_l  ≙
  SELECT
    d ≠ 0   ∧
    l ≠ n   ∧
    f(k + 1, l + 1) ≠ x
  THEN
    l := l + 1
  END
```

```
progress_k  ≙
  SELECT
    d ≠ 0   ∧
    l = n
  THEN
    k, l := k + 1, 0
  END
```

The side conditions are not difficult to prove: they concern the correct data refinements of events aparm and aprog relative to their respective abstractions, and the correct data refinements of events progress_k and progress_l relative to skip. Both these events decrease the natural number quantity $(n+1) \times (m-k) - l$. By applying RULE 2, we group together events aprog and progress_l. Events aparm and progress_k are left unchanged.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    x :∈ ran (f)   ||
    k, l, d := 0, 0, 1
  END
```

```
progress_k  ≙
  SELECT
    d ≠ 0   ∧
    l = n
  THEN
    k, l := k + 1, 0
  END
```

```
aprog_progress_l  ≙
  SELECT
    d ≠ 0   ∧
    l ≠ n
  THEN
    IF f(k + 1, l + 1) = x THEN
      i, j, d := k + 1, l + 1, 0
    ELSE
      l := l + 1
    END
  END
```

Quite naturally, we are tempted to group together (by applying RULE 2 again) events progress_k and aprog_progress_l to form event body. Events aparm is left unchanged.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    x :∈ ran (f)   ||
    k, l, d := 0, 0, 1
  END
```

```
body  ≙
  SELECT
    d ≠ 0
  THEN
    IF l = n THEN
      k, l := k + 1, 0
    ELSIF f(k + 1, l + 1) = x THEN
      i, j, d := k + 1, l + 1, 0
    ELSE
      l := l + 1
    END
  END
```

By applying now RULE 3, we transform event body into event cprog. Event aparm is left unchanged.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    x :∈ ran(f)  ||
    k, l, d := 0, 0, 1
  END
```

```
cprog  ≙
  SELECT
    d ≠ 0
  THEN
    WHILE d ≠ 0 DO
      IF l = n THEN
        k, l := k + 1, 0
      ELSIF f(k + 1, l + 1) = x THEN
        i, j, d := k + 1, l + 1, 0
      ELSE
        l := l + 1
      END
    END
  END
```

Notice that, *quite unexpectedly*, we end up with a single loop.

Example 4: Testing for a Row Filled in with 0's in a Number Matrix

As in previous example, we shall now consider a natural number matrix. Our intention is to develop a program searching for the possibility for that matrix to have a row entirely filled in with zeros. In previous examples we were certain to end up "before" the end of the exhaustive search. That is, we did not test in any way that the index k was reaching its natural end. And this was so because it was known in advance that the search was certainly successful. Here the situation is different in that we may terminate either before the natural end (in case of success, when there exists a row entirely filled in with zeros) or, alternatively, at the natural end (in case of failure, when there does not exist such a row). We would like to see whether this makes the final program look differently. We have thus two constants m and n and three variables f, d and r such that the following holds:

$$m \in \mathbb{N}_1$$

$$n \in \mathbb{N}_1$$

$$f \in (1\,..\,m) \times (1\,..\,n) \rightarrow \mathbb{N}$$

$$d \in \{0, 1\}$$

$$r \in \{0, 1\}$$

Let, finally, aparm, aprog_0 and aprog_1[6] be the following events, which alternatively modify f and calculate r. As can be seen, r is equal to 0 (success) if and only if there exists

[6] Notice that we have *two* aprogs here instead of one as advocated above. This is because there is a natural breakdown into two parts due to the fact that there are two result values discriminated according to a certain predicate (we could have used a conditional statement instead but this would have been heavier at this stage)

a row i in the matrix f that is entirely filled in with zeros: in other words, the image of this row under matrix f is exactly the singleton $\{0\}$

```
aparm  ≙
  SELECT
    d = 0
  THEN
    f :∈ (1 .. m) × (1 .. n) → ℕ   ||
    d := 1
  END
```

```
aprog_0  ≙
  SELECT
    d ≠ 0   ∧
    ∃ i · ( i ∈ 1 .. m  ∧  f [{i} × (1 .. n)] = {0} )
  THEN
    r, d := 0, 0
  END
```

```
aprog_1  ≙
  SELECT
    d ≠ 0   ∧
    ¬ ∃ i · ( i ∈ 1 .. m  ∧  f [{i} × (1 .. n)] = {0} )
  THEN
    r, d := 1, 0
  END
```

We now introduce variables k and l together with the following (gluing) invariant:

$$k \in 1 \,..\, m$$

$$l \in 0 \,..\, n$$

$$\forall i \cdot (\, i \in 1 \,..\, k-1 \;\Rightarrow\; f\,[\{i\} \times (1 \,..\, n)] \neq \{0\}\,)$$

$$l \neq 0 \;\Rightarrow\; f\,[\{k\} \times (1 \,..\, l)] = \{0\}$$

This invariant says (1) that none of the $(k-1)$ first rows of f are entirely filled in with zeros (in other words, we have a failure in all these rows: this is indeed why row k is considered), and, (2) that, on the contrary, the first l (when l is not equal to 0) elements of row k are all equal to zero (that is, success so far in this row: this is why we shall continue exploring it). We now apply RULE 1 and introduce the new events progress_l (success so far on row l) and progress_k (failure on row k, which is not the last one). Event aparm, aprog_0 (success on row k), and aprog_1 (failure on row m) are data refined.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    f :∈ (1..m) × (1..n) → ℕ  ||
    k, l, d := 1, 0, 1
  END
```

```
aprog_0  ≙
  SELECT
    d ≠ 0  ∧
    l = n
  THEN
    r, d := 0, 0
  END
```

```
aprog_1  ≙
  SELECT
    d ≠ 0  ∧
    l ≠ n  ∧
    f(k, l + 1) ≠ 0  ∧
    k = m
  THEN
    r, d := 1, 0
  END
```

```
progress_l  ≙
  SELECT
    d ≠ 0  ∧
    l ≠ n  ∧
    f(k, l + 1) = 0
  THEN
    l := l + 1
  END
```

```
progress_k  ≙
  SELECT
    d ≠ 0  ∧
    l ≠ n  ∧
    f(k, l + 1) ≠ 0  ∧
    k ≠ m
  THEN
    k, l := k + 1, 0
  END
```

It is not hard to prove that the old events are genuine refinements of their respective abstract counterparts, and that each of the new events indeed refines skip. Both these events decrease the natural number quantity $(n + 1) \times (m - k) - l$. We now apply RULE 2 three times in a straightforward way, first on events aprog_1 and progress_k, then on progress_l, and, finally, on aprog_0 to obtain event body. Event aparm is left unchanged. By applying now RULE 3, we obtain our final program cprog

```
body  ≙
  SELECT
    d ≠ 0
  THEN
    IF l = n THEN
      r, d := 0, 0
    ELSIF f(k, l + 1) = 0 THEN
      l := l + 1
    ELSIF k = m THEN
      r, d := 1, 0
    ELSE
      k, l := k + 1, 0
    END
  END
```

```
cprog  ≙
  SELECT
    d ≠ 0
  THEN
    WHILE d ≠ 0 DO
      IF l = n THEN
        r, d := 0, 0
      ELSIF f(k, l + 1) = 0 THEN
        l := l + 1
      ELSIF k = m THEN
        r, d := 1, 0
      ELSE
        k, l := k + 1, 0
      END
    END
  END
```

Example 5: Finding a Common Number of two Sets

The example we shall consider now is a bit different from the previous ones. Here, data refinement will play a more important rôle as we shall start from more abstract data structures (sets) in comparison with those encountered above (sequence, matrices). Our aim is to construct an algorithm for finding any element belonging to two finite sets of natural

numbers, whose intersection is guaranteed to be not empty. We have four variables a, b, x, and d defined as follows[7]:

$$
\begin{array}{|l|}
\hline
a \in \mathbb{F}(\mathbb{N}) \\
b \in \mathbb{F}(\mathbb{N}) \\
a \cap b \neq \emptyset \\
x \in \mathbb{N} \\
d \in \{0,1\} \\
\hline
\end{array}
$$

We now define the following abstract events aparm and aprog

$$
\begin{array}{|l|}
\hline
\textsf{aparm} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\quad\quad d = 0 \\
\quad \text{THEN} \\
\quad\quad a, b : \begin{pmatrix} a \in \mathbb{F}(\mathbb{N}) \;\wedge \\ b \in \mathbb{F}(\mathbb{N}) \;\wedge \\ a \cap b \neq \emptyset \end{pmatrix} \quad || \\
\quad\quad d := 1 \\
\quad \text{END} \\
\hline
\end{array}
\qquad
\begin{array}{|l|}
\hline
\textsf{aprog} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\quad\quad d \neq 0 \\
\quad \text{THEN} \\
\quad\quad x :\in a \cap b \quad || \\
\quad\quad d := 0 \\
\quad \text{END} \\
\hline
\end{array}
$$

The idea of the next step is to possibly remove from sets a or b some elements that are not common to both. For this, we introduce two sets a' and b' replacing a and b. As a gluing invariant we maintain the value of the intersection of a' and b' to be equal to that of the intersection of a and b.

$$
\begin{array}{|l|}
\hline
a' \in \mathbb{F}(\mathbb{N}) \\
b' \in \mathbb{F}(\mathbb{N}) \\
a' \cap b' = a \cap b \\
\hline
\end{array}
$$

We now apply RULE 1 thus introducing events rmv_a and rmv_b. Events aparm and aprog are slightly modified to deal now with a' and b' rather than a and b.

[7] The construct $\mathbb{F}(s)$ denotes the set of finite subsets of a set s

$$
\begin{array}{l}
\mathsf{aparm} \;\widehat{=} \\
\quad \text{SELECT} \\
\quad\quad d = 0 \\
\quad \text{THEN} \\
\quad\quad a', b' : \begin{pmatrix} a' \in \mathbb{F}(\mathbb{N}) \;\wedge \\ b' \in \mathbb{F}(\mathbb{N}) \;\wedge \\ a' \cap b' \neq \emptyset \end{pmatrix} \quad || \\
\quad\quad d := 1 \\
\quad \text{END}
\end{array}
\qquad
\begin{array}{l}
\mathsf{aprog} \;\widehat{=} \\
\quad \text{SELECT} \\
\quad\quad d \neq 0 \\
\quad \text{THEN} \\
\quad\quad x :\in a' \cap b' \quad || \\
\quad\quad d := 0 \\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{rmv_a} \;\widehat{=} \\
\quad \text{ANY } y \text{ WHERE} \\
\quad\quad d \neq 0 \quad \wedge \\
\quad\quad y \in a' - b' \\
\quad \text{THEN} \\
\quad\quad a' := a' - \{y\} \\
\quad \text{END}
\end{array}
\qquad
\begin{array}{l}
\mathsf{rmv_b} \;\widehat{=} \\
\quad \text{ANY } y \text{ WHERE} \\
\quad\quad d \neq 0 \quad \wedge \\
\quad\quad y \in b' - a' \\
\quad \text{THEN} \\
\quad\quad b' := b' - \{y\} \\
\quad \text{END}
\end{array}
$$

Notice the non-deterministic choice of an element y in these events. The side conditions are easy to establish: clearly the cardinals of sets a' and b' do decrease. We now make the two previous events more deterministic by choosing y to be the minimum of the corresponding set. Event aparm is left unchanged, whereas event aprog, rmv_a , and rmv_b are made completely deterministic.

$$
\begin{array}{l}
\mathsf{aparm} \;\widehat{=} \\
\quad \text{SELECT} \\
\quad\quad d = 0 \\
\quad \text{THEN} \\
\quad\quad a', b' : \begin{pmatrix} a' \in \mathbb{F}(\mathbb{N}) \;\wedge \\ b' \in \mathbb{F}(\mathbb{N}) \;\wedge \\ a' \cap b' \neq \emptyset \end{pmatrix} \quad || \\
\quad\quad d := 1 \\
\quad \text{END}
\end{array}
\qquad
\begin{array}{l}
\mathsf{aprog} \;\widehat{=} \\
\quad \text{SELECT} \\
\quad\quad d \neq 0 \quad \wedge \\
\quad\quad \mathsf{min}\,(a') \in b' \\
\quad \text{THEN} \\
\quad\quad x, d := \mathsf{min}\,(a'), 0 \\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{rmv_a} \;\widehat{=} \\
\quad \text{SELECT} \\
\quad\quad d \neq 0 \quad \wedge \\
\quad\quad \mathsf{min}\,(a') \notin b' \\
\quad \text{THEN} \\
\quad\quad a' := a' - \{\mathsf{min}\,(a')\} \\
\quad \text{END}
\end{array}
\qquad
\begin{array}{l}
\mathsf{rmv_b} \;\widehat{=} \\
\quad \text{SELECT} \\
\quad\quad d \neq 0 \quad \wedge \\
\quad\quad \mathsf{min}\,(b') \notin a' \\
\quad \text{THEN} \\
\quad\quad b' := b' - \{\mathsf{min}\,(b')\} \\
\quad \text{END}
\end{array}
$$

The membership tests of each minimum in the guards of the last three events can be realized by means of a comparison with the other minimum, yielding

$$
\begin{array}{l}
\mathsf{aparm} \;\;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d = 0\\
\quad \text{THEN}\\
\qquad a', b' : \left(\begin{array}{l} a' \in \mathbb{F}(\mathbb{N}) \;\wedge \\ b' \in \mathbb{F}(\mathbb{N}) \;\wedge \\ a' \cap b' \neq \emptyset \end{array}\right) \quad || \\
\qquad d := 1\\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{aprog} \;\;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d \neq 0 \quad \wedge\\
\qquad \mathsf{min}\,(a') = \mathsf{min}\,(b')\\
\quad \text{THEN}\\
\qquad x, d := \mathsf{min}\,(a'), 0\\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{rmv_a} \;\;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d \neq 0 \quad \wedge\\
\qquad \mathsf{min}\,(a') < \mathsf{min}\,(b')\\
\quad \text{THEN}\\
\qquad a' := a' - \{\mathsf{min}\,(a')\}\\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{rmv_b} \;\;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d \neq 0 \quad \wedge\\
\qquad \mathsf{min}\,(b') < \mathsf{min}\,(a')\\
\quad \text{THEN}\\
\qquad b' := b' - \{\mathsf{min}\,(b')\}\\
\quad \text{END}
\end{array}
$$

Clearly the guards are strengthened. We now decide to implement both sets a and b by means of two *ascending and injective* sequences f and g. The size of these sequences are m and n respectively. We also introduce two more variables i and j in such a way that a' and b' are respectively $f\,[i\,..\,m]$ and $g\,[j\,..\,n]$ (this is the gluing invariant). This implies immediately that $\mathsf{min}\,(a')$ is equal to $f(i)$ and $\mathsf{min}\,(b')$ is equal to $g(j)$. It also implies that removing $\mathsf{min}\,(a')$ from a' or $\mathsf{min}\,(b')$ from b' just corresponds to incrementing i or j.

$$
\begin{array}{l}
m \in \mathbb{N}\\[1em]
n \in \mathbb{N}\\[1em]
f \in 1\,..\,m \rightarrowtail \mathbb{N}\\[1em]
g \in 1\,..\,n \rightarrowtail \mathbb{N}\\[1em]
\forall\,(k,l)\cdot(\,k \in \mathsf{dom}\,(f) \;\;\wedge\;\; l \in \mathsf{dom}\,(f) \;\;\wedge\;\; k \leq l \;\;\Rightarrow\;\; f(k) \leq f(l)\,)\\[1em]
\forall\,(k,l)\cdot(\,k \in \mathsf{dom}\,(g) \;\;\wedge\;\; l \in \mathsf{dom}\,(g) \;\;\wedge\;\; k \leq l \;\;\Rightarrow\;\; g(k) \leq g(l)\,)\\[1em]
i \in 1\,..\,m\\[1em]
j \in 1\,..\,n\\[1em]
a' = f\,[i\,..\,m]\\[1em]
b' = g\,[j\,..\,n]
\end{array}
$$

We obtain the following refinements

```
aparm  ≙
  SELECT
    d = 0
  THEN
                    ⎛ m ∈ ℕ                                                                  ⎞
                    ⎜ n ∈ ℕ                                                                  ⎟
                    ⎜ f ∈ 1..m → ℕ                                                           ⎟
    m, n, f, g :    ⎜ g ∈ 1..n → ℕ                                                           ⎟   ||
                    ⎜ ∀(k,l)·( k ∈ dom(f)  ∧  l ∈ dom(f)  ∧  k ≤ l  ⇒  f(k) ≤ f(l) )         ⎟
                    ⎜ ∀(k,l)·( k ∈ dom(g)  ∧  l ∈ dom(g)  ∧  k ≤ l  ⇒  g(k) ≤ g(l) )         ⎟
                    ⎝ ran(f) ∩ ran(g) ≠ ∅                                                    ⎠
    i, j, d := 1, 1, 1
  END
```

```
aprog  ≙                 rmv_a  ≙                 rmv_b  ≙
  SELECT                   SELECT                   SELECT
    d ≠ 0  ∧                 d ≠ 0  ∧                 d ≠ 0  ∧
    f(i) = g(j)              f(i) < g(j)              g(j) < f(i)
  THEN                     THEN                     THEN
    x, d := f(i), 0          i := i + 1               j := j + 1
  END                      END                      END
```

We can rearrange equivalently the guards of these events as follows:

```
aprog  ≙                 rmv_a  ≙                 rmv_b  ≙
  SELECT                   SELECT                   SELECT
    d ≠ 0  ∧                 d ≠ 0  ∧                 d ≠ 0  ∧
    f(i) = g(j)              f(i) ≠ g(j)              f(i) ≠ g(j)
  THEN                       f(i) < g(j)              g(j) ≤ f(i)
    x, d := f(i), 0        THEN                     THEN
  END                        i := i + 1               j := j + 1
                           END                      END
```

Putting the last two together by means of RULE 2, we obtain

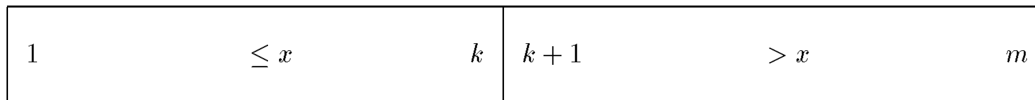
```
aprog  ≙                 rmv_ab  ≙
  SELECT                   SELECT
    d ≠ 0  ∧                 d ≠ 0  ∧
    f(i) = g(j)              f(i) ≠ g(j)
  THEN                     THEN
    x, d := f(i), 0          IF  f(i) < g(j)  THEN
  END                          i := i + 1
                             THEN
                               j := j + 1
                             END
                           END
```

yielding, by applying RULE 4

```
cprog  ≙
  SELECT
    d ≠ 0
  THEN
    WHILE  f(i) ≠ g(j)  DO
      IF  f(i) < g(j)  THEN
        i := i + 1
      ELSE
        j := j + 1
      END
    END ;
    x, d := f(i), 0
  END
```

Example 6: Array Partitioning

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



Note that in case all elements of f are all 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 m. We have five variables m, f, x, k and d with the following invariant:

$$m \in \mathbb{N}_1$$

$$f \in 1\,..\,m \rightarrowtail \mathbb{N}$$

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

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

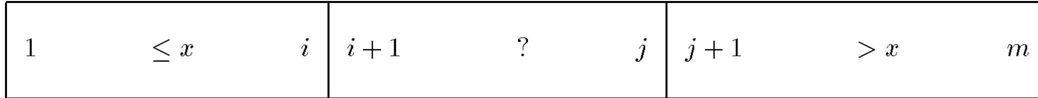
$$d \in \{0, 1\}$$

Note that in the sequel $\overline{0\,..\,x}$ is a shorthand for $\mathbb{N} - (0\,..\,x)$. We have the events aparm and aprog defined as follows

$$
\begin{array}{l}
\textsf{aparm} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\qquad d = 0 \\
\quad \text{THEN} \\
\qquad m, f, x : \begin{pmatrix} m \in \mathbb{N}_1 \\ f \in 1\,..\,m \rightarrowtail \mathbb{N} \\ x \in \mathbb{N} \end{pmatrix} \quad || \\
\qquad d := 1 \\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\textsf{aprog} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\qquad d \neq 0 \\
\quad \text{THEN} \\
\qquad k, f : \begin{pmatrix} k \in 0\,..\,m \\ f \in 1\,..\,m \rightarrowtail \mathbb{N} \\ \mathsf{ran}\,(f) = \mathsf{ran}\,(f_0) \\ f[1\,..\,k] \;\subseteq\; 0\,..\,x \\ f[k+1\,..\,m] \;\subseteq\; \overline{0\,..\,x} \end{pmatrix} \quad || \\
\qquad d := 0 \\
\quad \text{END}
\end{array}
$$

We remind the reader that a variable indexed by 0 (as in f_0 above) denotes by definition its "before-value". Our next step is to introduce two variables i and j partitioning the sequence f' (replacing f) as indicated by the following diagram:



The idea is then to possibly increment i and decrement j while maintaining the corresponding invariant. When the process is completed then the concrete f' and abstract f should be the same. More formally, this yields:

$$
\begin{array}{l}
i \in 0\,..\,m \;\;\wedge\;\; j \in 0\,..\,m \;\;\wedge\;\; i \leq j \\[6pt]
f' \in 1\,..\,m \rightarrowtail \mathbb{N} \\[6pt]
\mathsf{ran}\,(f') = \mathsf{ran}\,(f) \\[6pt]
f'[1\,..\,i] \;\subseteq\; 0\,..\,x \\[6pt]
f'[j+1\,..\,m] \;\subseteq\; \overline{0\,..\,x} \\[6pt]
d = 0 \;\;\Rightarrow\;\; f' = f
\end{array}
$$

We have three new events called $\mathsf{progress_1}$, $\mathsf{progress_2}$ and swap.

$$
\begin{array}{l}
\mathsf{aparm} \;\;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d = 0\\
\quad \text{THEN}\\
\qquad m, f', x : \begin{pmatrix} m \in \mathbb{N}_1 \\ f' \in 1\,..\,m \rightarrowtail \mathbb{N} \\ x \in \mathbb{N} \end{pmatrix} \quad ||\\
\qquad i, j, d := 0, m, 1\\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{aprog} \;\;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d \neq 0 \;\wedge\; i = j\\
\quad \text{THEN}\\
\qquad k, d := i, 0\\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{progress_1} \;\;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d \neq 0 \;\wedge\; i \neq j \;\wedge\; f'(i+1) \leq x\\
\quad \text{THEN}\\
\qquad i := i + 1\\
\quad \text{END}
\end{array}
$$

In the next event, the guard $f'(i+1) > x$ is not strictly speaking necessary. We have introduced it however in order to have each guard being the negation of another one (in this way, we can always apply RULE 2 as can be seen below).

$$
\begin{array}{l}
\mathsf{progress_2} \;\;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d \neq 0 \;\wedge\; i \neq j \;\wedge\; f'(i+1) > x \;\wedge\; f'(j) > x\\
\quad \text{THEN}\\
\qquad j := j - 1\\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{swap} \;\;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d \neq 0 \;\;\wedge\;\; i \neq j \;\;\wedge\;\; f'(i+1) > x \;\;\wedge\;\; f'(j) \leq x\\
\quad \text{THEN}\\
\qquad i, j := i + 1, j - 1 \quad ||\\
\qquad f' := f' \Leftarrow \{i + 1 \mapsto f'(j)\} \Leftarrow \{j \mapsto f'(i+1)\}\\
\quad \text{END}
\end{array}
$$

It can be proved that aparm and aprog refine their respective abstractions and that progress_1, progress_2, and swap all refine skip. They also decrease the natural number quantity $j - i$. By putting together events progress_1, progress_2, and swap (using RULE 2), we obtain:

```
progress_swap  ≙
  SELECT
    d ≠ 0  ∧  i ≠ j
  THEN
    IF  f′(i + 1) ≤ x  THEN
      i := i + 1
    ELSIF  f′(j) > x  THEN
      j := j − 1
    ELSE
      i, j := i + 1, j − 1    ||
      f′ := f′ ⩤ {i + 1 ↦ f′(j)} ⩤ {j ↦ f′(i + 1)}
    END
  END
```

By applying now RULE 4 to events progress_swap and aprog, we obtain the following:

```
cprog  ≙
  SELECT
    d ≠ 0
  THEN
    WHILE  i ≠ j  DO
      IF  f′(i + 1) ≤ x  THEN
        i := i + 1
      ELSIF  f′(j) > x  THEN
        j := j − 1
      ELSE
        i, j := i + 1, j − 1    ||
        f′ := f′ ⩤ {i + 1 ↦ f′(j)} ⩤ {j ↦ f′(i + 1)}
      END
    END ;
    k, d := i, 0
  END
```

Example 7: In place Reversing of an Array

Our next example is the classical "in place" reversing of an array. Given a set S, we have three variables m, f, and d where

$$m \in \mathbb{N}_1$$

$$f \in 1\,..\,m \rightarrow S$$

$$d \in \{0, 1\}$$

Our event aparm is as follows:

$$
\begin{array}{l}
\mathsf{aparm} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\qquad d = 0 \\
\quad \text{THEN} \\
\qquad m, f : \begin{pmatrix} m \in \mathbb{N}_1 & \wedge \\ f \in 1\,..\,m \rightarrow S \end{pmatrix} \quad \| \\
\qquad d := 1 \\
\quad \text{END}
\end{array}
$$

The event aprog reverses the original array f_0.

$$
\begin{array}{l}
\mathsf{aprog} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\qquad d \neq 0 \\
\quad \text{THEN} \\
\qquad f : \begin{pmatrix} f \in 1\,..\,m \rightarrow S \quad \wedge \\ \forall k \cdot (\, k \in 1\,..\,m \;\; \Rightarrow \;\; f(k) = f_0(m - k + 1)\,) \end{pmatrix} \quad \| \\
\qquad d := 0 \\
\quad \text{END}
\end{array}
$$

The next transformation consists in introducing two indices i and j starting at 1 and m respectively and moving towards each other. The sequence f' (a new variable of this refinement replacing f) is gradually reversed by swapping elements $f'(i)$ and $f'(j)$ while, of course, i is strictly smaller than j. This is done in a new event called swap. In this way, the sub-sequences of f' ranging from 1 to $i-1$ and from $j+1$ to m respectively have all their elements reversed with regard to the original sequence f. Notice that the quantity $i+j$ is always equal to $m+1$. At the end of the process either i is equal to j when m is odd, or i is equal to $j+1$ when m is even (but, in both case, we have $i \geq j$). Here is the new invariant

$$
\begin{array}{l}
f' \in 1\,..\,m \rightarrow S \\
i \in 1\,..\,m \\
j \in 1\,..\,m \\
i + j = m + 1 \\
i \leq j + 1 \\
\forall k \cdot (\, k \in 1\,..\,i-1 \;\; \Rightarrow \;\; f'(k) = f(m - k + 1)\,) \\
\forall k \cdot (\, k \in i\,..\,j \;\; \Rightarrow \;\; f'(k) = f(k)\,) \\
\forall k \cdot (\, k \in j+1\,..\,m \;\; \Rightarrow \;\; f'(k) = f(m - k + 1)\,) \\
d = 0 \;\; \Rightarrow \;\; f' = f
\end{array}
$$

Here are the refined events:

```
aparm  ≙
  SELECT
    d = 0
  THEN
    m, f', i, j : ( m ∈ ℕ1   ∧
                    f' ∈ 1 .. m → S   ∧
                    i = 1   ∧
                    j = m )                ||
    d := 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0  ∧  i ≥ j
  THEN
    d := 0
  END
```

```
swap  ≙
  SELECT
    d ≠ 0  ∧  i < j
  THEN
    f' := f' <+ {i ↦ f'(j)} <+ {j ↦ f'(i)}   ||
    i, j := i + 1, j − 1
  END
```

Applying the usual rules to aprog and swap, we obtain the following final program cprog:

```
cprog  ≙
  SELECT
    d ≠ 0
  THEN
    WHILE  i < j  DO
      f' := f' <+ {i ↦ f'(j)} <+ {j ↦ f'(i)}   ||
      i, j := i + 1, j − 1
    END ;
    d := 0
  END
```

Example 8: In Place Reversing of a Linear Chain

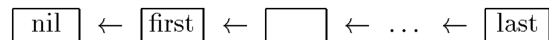
So far all our examples were dealing with arrays and corresponding indices. As a consequence some of the proofs heavily relied on (elementary) arithmetic properties. In this example, we experiment our approach on a data structure that deals with "pointers".

The problem we shall tackle is very classical and simple: we just want to reverse a chained linear list. Notice that to simplify matters the chain we consider is made of pointers only (it has no information "field").

Each element in the list "points" to its immediate successor. The list starts with an element called f (for "first") and ends up with an element called l (for "last"). Notice that f and l might be equal, so that the list has (at least) one element. By convention, l points to a "special element" called nil which is *not considered to be part of the list*. All this can be

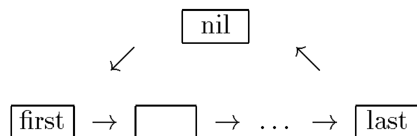
represented in the following diagram:

We would like to reverse that list in such a way that l becomes the first element of the result and f its last element (pointing, of course, to *nil*). Thus the transformed list looks like this:

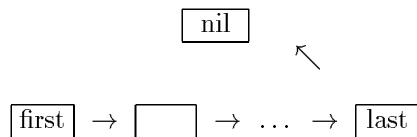


Clearly it would have been simpler to transform our initial list into the following one

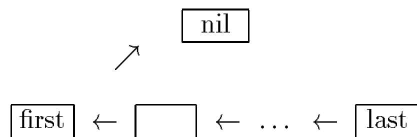
because then the specification of our problem would have been straightforward : representing the "pointer structure" by a certain injective function, the final result would have been the inverse of it. Unfortunately, the problem is different, we have thus to find a better abstraction. The idea then is to consider that *nil* might point to f as indicated in the following diagram, representing a "circular list" (a cycle) :



This is, of course, different from our initial list, but this is just an abstraction. In fact, our initial list is as follows (it is obtainable from the cycle by removing the link $nil \mapsto f$)



and our transformed list is as follows (likewise it is obtainable from the inverse of the cycle by removing the link $nil \mapsto l$):



Our next problem is then to formally represent a cycle. Let S be the set made of all the elements of the cycle (including *nil* thus). Of course, the set S is supposed to be *finite*. In first approximation, the circular list can be represented by a bijection h from S to S. But there are many bijections, which are not forming circular chains: these are those having some internal sub-cycles. So our next problem is to define the characteristic property of such sub-cycles. If a subset A of S forms a cycle under h then, clearly, the image of A under h is exactly A (a property also shared by the empty set). As a consequence the only possibility for h to have a single outermost cycle is that the subsets A of S having that property (that is, $h[A] = A$) are just S itself and, of course, the empty set. Our circular list is thus formalized as follows:

$$h \in S \rightarrowtail\!\!\!\!\rightarrow S$$

$$\forall A \cdot (A \subseteq S \;\wedge\; A \neq S \;\wedge\; h[A] = A \;\Rightarrow\; A = \emptyset) \qquad \textbf{(1)}$$

We now define our three constants f, l, and nil as follows

$$l \in S$$

$$nil = h(l)$$

$$f = h(nil)$$

Our result variable r is just typed as a relation from S to S

$$r \in S \leftrightarrow S$$

We have then the two usual events aparm and aprog. Notice that the result r in aprog is obtained, as announced above, by taking the inverse of the cycle h with the link $nil \mapsto l$ removed.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    d := 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0
  THEN
    r := h⁻¹ ⩥ {l}   ||
    d := 0
  END
```

Our eventual goal is to construct the classical "in place" reversing of the pointers. But this is too big a step right now. We shall first go through various abstractions before reaching that stage. The next step proceeds with three variables : r' denotes the reverse list in formation (it replaces the more abstract r), p is a pointer moving through the list (it starts at nil !), and T is a subset of S (which will not be implemented eventually) representing the various elements whose links still have to be reversed. Here are the declarations of these variables:

$$r' \in S \leftrightarrow S$$

$$T \subseteq S$$

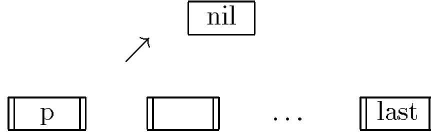
$$p \in S$$

The dynamic situation can be depicted on the following diagrams where a double box means membership to T. The arrows corresponds to the pairs composing the function r'. At the beginning, T covers the entire set S, p is equal to nil, and r' is empty.

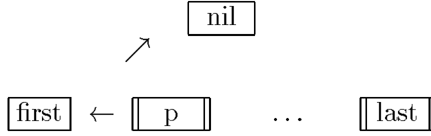
p

first ... last

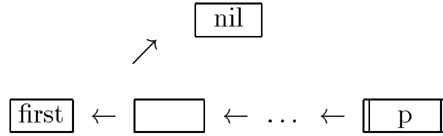
At each step, p is removed from T and moved to $h(p)$. Moreover, the link from $h(p)$ to p is added to r'. So, after the first step we have the following situation:



After the second step, we have thus the following



The process does stop when we reach the following situation where p is equal to l. We can figure out on the diagram that, in this case, T is also exactly equal to $\{l\}$.



From this diagrammatic description, we can "see" a number of invariant properties. First, p and l are always members of T. Second, the image of the set $T - \{l\}$ under h is exactly $T - \{p\}$ (a "fixpoint" is thus obtained when p is equal to l). In other words, p is the "smallest" element of T with respect to the relation h. Third, the variable r' is exactly h^{-1} with links ending in T removed. Finally when the condition $d = 0$ holds (when the process is completed) then r' is exactly equal to the abstraction r (r' satisfies then our abstract specification). Formally

$$
\begin{array}{ll}
p \in T & \\
l \in T & \\
h[T - \{l\}] \ = \ T - \{p\} & \quad \mathbf{(2)} \\
r' = h^{-1} \rhd T & \\
d = 0 \ \Rightarrow \ r' = r &
\end{array}
$$

The property of p (that is, $p \in T$) implies that when T is equal to $\{l\}$, then p clearly is equal to l. Conversely, when p is equal to l then we have $h[T - \{l\}] = T - \{l\}$ according to **(2)**, thus $T - \{l\}$ is empty according to **(1)** (since $T - \{l\}$ is clearly not equal to S). As a consequence, the set T is also equal to $\{l\}$ in that case since l always belongs to T ($l \in T$ being an invariant). We have thus informally proved the following equivalence:

$$T = \{l\} \;\Leftrightarrow\; p = l$$

We now refine aparm and aprog and we introduce the new event progress.

```
aparm  ≙
  SELECT
    d = 0
  THEN
    r' := ∅    ||
    T := S     ||
    p := nil   ||
    d := 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0 ∧ p = l
  THEN
    d := 0
  END
```

```
progress  ≙
  SELECT
    d ≠ 0 ∧ p ≠ l
  THEN
    r' := r' ∪ {h(p) ↦ p}   ||
    T := T − p   ||
    p := h(p)
  END
```

The refinement proofs are not difficult. Notice that progress decreases the cardinal of the set T since p always belongs to T. Our next step proceeds by throwing the set T, which just appears thus to be an *abstract artifact* needed to perform the various proofs. We also introduce an extra pointer q, which is equal to $h(p)$.

$$q = h(p)$$

The test $p = l$ is then clearly equivalent to $q = nil$ since nil is equal, by definition, to $h(l)$ and since h is injective (that is, we have indeed $p = l \;\Leftrightarrow\; h(p) = h(l)$). Formally

$$p = l \;\Leftrightarrow\; q = nil$$

Next are the three refinements of our events:

```
aparm  ≙
  SELECT
    d = 0
  THEN
    r' := ∅ ||
    p := nil ||
    q := f ||
    d := 1
  END
```

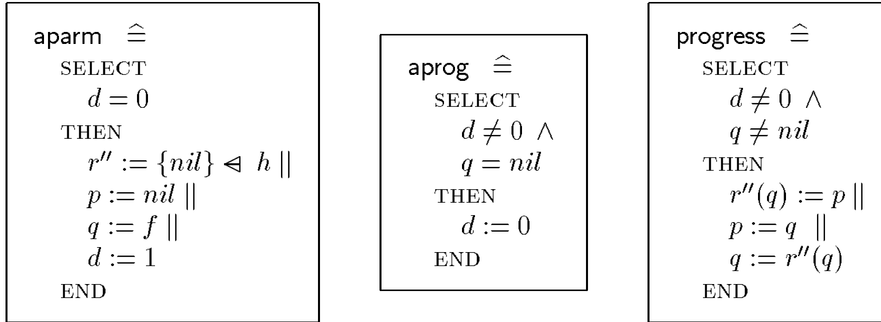
```
aprog  ≙
  SELECT
    d ≠ 0 ∧
    q = nil
  THEN
    d := 0
  END
```

```
progress  ≙
  SELECT
    d ≠ 0 ∧
    q ≠ nil
  THEN
    r' := r' ∪ {q ↦ p} ||
    p := q  ||
    q := h(q)
  END
```

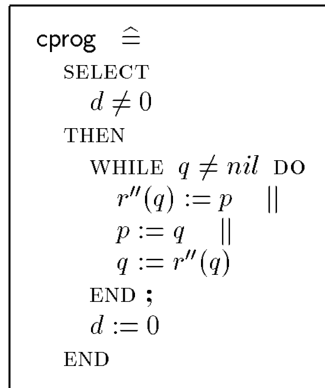
The final touch consists now in doing the “in place” moving of the pointers. For this we replace the variable r' by a new variable r'' defined as follows

$$r'' \;=\; (\{nil\} \lhd h) \mathbin{\lhd\!\!+} r'$$

The three events are modified accordingly. Notice that r'' is indeed initialized with the original list, that is $\{nil\} \lhd h$. Also notice that l is not mentioned any more.



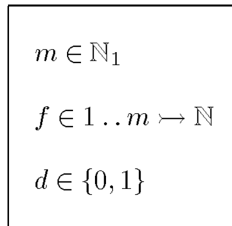
Applying RULE 4 to aprog and progress, we obtain our final loop:



Example 9: Sorting

We are not going to develop a very smart sorting algorithm in this example. Our intention is only to use sorting as an opportunity to develop a little program containing an *embedded loop*. We want to figure out whether this embedding would come *naturally*.

We have a constant m, which is a positive natural number. We also have a variable f, which is a total injective function from the interval $1..m$ to the natural numbers. We finally have the usual variable d :



The two events aparm and aprog are as follows:

$$
\begin{array}{l}
\mathsf{aparm} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\qquad d = 0 \\
\quad \text{THEN} \\
\qquad f :\in 1\,..\,m \rightarrowtail \mathbb{N} \quad \| \\
\qquad d := 1 \\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{aprog} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\qquad d \neq 0 \\
\quad \text{THEN} \\
\qquad f : \begin{pmatrix} f \in 1\,..\,m \rightarrow \mathbb{N} \;\; \land \\ \forall\,(i,j) \cdot (\, i \in 1\,..\,m-1 \;\; \land \;\; j \in 1\,..\,m \;\; \land \;\; i < j \;\; \Rightarrow \;\; f(i) < f(j)\,) \;\; \land \\ \mathsf{ran}\,(f) = \mathsf{ran}\,(f_0) \end{pmatrix} \quad \| \\
\qquad d := 0 \\
\quad \text{END}
\end{array}
$$

The non-deterministic condition in event aprog stipulates that the resulting f is sorted in an ascending way and that it exactly has the same elements as the original f.

We now introduce two variables : first f' that replaces f, and second k, which is an index ranging from 1 to m. We shall suppose, as an invariant, that the sequence f' is indeed sorted in its sub-part ranging from 1 to $k-1$. Moreover the elements of that sub-part are all smaller than or equal than the elements lying in the other sub-part, namely that ranging from k to m. Finally, we have the (gluing) invariant stipulating that f' and f are alike when the process is over (when the condition $d = 0$ holds). Formally :

$$
\begin{array}{l}
f' \in 1\,..\,m \rightarrowtail \mathbb{N} \\[1ex]
\mathsf{ran}\,(f') = \mathsf{ran}\,(f) \\[1ex]
k \in 1\,..\,m \\[1ex]
\forall\,(i,j) \cdot (\, i \in 1\,..\,k-1 \;\; \land \;\; j \in 1\,..\,m \;\; \land \;\; i < j \;\; \Rightarrow \;\; f'(i) < f'(j)\,) \\[1ex]
d = 0 \;\; \Rightarrow \;\; f' = f
\end{array}
$$

Next are the refinement of aparm and aprog.

$$
\begin{array}{l}
\mathsf{aparm} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\qquad d = 0 \\
\quad \text{THEN} \\
\qquad f' :\in 1\,..\,m \rightarrowtail \mathbb{N} \quad \| \\
\qquad k, d := 1, 1 \\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{aprog} \;\; \widehat{=} \\
\quad \text{SELECT} \\
\qquad d \neq 0 \;\; \land \;\; k = m \\
\quad \text{THEN} \\
\qquad d := 0 \\
\quad \text{END}
\end{array}
$$

We also have the new event progr. As can be seen, an index l is chosen "arbitrarily" in the range $k \,..\, m$, with a corresponding value $f'(l)$ that happens to be the minimum of f' in that sub-part. This index is then exchanged with k. Finally, k is incremented.

```
progr  ≙
  SELECT
    d ≠ 0  ∧  k < m
  THEN
    ANY l WHERE
      l ∈ k .. m    ∧
      ∀i · ( i ∈ (k .. m) − {l}  ⇒  f′(l) < f′(i) )
    THEN
      f′ := f′ <+{k ↦ f′(l)} <+{l ↦ f′(k)}    ||
      k := k + 1
    END
  END
```

Our next step consists in introducing two more events whose rôle is to determine the minimum chosen "arbitrarily" above. For this, we introduce two new indices, j and l. The index j ranges from k to m, whereas l ranges from k to j. The value of f' at index l is supposed to be the minimum of f' on the sub-part ranging from k to j, Formally

$$j \in k \,..\, m$$

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

$$\forall i \cdot (\, i \in (k \,..\, j) - \{l\} \;\Rightarrow\; f'(l) < f'(k)\,)$$

Next are the refinements of the abstract events with a few modifications only for the first three:

```
aparm  ≙
  SELECT
    d = 0
  THEN
    f′ :∈ 1 .. m ↣ ℕ    ||
    k, l, j, d := 1, 1, 1, 1
  END
```

```
aprog  ≙
  SELECT
    d ≠ 0  ∧  k = m
  THEN
    d := 0
  END
```

In the concrete event progr, the strengthening of the guard (with condition $j = m$) implies that the value of the variable l corresponds exactly to the minimum chosen arbitrarily in the abstraction.

$$
\begin{array}{l}
\mathsf{progr} \;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d \neq 0 \;\wedge\; k < m \;\wedge\; j = m\\
\quad \text{THEN}\\
\qquad f' := f' \lhd\!\!- \{k \mapsto f'(l)\} \lhd\!\!- \{l \mapsto f'(k)\} \quad ||\\
\qquad k, j, l := k+1, k+1, k+1\\
\quad \text{END}
\end{array}
$$

We have two more events for determining the index l at which the value of the function f' on the interval $k\,..\,m$ is minimum. Notice that in the event prog1, the condition $f'(l) \leq f'(j+1)$ is equivalent to $f'(l) < f'(j+1)$ since $f'(l)$ and $f'(j+1)$ cannot be equal. This is due to the facts that: (1) the variable l (ranging from k to j) and the expression $j+1$ have distinct values, and (2) f is injective.

$$
\begin{array}{l}
\mathsf{prog1} \;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d = 0 \;\wedge\; k < m \;\wedge\; j < m \;\wedge\; f'(l) \leq f'(j+1)\\
\quad \text{THEN}\\
\qquad j := j+1\\
\quad \text{END}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{prog2} \;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d = 0 \;\wedge\; k < m \;\wedge\; j < m \;\wedge\; f'(l) > f'(j+1)\\
\quad \text{THEN}\\
\qquad j, l := j+1, j+1\\
\quad \text{END}
\end{array}
$$

Applying RULE 2, we now put together these events as follows in the event prog12 :

$$
\begin{array}{l}
\mathsf{prog12} \;\widehat{=}\\
\quad \text{SELECT}\\
\qquad d = 0 \;\wedge\; k < m \;\wedge\; j < m\\
\quad \text{THEN}\\
\qquad \text{IF } f'(l) \leq f'(j+1) \text{ THEN}\\
\qquad\quad j := j+1\\
\qquad \text{ELSE}\\
\qquad\quad j, l := j+1, j+1\\
\qquad \text{END}\\
\quad \text{END}
\end{array}
$$

Applying RULE 4 on events prog12 and progr, we obtain the following event body

```
body  ≙
  SELECT
    d ≠ 0  ∧  k < m
  THEN
    WHILE  j < m  DO
      IF  f′(l) ≤ f′(j + 1)  THEN
        j := j + 1
      ELSE
        j, l := j + 1, j + 1
      END
    END ;
    BEGIN
      f′ := f′ ◁+ {k ↦ f′(l)} ◁+ {l ↦ f′(k)}   ||
      k, j, l := k + 1, k + 1, k + 1
    END
  END
```

By applying again RULE 4, this time on events body and aprog, we obtain eventually the following event cprog together with the event aparm :

```
aparm  ≙
  SELECT
    d = 0
  THEN
    f′ :∈ 1 .. m ↣ ℕ   ||
    k, l, j, d := 1, 1, 1, 1
  END
```

```
cprog  ≙
  SELECT
    d ≠ 0
  THEN
    WHILE  k < m  DO
      WHILE  j < m  DO
        IF  f′(l) ≤ f′(j + 1)  THEN
          j := j + 1
        ELSE
          j, l := j + 1, j + 1
        END
      END ;
      BEGIN
        f′ := f′ ◁+ {k ↦ f′(l)} ◁+ {l ↦ f′(k)}   ||
        k, j, l := k + 1, k + 1, k + 1
      END
    END ;
    d := 0
  END
```

Acknowledgments: I would like to thank Louis Mussat for giving me very useful comments and advice. Thanks also to Dominique Cancell for his remarks.