Part I
Informal notions of refinement

Contents

I  Informal notions of refinement  

1  What is Refinement?  

1.1  What does refinement guarantee?  

1.2  Some things you can do in refinement  

1.3  Reducing nondeterminism: examples  

1.4  Weakening the precondition  

1.5  Refinement is not equivalence  

1.6  Refining the state  

1.7  The Array machine  

1.8  A refinement of the Array machine  

1.9  A function machine  

1.10  A refinement of the function machine  

II  Formalisation of refinement  

2  Towards a formal understanding of refinement
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Coin flip machine and refinement</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Operation refinement</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Simple refinement intuition</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Checking our intuition</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>The effect of nondeterminism</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>A revised formalisation of refinement</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Isolating the problem</td>
<td>11</td>
</tr>
<tr>
<td>2.8</td>
<td>Final formulation</td>
<td>11</td>
</tr>
<tr>
<td>2.9</td>
<td>Validating Flip under the new formulation</td>
<td>12</td>
</tr>
<tr>
<td>2.10</td>
<td>Notes on the conjugate weakest precondition</td>
<td>12</td>
</tr>
<tr>
<td>2.11</td>
<td>Refinement and feasibility</td>
<td>12</td>
</tr>
<tr>
<td>2.12</td>
<td>Avoiding the infeasible</td>
<td>13</td>
</tr>
<tr>
<td>2.13</td>
<td>Formality does not guarantee feasibility</td>
<td>13</td>
</tr>
<tr>
<td>2.14</td>
<td>Proving feasibility</td>
<td>14</td>
</tr>
<tr>
<td>2.15</td>
<td>The Infeasible cannot be made Feasible</td>
<td>14</td>
</tr>
</tbody>
</table>

III The final refinement step: Implementation 14

3 Implementation 15

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A Simple Complete Development</td>
<td>15</td>
</tr>
<tr>
<td>3.1.1</td>
<td>A square operation</td>
<td>15</td>
</tr>
<tr>
<td>3.1.2</td>
<td>A square root operation</td>
<td>15</td>
</tr>
<tr>
<td>3.1.3</td>
<td>The use of non-determinism in specification</td>
<td>16</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Implementation of Square</td>
<td>16</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Implementing ApproxSqrt</td>
<td>16</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Refining SqrtUtility</td>
<td>18</td>
</tr>
<tr>
<td>3.1.7</td>
<td>The first refinement SqrtUtilityR</td>
<td>18</td>
</tr>
<tr>
<td>3.1.8</td>
<td>The second refinement SqrtUtilityRR</td>
<td>19</td>
</tr>
<tr>
<td>3.1.9</td>
<td>Finally the implementation SqrtUtilityRRR</td>
<td>19</td>
</tr>
<tr>
<td>3.1.10</td>
<td>Generating code</td>
<td>20</td>
</tr>
<tr>
<td>3.1.11</td>
<td>Generating an interface</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>A second complete implementation</td>
<td>21</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Modelling of queue</td>
<td>22</td>
</tr>
<tr>
<td>3.2.2</td>
<td>The Queue machine</td>
<td>22</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Refining the Queue machine</td>
<td>23</td>
</tr>
<tr>
<td>3.2.4</td>
<td>The invariant of QueueR</td>
<td>25</td>
</tr>
<tr>
<td>3.2.5</td>
<td>The refinement relation of QueueR</td>
<td>25</td>
</tr>
</tbody>
</table>
Objectives of this Lecture

- to introduce the concept of refinement, both algorithmic and data refinement;
- to understand the concept of refinement both informally and formally;
- to explore a number of examples of refinement;

1 What is Refinement?

Refinement is the name given to the process of transforming an abstract specification into a concrete implementation.

There are two aspects of refinement:

Algorithmic refinement, in which an algorithm is transformed, and

Data refinement, in which the variables are transformed.

Both forms of refinement are required in general to take abstract variables to a concrete form that can be implemented. For obvious reasons, data refinement requires algorithmic refinement.

In general, we will use the term refinement to cover either or both.

1.1 What does refinement guarantee?

The refinement of a construct—for example an operation—promises behaviour that is consistent with the behaviour of the construct being refined.
That is, in the context, the behaviour offered by the refining construct could have been offered by the refined construct.

The context may depend on the state of the construct and the precondition.

Consistency must take nondeterminism into account.

1.2 Some things you can do in refinement

- Reduce nondeterminism: nondeterminism in a construct is interpreted as a choice in which any of the outcomes are satisfactory, so the refiner can choose between those options.

- Weaken the precondition: given that the behaviour of an refined operation is specified on the assumption of the precondition, the refinement can do anything outside of the precondition.

1.3 Reducing nondeterminism: examples

**Specification**

\[
\text{result} \leftarrow \text{SimpleChoice} \stackrel{\triangle}{=} \text{result} : \in \{3, 6, 9, 12\} ;
\]

**Refinement**

\[
\text{result} \leftarrow \text{SimpleChoice} \stackrel{\triangle}{=} \text{result} := 6 ;
\]

1.4 Weakening the precondition

**Specification**

\[
\text{result} \leftarrow \text{Divide} (n1, n2) \stackrel{\triangle}{=}
\]

\[
\text{pre} \; n1 \in \mathbb{N} \land n2 \in \mathbb{N} \land n2 \neq 0 \; \text{then} \quad \text{result} := n1 / n2
\]

\[
\text{end} ;
\]

**Refinement**

\[
\text{result} \leftarrow \text{Divide} (n1, n2) \stackrel{\triangle}{=}
\]

\[
\text{if} \; n2 \neq 0 \quad \text{then} \; \text{result} := n1 / n2
\]

\[
\text{else} \; \text{result} := 42
\]

\[
\text{end} ;
\]

1.5 Refinement is not equivalence

It is important to understand that refined behaviour is not equivalent behaviour, as the following should make clear.

The COIN set
SETS COIN = { Head, Tail }

Specification
coin ← Flip \( \equiv \)

Refinement
coin ← Flip \( \equiv \)

It should be clear that it is reasonable for an operation to be refined by itself, but it should also be clear that the two independent coin flips are not guaranteed to produce equivalent behaviour.

But the behaviour of each is consistent with the possible behaviour of the other.

1.6 Refining the state

The refining machine can have its own state, which in some way simulates the state of the refined machine.

The invariant of the refining machine has two components:

1. the constraints on its own state variables as for any other machine;
2. a refinement relation that describes how the refining machine’s state simulates the state of the refined machine.

Consider the following examples for a simple array machine.

1.7 The Array machine

The array machine models an array as a partial function. This is probably the way an array behaves in a programming language: no element of the array is defined until a value is assigned to it.

MACHINE Array ( maxarray, VAL )
CONSTRAINTS maxarray \( \in \mathbb{N} \)
VARIABLES array
INARIANT array \( \in 1 \ldots \text{maxarray} \rightarrow \text{VAL} \)
INITIALISATION array := {}
1.8 A refinement of the Array machine

In the refinement the array is represented as a total function. This mimics the situation on a real machine where an array may be implemented as a contiguous area of main storage. In that case every element of the array has a value whether the programmer assigned one or not.

The refinement relation shows the partial function as a subset of the total function. This implies that everywhere in the domain of the partial function the value will be given by the total function.

**REFINEMENT** \( Array_R \)
**REFINES** \( Array \)
**VARIABLES** \( array_r \)

**INCOMPLETE**

\( array_r \in 1 \ldots \text{maxarray} \rightarrow \text{VAL} \land \)

Refinement relation

\( array \subseteq array_r \)

**INITIALISATION** \( array_r : \in 1 \ldots \text{maxarray} \rightarrow \text{VAL} \)

**OPERATIONS**

\[ \text{store} (\ pos, \ val ) \triangleq \]

\[ arrayr (\ pos ) := val ; \]

\[ \text{val} \leftarrow \text{get} (\ pos ) \triangleq \]

\[ \text{val} := arrayr (\ pos ) \]

END

1.9 A function machine

**MACHINE** \( \text{Function} (\ maxdom ) \)
**CONSTRAINTS** \( maxdom \in \mathbb{N}_1 \)
**SETS** \( \text{DOM} ; \text{RAN} \)
**PROPERTIES** \( \text{card} (\ \text{DOM} ) = \text{maxdom} \)
**VARIABLES** \( \text{fun} \)
**INCOMPLETE**

\( \text{fun} \in \text{DOM} \rightarrow \text{RAN} \)
**INITIALISATION** \( \text{fun} := \{} \)

**OPERATIONS**

\[ \text{update} (\ dval, \ rval ) \triangleq \]

\[ \text{pre} \ dval \in \text{DOM} \land rval \in \text{RAN} \text{ then} \]

\[ \text{fun} (\ dval ) := rval \]
end ;
\texttt{rval} \leftarrow \texttt{fetch (dval)} \triangleq \begin{aligned}
\text{pre} & \text{ dval } \in \text{DOM} \land dval \in \text{dom (fun)} \text{ then} \\
\text{rval} & := \text{fun (dval)} \\
\text{end} \end{aligned}

\text{END}

1.10 A refinement of the function machine

Functions are commonly used concepts and there are many algorithms, that are, essentially, concerned with implementing function application.

Although arrays can be viewed as functions, the important property of an array is that it has a coherent domain of natural numbers. Generally, the domain of a function will not be coherent and in many cases consists of values from some opaque set. Thus, while an array can be simply mapped onto computer storage, a function generally cannot.

The strategy we adopt here is to store the domain of the function in an injective sequence and the range in a parallel sequence as shown in FunctionR.

\textbf{REFINEMENT} FunctionR
\textbf{REFINES} Function

\textbf{VARIABLES} fundom, funran

\textbf{INVARIANT}
\begin{align*}
\text{fundom} & \in \text{iseq (DOM)} \land \\
\text{funran} & \in \text{seq (RAN)} \land \\
\text{size (fundom)} & = \text{size (funran)} \land
\end{align*}

Refinement relation
\begin{align*}
\text{fun} & = (\text{fundom}^{-1} ; \text{funran}) \\
\text{ASSERTIONS}
\begin{align*}
\text{dom (fundom)} & = \text{dom (funran)} \land \\
\text{dom (fun)} & = \text{ran (fundom)} \land \\
\text{ran (fun)} & = \text{ran (funran)} \land \\
\text{dom (funran)} & = \text{ran (fundom}^{-1})
\end{align*}

\textbf{INITIALISATION} fundom, funran := [ ] , [ ]

\textbf{OPERATIONS}
\begin{align*}
\text{update (dval, rval)} & \triangleq \begin{aligned}
\text{begin} & \\
\text{if} & \text{ dval } \notin \text{ran (fundom)} \\
\text{then} & \text{ fundom } := \text{fundom } \leftarrow \text{dval} \\
\text{end} \\
\end{aligned}
\end{align*}
\[
\text{funran} \ (\text{fundom}^{-1} \ (\text{dval})) := \text{rval}
\]
\[
\text{end} ;
\]
\[
\text{rval} \leftarrow \text{fetch} \ (\text{dval}) \triangleq
\]
\[
\text{rval} := \text{funran} \ (\text{fundom}^{-1} \ (\text{dval}))
\]

END

Part II

Formalisation of refinement

Contents

2 Towards a formal understanding of refinement

In this section we are going to explore the formalisation of the notion of refinement that has been described in section 2.

We will start with operations that change the state, but have no results. Then we will add results. To start our exploration we will use the simple coin flip operation recast as an operation that changes the state.

2.1 Coin flip machine and refinement

MACHINE Flip
SETS COIN = \{Head, Tail\}
VARIABLES coinA
INVARIANT coinA ∈ COIN
INITIALISATION coinA :∈ COIN
OPERATIONS
Flip \triangleq coinA ∈ COIN
END

REFINEMENT FlipR
REFINES Flip
VARIABLES coinC
INVARIANT
coinC ∈ COIN \land
Refinement relation
coinC = coinA
INITIALISATION coinC :∈ COIN
OPERATIONS
Flip \triangleq coinC ∈ COIN
END

2.2 Operation refinement

The refined operation will be referred to as the abstract operation and the refining operation will be referred to as the concrete operation.
Without any loss of generality we will assume that the states $v_A$ and $v_C$ are disjoint.

### 2.3 Simple refinement intuition

![Initial refinement intuition](image)

A simple intuition about refinement is shown in figure 1.

This leads to the following mathematical formulation of refinement

$$ I_A \land I_C \land R \land P \Rightarrow [B_C ; B_A](I_A \land I_C \land R) \quad (1) $$

### 2.4 Checking our intuition

We will check our current intuition for the Flip operations as presented in Flip and FlipR.

We have to prove:

$$ coinA \in COIN \land coinC \in COIN \land coinC = coinA $$

$$ \Rightarrow [coinC \in COIN \land coinA \in COIN] $$

$$ (coinA \in COIN \land coinC \in COIN \land coinC = coinA) $$
To simplify the proof we will omit $coinA \in COIN$ and $coinC \in COIN$

\[
\begin{align*}
coinC &= coinA \\
\Rightarrow & [coinC : \in \text{COIN} \mid coinA : \in \text{COIN}] (coinC = coinA) \\
\Rightarrow & [coinC : \in \text{COIN}][coinA : \in \text{COIN}] (coinC = coinA) \\
\Rightarrow & [coinC : \in \text{COIN}][[coinA := \text{Head}][coinC = coinA] \land
[coinA := \text{Tail}][\text{coinC} = \text{coinA}]) \\
\Rightarrow & [coinC : \in \text{COIN}][\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \\
\Rightarrow & [\text{coinC} := \text{Head}][\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \land
[\text{coinC} := \text{Tail}][\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \\
\Rightarrow & (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \\
\Rightarrow & (\text{Head} = \text{Tail}) \\
\Rightarrow & false
\end{align*}
\]

Thus, it appears that our initial intuition doesn’t work.

*What went wrong?*

2.5 The effect of nondeterminism

The problem is, we have failed to take into account the effect of nondeterminism.

Both the abstract and concrete operations may be nondeterministic.

In our initial intuition (figure [1]) we took abstract and concrete initial states that were related by the refinement relation $R$.

We then considered abstract and concrete final states obtained by respectively invoking the abstract and concrete operations and then requiring the final states to be related by $R$.

The Flip operation clearly demonstrates that this is not a valid expectation.

2.6 A revised formalisation of refinement

Consider any final state of the concrete operation, contained in the *blue* set. Since $R$ is, in general, a relation the blue set of concrete states will be related to a the *green* set of abstract states. The informal refinement condition stated earlier requires only that at least one of these states can be reached by the abstract operation invoked in an initial abstract state related by $R$ to the initial concrete state.
The problem with the initial simple formalisation (1) is that it requires all reachable abstract states, the red set, to be reachable from the from the concrete operation. The requirement is that the green set of states should be a subset of the red set.

The story presented by Fig[2] is as follows:

• from a single initial concrete state, •, the operation $O_{PC}$ terminates in any state in blue set of states;
• the refinement relation $R$ maps the initial concrete state to the set of states shown by the orange set of states;
• starting in any state in the orange set of states, the abstract operation $O_{PA}$ terminates in some state in the red set of states;
• the refinement relation $R$ maps the blue set of states to the green set of states

For $O_{PC}$ to be a refinement of $O_{PA}$, the green set must be contained in the red set. This needs to be true for all initial concrete states.

2.7 Isolating the problem

In the substitution $[B_C][B_A](I_A \land I_C \land R)$, the substitution $[B_A](I_A \land I_C \land R)$ is the problem. By definition, $[S](P)$ yields the weakest precondition that guarantees that $P$ will terminate in a state satisfying $P$.

This is too strong, we require only that $B_A$ can terminate in a state satisfying $I_C \land R$. We need something weaker.

Consider $\neg [S] \neg P$:

$[S](\neg P)$ gives the weakest precondition guaranteeing that $S$ will terminate in a state satisfying $\neg P$.
$\neg [S](\neg P)$ gives the weakest precondition guaranteeing that $S$ will not terminate in a state satisfying $\neg P$, that is, it may satisfy $P$.

$\neg [S] \neg P$ is sometimes called the conjugate weakest precondition of $S$ with respect to $P$.

We can now recast[1] as:

$$I_A \land I_C \land R \land P \Rightarrow [B_C] \neg [B_A](\neg (I_C \land R))$$

2.8 Final formulation

Currently we have ignored operation results; we now have to take them into account. Suppose that the abstract operation is

$$result \leftarrow O_{PA} = P \mid B_A$$
and the concrete operation is

\[
result \leftarrow Op_C = P \mid B_C
\]

It is clearly a requirement of refinement that the value of the results of an operation and its refinement must be equal. The results have the same name, so to differentiate we will temporarily rename the result of the concrete operation to \(result'\) and let

\[
B'_C = [result := result']B_C
\]

then the general refinement condition becomes

\[
I_A \land I_C \land R \land P \Rightarrow [B'_C] \neg [B_A](\neg (result' = result \land I_C \land R))
\]

(3)

2.9 Validating Flip under the new formulation

\[coinC = coinA \Rightarrow [coinC \in COIN] \neg [coinA \in COIN] \neg (coinC = coinA)\]

\[= \neg \exists s.(s \land \neg (v := s)(\neg P))\] \hspace{1cm} \text{(distributed through simple substitution)}

Existential quantification captures the arbitrary choice from nondeterminism during refinement.

In contrast, \([S]P = \forall s.([v := s]P)\)

2.10 Notes on the conjugate weakest precondition

If \(S\) is deterministic then \(\neg [S] \neg P = [S]P\).

The behaviour of \(\neg [S] \neg P\) can be demonstrated as follows:

1. Assume that \(S\) has the form \(v \in s\) then

\[
\neg [S] \neg P = \neg [v \in s] \neg P
\]

\[
= \neg \forall \exists x.(xe \land \neg (v := xe)(\neg P))\] \hspace{1cm} \text{(semantics of \(v \in s\))}

\[
= \exists x.(xe \land \neg (v := xe)(\neg P))\] \hspace{1cm} \text{(\(\forall z.(P \Rightarrow Q) = \exists z.(P \land \neg Q)\))}

\[
= \exists x.(xe \land \neg (v := xe)P)\] \hspace{1cm} \text{(\(\neg \) distributes through simple substitution)}

2. Assume that \(S\) has the form \(v := e\) then

\[
\neg [S] \neg P = \neg [v := e] \neg P
\]

\[
= \neg (\neg [v := e]P)
\]

\[
= [v := e]P
\]

\[
= [S]P
\]

2.11 Refinement and feasibility

Refinements form a partial ordering stretching from Abort to Magic, where

\[\text{more generally we could use } S = S_1 \parallel \cdots \parallel S_n\]
1. for all $P$, $[\text{Abort}]P = false$, and
2. for all $P$, $[\text{Magic}]P = true$

Let $\sqsubseteq$ represent refinement then a sequence of refinements $R_i$ is ordered as follows:

$$\text{Abort} \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots \sqsubseteq R_n \sqsubseteq \text{Magic}$$

Thus, $\text{Abort}$ is refined by anything, while $\text{Magic}$ is a refinement of anything. $\text{Abort}$ is easy to implement, while $\text{Magic}$ is impossible to implement. $\text{Magic}$ is infeasible.

2.12 Avoiding the infeasible

The refinement ordering demonstrates that at any refinement step the refinement may, become infeasible. But, it is important to understand that a construct that is feasible can always be refined to a feasible construct. That is, infeasibility can always be avoided — provided, of course, that the original construct was feasible.

2.13 Formality does not guarantee feasibility

To drive home the fact that specifying something using predicates does not preclude infeasibility, here is a specification of an operation that defies Fermat’s last theorem that conjectures,

“there is no integer solutions for $x, y, z$ to the equation $x^n + y^n = z^n$ for integer $n$ with $n > 2$,”

This conjecture was presented in 1637 and not proved until 1995.

**MACHINE** Fermat  
**CONSTANTS** EXP  
**PROPERTIES**  
$\forall \, xx \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \land$  
$\forall \, (xx, nn) \in \mathbb{N} \land \mathbb{N} \land \mathbb{N} \land$  
$\forall \, (xx, 0) = 1$  
$\forall \, (xx, xx) = xx \times \text{EXP}(xx, nn - 1)$

**OPERATIONS**  
$aa, bb, cc \leftarrow \text{Fermat}(nn) \triangleq$  
$\text{pre} \, nn \in \mathbb{N}$  
$\text{then}$  
$\text{any} \, xx, yy, zz$  
$\text{where} \, xx \in \mathbb{N} \land yy \in \mathbb{N} \land zz \in \mathbb{N} \land$  
$\text{EXP}(xx, nn) + \text{EXP}(yy, nn) = \text{EXP}(zz, nn)$

$^2$The ordering, in general, is not linear as shown here, but a lattice.
then \( aa, bb, cc := xx, yy, zz \)
end
END

2.14 Proving feasibility

Feasibility proof obligations can be generated, but generally they are existential proof obligations. The general strategies for discharging existential proof obligations involve producing witnesses, that is giving values that demonstrate that there is at least one solution. This, of course, is equivalent to producing an implementation.

Thus, proof of the feasibility of producing an implementation can involve producing an implementation. This is not a productive solution.

But the situation can be inverted:

if an implementation —with accompanying discharged proof obligations— can be produced then the feasibility proof obligations could have been discharged. Conversely, if the feasibility proof obligations cannot be discharged, then any attempts at implementation will fail.

2.15 The Infeasible cannot be made Feasible

While refinement can produce an infeasible refinement, the converse cannot happen.

Put more strongly: if you start with an infeasible specification, you will not be able to implement it through refinement. This may not be obvious given that infeasibility may be cloaked behind magic at the specification stage.

This can be simply demonstrated.

Consider an operation whose body is represented by the nondeterministic assignment

\[ v_A := \in s \]

Assume that the state invariant is \( I_A \), then the proof obligation will be

\[ I_A \Rightarrow \forall xx. (xx : s \Rightarrow [v_A := xx] I_A) \]

and this is true, if \( s \) is empty, that is the operation is infeasible.

Now suppose that we claim to refine the body of that operation to \( v_C := e \), ie a deterministic refinement, with invariant \( I_B \) and refinement relation \( v_A = v_C \),

then we would have to prove

\[
\begin{align*}
I_A \land I_C \land v_A &= v_C \Rightarrow [v_C := e] \neg [v_A \in s] \neg (I_A \land I_C \land v_A = v_C) \\
&= I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \neg [v_A \in s] \neg (I_A \lor (v_A \neq v_C)) \\
&= I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \neg (\forall xx. (xx : s \Rightarrow \neg I_A \lor (v_A \neq v_C))) \\
&= I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \exists xx. (xx : s \land I_A \land I_C \land xx = v_C) \\
&= I_A \land I_C \land v_A = v_C \Rightarrow \exists xx. (xx : s \land I_A \land I_C \land xx = e) \\
&= false, \text{ if } s \text{ is empty}
\end{align*}
\]

This demonstrates the sting in the tail of magic: it is truly impossible to implement.
Part III

The final refinement step: Implementation

Contents

3 Implementation

Implementation is the final refinement step; an implementation cannot be further refined.
Implementation is a refinement in which the target is B0, a simple procedural programming language, which can be easily translated to C and other programming languages.

There are a number of restrictions on implementations:

- an implementation has no state of its own;
- the implementation imports machines that provides the operations and state that are used to provide the implementation;
- full hiding is enforced, meaning that the state of imported machines can only be accessed through operations;
- parallel composition is not allowed, only sequential composition is allowed;

3.1 A Simple Complete Development

We will present complete developments of simple stateless machines. All refinements are algorithmic refinements.

Consider the following two machines, Square, and SquareRoot containing respectively:

1. Sqr is an operation with a single natural number argument that computes the numerical square of the value of the argument.

2. ApproxSqrt(num) returns the largest natural number that does not exceed $\sqrt{num}$, the real mathematical square root of num.

3.1.1 A square operation

MACHINE Square

OPERATIONS

\[
\text{sqr} \leftarrow \text{Sqr}(\text{num}) \Rightarrow \\
\text{pre num} \in \mathbb{N} \text{ then} \\
\quad \text{sqr} := \text{num} \times \text{num} \\
\text{end}
\]

END
3.1.2 A square root operation

MACHINE SquareRoot

OPERATIONS

\[ \text{sqrt} \leftarrow \text{ApproxSqrt} \left( \text{num} \right) \Leftarrow \]

\[ \text{pre} \; \text{num} \in \mathbb{N} \; \text{then} \]

\[ \text{any} \; \text{approx} \; \text{where} \]

\[ \text{approx} \in \mathbb{N} \; \land \]

\[ \text{approx} \times \text{approx} \leq \text{num} \; \land \]

\[ \text{num} < (\text{approx} + 1) \times (\text{approx} + 1) \]

\[ \text{then} \; \text{sqrt} := \text{approx} \]

\[ \text{end} \]

\[ \text{end} \]

END

3.1.3 The use of non-determinism in specification

The specification of Sqr is constructive, that is, it not only specify what the operation should do, it also describes how the result is computed.

In contrast, the specification of ApproxSqrt is non-constructive; it gives no clue to how the result should be computed.

Rather, the specification provides an acceptance test.

3.1.4 Implementation of Square

Because the specification of Sqr is constructive there is no refinement required for the implementation of that operation.

IMPLEMENTATION SquareI
REFINES Square

OPERATIONS

\[ \text{sqr} \leftarrow \text{Sqr} \left( \text{num} \right) \Leftarrow \]

\[ \text{begin} \]

\[ \text{sqr} := \text{num} \times \text{num} \]

\[ \text{end} \]

END

3.1.5 Implementing ApproxSqrt

The problem with the specification of ApproxSqrt is that we have to choose the value of a single variable that satisfies two conjuncts:
It is relatively easy to choose a value of \emph{approx} to satisfy either conjunct, but not both. Thus we refine the operation by using two variables, \emph{low} and \emph{high} with
\[
\text{low} \times \text{low} \leq \text{num} \\
\text{num} < \text{high} \times \text{high} \\
\text{low} + 1 \leq \text{high}
\]
as shown in the refinement, and implementation machine \texttt{SquareRootI}.
Notice that this machine \texttt{IMPORTS} a new machine \texttt{SqrtUtility}.

\texttt{SquareRootI}

\texttt{IMPLEMENTATION} \texttt{SquareRootI}  \\
\texttt{REFINES} \texttt{SquareRoot}  \\
\texttt{IMPORTS} \texttt{SqrtUtility}

\texttt{OPERATIONS}

\begin{verbatim}
sqrt ←− \text{ApproxSqrt} (\text{num}) \equiv  \\
\text{var} \ \text{low, high in} \  \\
\text{low, high} ←− \text{ChooseInitApprox} (\text{num}) ;  \\
\text{while} \ \text{low} + 1 \neq \text{high} \  \\
\text{do} \ \text{low, high} ←− \text{ChooseBetterApprox} (\text{num}, \text{low}, \text{high})  \\
\text{invariant} \  \\
\text{low} \in \mathbb{N} \land \text{high} \in \mathbb{N} \land \  \\
\text{low} + 1 \leq \text{high} \land \  \\
\text{low} \times \text{low} \leq \text{num} \land \  \\
\text{num} < \text{high} \times \text{high}  \\
\text{variant} \  \\
\text{high} - \text{low}  \\
\text{end} ;  \\
sqrt := \text{low}  \\
\text{end}
\end{verbatim}

\texttt{END}

\texttt{SqrtUtility}

The \texttt{SqrtUtility} machine has operations for choosing an initial approximation for the variables \emph{low} and \emph{high}, and an operation for improving the values of those variables.

\texttt{MACHINE} \texttt{SqrtUtility}

\texttt{OPERATIONS}
\[ \text{low, high} \leftarrow \text{ChooseInitApprox} \ (\text{num}) \ \triangleq \]

\[ \text{pre} \quad \text{num} \in \mathbb{N} \quad \text{then} \]
\[ \qquad \text{any} \quad xx, yy \ \text{where} \]
\[ \qquad \quad xx \in \mathbb{N} \land yy \in \mathbb{N} \land \]
\[ \qquad \quad xx + 1 \leq yy \land \]
\[ \qquad \quad xx \times xx \leq \text{num} \land \]
\[ \qquad \quad \text{num} < yy \times yy \]
\[ \text{then} \quad \text{low, high} := xx, yy \]
\[ \text{end} \]
\[ \text{end} ; \]

\[ \text{low, high} \leftarrow \text{ChooseBetterApprox} \ (\text{num, oldlow, oldhigh}) \ \triangleq \]

\[ \text{pre} \quad \text{num} \in \mathbb{N} \land \text{oldlow} \in \mathbb{N} \land \text{oldhigh} \in \mathbb{N} \land \]
\[ \quad \text{oldlow} \times \text{oldlow} \leq \text{num} \land \text{num} < \text{oldhigh} \times \text{oldhigh} \land \]
\[ \quad \text{oldlow} + 1 < \text{oldhigh} \quad \text{then} \]
\[ \quad \text{any} \quad xx, yy \ \text{where} \]
\[ \quad \quad xx \in \mathbb{N} \land yy \in \mathbb{N} \land \]
\[ \quad \quad xx + 1 \leq yy \land \]
\[ \quad \quad xx \times xx \leq \text{num} \land \]
\[ \quad \quad \text{num} < yy \times yy \land \]
\[ \quad \quad yy - xx < \text{oldhigh} - \text{oldlow} \]
\[ \text{then} \quad \text{low, high} := xx, yy \]
\[ \text{end} \]
\[ \text{end} \]

END

3.1.6 Refining SqrtUtility

Having introduced the machine SqrtUtility we now have to refine it to an implementation.

This is done in three refinement steps as shown in machines SqrtUtilityR, SqrtUtilityRR and SqrtUtilityRRI.

3.1.7 The first refinement SqrtUtilityR

REFINEMENT SqrtUtilityR
REFINES SqrtUtility

OPERATIONS
\[ \text{low, high} \leftarrow \text{ChooseInitApprox} \ (\text{num}) \ \triangleq \]
\[ \text{low, high} := 0, (\text{num} + 1) / 2 + 1 ; \]
\begin{verbatim}
low, high ← ChooseBetterApprox (num, oldlow, oldhigh) ≜
any mid where
mid ∈ N ∧ oldlow < mid ∧ mid < oldhigh then
select mid × mid ≤ num then
low, high := mid, oldhigh
when num < mid × mid then
low, high := oldlow, mid
end
end
END

3.1.8 The second refinement SqrtUtilityRR

REFINEMENT SqrtUtilityRR
REFINES SqrtUtilityR

OPERATIONS
low, high ← ChooseInitApprox (num) ≜
begin low := 0; high := (num + 1) / 2 + 1 end;

low, high ← ChooseBetterApprox (num, oldlow, oldhigh) ≜

var mid in
mid := (oldlow + oldhigh) / 2;
if mid × mid ≤ num
then low := mid; high := oldhigh
else low := oldlow; high := mid
end
end
END

3.1.9 Finally the implementation SqrtUtilityRRI

IMPLEMENTATION SqrtUtilityRRI
REFINES SqrtUtilityRR

OPERATIONS
low, high ← ChooseInitApprox (num) ≜
begin
low := 0; high := (num + 1) / 2 + 1
end;
\end{verbatim}
\begin{align*}
\text{low, high} & \leftarrow \textbf{ChooseBetterApprox}(\ num, \ oldlow, \ oldhigh) \triangleq \\
\text{var} \quad & \text{mid in} \\
& \quad mid := (\ oldlow + \ oldhigh) / 2; \\
& \quad \text{if} \quad mid \leq num / mid \\
& \quad \quad \text{then} \quad low := mid; \quad high := oldhigh \\
& \quad \quad \text{else} \quad low := oldlow; \quad high := mid \\
& \quad \text{end} \\
& \quad \text{end} \\
\text{END} \\
\end{align*}

Notice that the one change in the last refinement step is to replace the guard $mid \times mid \leq num$ by the equivalent $mid \leq num / mid$. The former is logically the condition we want, but it could overflow in a finite integer implementation; the latter is equivalent, but avoids overflow.

### 3.1.10 Generating code

Having produced an implementation machine, \texttt{SqrtUtilityRRI}, we can produce code by selecting the \texttt{trl} button in the \textit{Translators} environment of the B-Toolkit.

The C code is shown in the following frames.

**SquareRoot.c**

```c
#include "SquareRoot.h"
#include "SqrtUtility.h"

void
ApproxSqrt(_sqrt,_num)
int *_sqrt,_num;
{
    int low,high;
    ChooseInitApprox(&low,&high,_num);
    while ( low+1 != high ) {
        ChooseBetterApprox(&low,&high,_num,low,high);
    }
    *_sqrt = low;
}
```

**SqrtUtility.c**

```c
#include "SqrtUtility.h"

void
INI_SqrtUtility()
{
    
} 
```
ChooseInitApprox(_low, _high, _num)
int *_low, *_high, _num;
{
    *_low = 0;
    *_high = (_num+1)/2+1;
}

ChooseBetterApprox(_low, _high, _num, _oldlow, _oldhigh)
int *_low, * _high, _num, _oldlow, _oldhigh;
{
    int mid;
    mid = (_oldlow+_oldhigh)/2;
    if ( mid <= _num/mid ) {
        *_low = mid;
        *_high = _oldhigh;
    }
    else {
        *_low = _oldlow;
        *_high = mid;
    }
}

3.1.11 Generating an interface

To run the code we can introduce an interface to SquareRoot and then generate the interface in the Generators environment of the B-Toolkit.

The interface can be run by selecting the exe (execute) button in the Translators environment.

The interface looks similar to that of the Animator, but this is not an interpretative execution; you are now running actual code.

3.2 A second complete implementation

The second implementation will involve data refinement.

We will specify a simply Queue machine that models a queue manager. A queue, of course, is a first in first out structure. The machine has the following operations:

queueid ← Enqueue(item) an operation that places an item on the end of the queue. The operation returns a token that uniquely identifies this instance of the item in the queue. A queue can contain multiple instances of the same item value.

item ← Dequeue an operation that returns the item that is at the head of the queue.

Unqueue(qid) removes the item from the queue identified by the queue identifier, qid.
3.2.1 Modelling of queue

The queue is modelled by a sequence with the head of the queue being the first element of the sequence; the end of the queue is the last element of the sequence.

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>n</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>items Head               Tail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The Unqueue operation requires unique identification of items in the queue. The position in the queue is continually changing, so we cannot use an item’s initial position. For that reason the elements of the queue will be unique identifiers, `queuetokens`. A function, `queueitem`, maps from `queuetokens` to the actual items.

3.2.2 The Queue machine

```
MACHINE Queue_ctx
SETS QUEUE
END

MACHINE Queue (ITEM, maxqueue)
CONSTRAINTS maxqueue ∈ N_1
SEES Queue_ctx
VARIABLES queuetokens, queue, queueitem
INVARIANT
queuetokens ⊆ QUEUE ∧
queue ∈ iseq (queuetokens) ∧
size (queue) ≤ maxqueue ∧
queuetokens = ran (queue) ∧
queueitem ∈ queuetokens → ITEM
ASSERTIONS card (queuetokens) = size (queue)
INITIALISATION queuetokens, queue, queueitem := {}, {}, {} 

OPERATIONS
Enqueue(item) adds an item to the end of the queue, and returns a unique queue identifier.

queueid ← Enqueue (item) ≜
pre item ∈ ITEM ∧ size (queue) ≠ maxqueue then
any qid where qid ∈ QUEUE − queuetokens then

22
\begin{verbatim}
queuetokens := queuetokens ∪ { qid } ||
queue := queue ← qid ||
queueitem ( qid ) := item ||
queueid := qid
end
end ;

Dequeue returns the item at the head of the queue

\texttt{item ← Dequeue} \iffalse \texttt{Dequeue} \fi
\begin{verbatim}
pre 0 < size ( queue ) then
  any qfirst where qfirst ∈ ran ( queue ) ∧ qfirst = queue ( 1 ) then
    item := queueitem ( qfirst ) || queue := tail ( queue ) ||
    queuetokens := queuetokens − { qfirst } ||
    queueitem := { qfirst } ⊲ queueitem
end
end ;
\end{verbatim}

Unqueue(qid) removes an item from the queue

\texttt{Unqueue ( qid )} \iffalse \texttt{Unqueue} \fi
\begin{verbatim}
pre 0 < size ( queue ) ∧ qid ∈ QUEUE ∧ qid ∈ queuetokens then
  any pos, qhead, qtail where
    pos = queue −1 ( qid ) ∧
    qhead ∈ iseq ( queuetokens − { qid } ) ∧
    qtail ∈ iseq ( queuetokens − { qid } ) ∧
    queue = qhead ⌢ [ qid ] ⌢ qtail
then
  queue := qhead ⌢ qtail ||
  queuetokens := queuetokens − { qid } ||
  queueitem := { qid } ⊲ queueitem
end
end
\end{verbatim}

\textbf{END}

\subsection*{3.2.3 Refining the Queue machine}

The refinement is to replace the monolithic sequence by “pointer” model using the following variables:

- \texttt{qfirst} a pointer to the first element on the queue;
- \texttt{qlast} a pointer to the last element on the queue;
- \texttt{qnext} a pointer that links the elements of the queue —relevant only to queues with more than one item;

\end{verbatim}
Additionally, the refinement has variables \textit{queuetokens} and \textit{queueitem}, which have the same role as they did in the \textit{Queue} machine. Although these variables have the same name they are new variables that are related by equivalence to the variables in the refined machine.

\textit{Refinements may not use variables of the refined machine except in invariants. Complete hiding is enforced.}

\textbf{Relational composition and iteration}

Since we are modelling a list structure by the use of the \textit{qnext} function we will use \textit{relational composition} and transitive \textit{closure} of composition. Suppose we have a list with at least 2 elements, then

\begin{align*}
qfirst & \text{ gives the identity of the first item in the list} \\
qnext(qfirst) & \text{ gives the identity of the second item in the list} \\
(qnext \circ qnext)(qfirst) & \text{ gives the identity of the third item in the list} \\
& \quad \text{ etc}
\end{align*}

Multiple composition is expressed by \textit{iteration}: \textit{qnext}^n (written \textit{iterate}(qnext, n) in ASCII), is the result of composing \textit{qnext} with itself \textit{n} times.

\textit{If} \( r \in X \leftrightarrow X \), \textit{then} \( r^0 = id(X) \) \textit{and} \( r^{n+1} = r^n \circ r \).

\textbf{Closure}

Reflexive transitive closure of a relation \( r \), written \( r^* \), is the union of all iterations of \( r \), that is

\[ r^* = \bigcup n. (n \in \mathbb{N} | r^n)^\dagger \]

Irreflexive transitive closure of a relation, written \( r^+ \), excludes \( r^0 \) from the union

\[ r^+ = \bigcup n. (n \in \mathbb{N}_1 | r^n) \]

Unfortunately, the mathematical toolkit supported by the B-Toolkit does not include \( r^+ \), so the \textit{bigunion} will have to be written wherever it is required.

\textbf{Relation composition of functions}

It should be clear that if \( f \) is a function then \( f \circ f \) is also a function and by extrapolation \( f^n \) is a function.

Further, if \( f \) is an injective function then \( f^n \) is also an injective function.

Thus, \( qnext^n \) is an injective function that gives all paths of length \( n \) within the list.

\( qnext^* \) is a set of injective functions representing all paths, of all lengths from 0 to the length of the list, within the list.

It follows that \( qnext^*[\{qfirst\}] \), the image of the first node in the list under \( qnext^* \), is the set of all nodes in the list.

\footnote{It should be clear that continuous composition of a relation with itself will eventually reach a stationary relation.}
3.2.4  The invariant of QueueR

The list is linear and connected; hence qnext is injective,

\[ qnext \in \text{QUEUE} \rightarrow \text{QUEUE} \]

and there are no loops,

\[ qnext \land \text{id}(\text{QUEUE}) = \{\} \]

We should be able to prove as an assertion

\[ qnext^{+} \land \text{id}(\text{QUEUE}) = \{\} \]

If queuetokens is the set of tokens currently in the list representation of the queue then,

\[ qnext^{*}[\{qfirst\}] = \text{queuetokens} \]

Since each node in the list, except the last, is in the domain of qnext,

\[ \forall nn.(nn : 0..\text{qsize} - 2 \Rightarrow qnext^{nn}(qfirst) \in \text{dom}(qnext)) \]

This is a generalisation of \( qfirst \in \text{dom}(qnext) \)

Since each node in the list, except the first, is in the range of qnext,

\[ \forall nn.(nn : 1..\text{qsize} - 1 \Rightarrow qnext^{nn}(qfirst) \in \text{ran}(qnext)) \]

This is a generalisation of \( qlast \in \text{ran}(qnext) \)

If you take \( \text{qsize} - 1 \) steps along the list for the first node you will reach the last node,

\[ qnext^{\text{qsize} - 1}(qfirst) = qlast \]

for non-empty lists.

3.2.5  The refinement relation of QueueR

the number of items in the linked version is the same as the length of the sequence,

\[ \text{qsize} = \text{size(queue)} \]

the item at position pos in the queue can be recovered by following \( qnext pos - 1 \) steps from the first item,

\[ \forall pos.(pos \in \text{dom}(queue) \Rightarrow qnext^{pos - 1}(qfirst) = \text{queue}(pos)) \]
3.2.6 The QueueR machine

**REFINEMENT** QueueR
**REFINES** Queue
**SEES** Queue_ctx

**VARIABLES**
- queuetokens, queueitem
- qfirst, qlast, qnext, qsize

**INVARIANT**

queuetokens is the set of identifiers for items currently in the queue

\( \text{queuetokens} \subseteq \text{QUEUE} \land \)

queueitem links an item’s token to the item

\( \text{queueitem} \in \text{queuetokens} \rightarrow \text{ITEM} \land \)

\( q\text{first} \in \text{QUEUE} \land q\text{last} \in \text{QUEUE} \land \)

if the queue is not empty then qfirst is one of the current tokens

\( (0 < q\text{size} \Rightarrow q\text{first} \in \text{queuetokens}) \land \)

if the queue is not empty then qlast is one of the current tokens

\( (0 < q\text{size} \Rightarrow q\text{last} \in \text{queuetokens}) \land \)

the list is linear and connected with no loops; hence qnext is injective

\( q\text{next} \in \text{QUEUE} \rightarrow \text{QUEUE} \land \)

the queue has no loops

\( q\text{next} \cap \text{id}(\text{QUEUE}) = \{\} \land \)

\( \bigcup nn . (\text{NN} \land nn \in 0..q\text{size} - 2 \Rightarrow q\text{next}^{nn} (q\text{first}) \in \text{dom}(q\text{next})) \) \land

All of the tokens in the queue are in queuetokens

\( (0 < q\text{size} \Rightarrow \text{queuetokens} = q\text{next}^* [\{q\text{first}\}] ) \land \)

\( q\text{size} \in \text{NN} \land \)

all items in the queue, except the last, have a successor

\( (1 < q\text{size} \Rightarrow \forall nn . (\text{NN} \land nn \in 0..q\text{size} - 2 \Rightarrow q\text{next}^{nn} (q\text{first}) \in \text{dom}(q\text{next})) \) \land

all items in the queue, except the first, have a predecessor

\( (1 < q\text{size} \Rightarrow \forall nn . (\text{NN} \land nn \in 1..q\text{size} - 1 \Rightarrow q\text{next}^{nn} (q\text{first}) \in \text{ran}(q\text{next})) \) \land

the first item in the queue is not the next item of any item

\( q\text{first} \not\in \text{ran}(q\text{next}) \land \)

for a singleton queue the first item is also the last item

\( (q\text{size} = 1 \Rightarrow q\text{last} = q\text{first}) \land \)
the last item does not have a next item

\((0 < qsize \Rightarrow qlast \notin \text{dom}(qnext)) \land \)

taking qsize-1 steps from qfirst reaches qlast

\((0 < qsize \Rightarrow qnext^{qsize-1}(qfirst) = qlast) \land \)

each queue item follows at most one other item

\(qnext \in \text{queue}tokens \mapsto \text{queue}tokens \land \)

\((0 < qsize \Rightarrow \text{card}(qnext) = qsize - 1) \land \)

empty and singleton queues have no next items

\((qsize \leq 1 \Rightarrow qnext = \{\}) \land \)

the number of queue tokens is the same as the size of the queue

\(qsize = \text{card}(\text{queue}tokens) \land \)

composing the qnext function with itself any number of times produces an injective function

\(\forall nn . (nn \in \mathbb{N} \land nn \in 1..qsize-1 \Rightarrow qnext^{nn} \in \text{queue}tokens \mapsto \text{queue}tokens) \land \)

**Refinement relation**

the number of items in the linked version is the same as the length of the sequence

\(qsize = \text{size}(\text{queue}) \land \)

the item at position pos in the queue can be recovered by following qnext pos - 1 steps from the first item

\(\forall pos . (pos \in \text{dom}(\text{queue}) \Rightarrow qnext^{pos-1}(qfirst) = \text{queue}(pos)) \)

**ASSERTIONS**

the range of queue is the union of all the elements in sequence

\(\text{ran}(\text{queue}) = \bigcup pos . (pos \in \text{dom}(\text{queue}) \mid \{\text{queue}(pos)\}) \land \)

\((1 < qsize \Rightarrow \text{queue}tokens = \text{ran}(\text{queue})) \land \)

if the queue has more than 1 item then the first item has a successor

\((1 < qsize \Rightarrow qfirst \in \text{dom}(qnext)) \land \)

if the queue has more than 1 item then the last item has a predecessor

\((1 < qsize \Rightarrow qlast \in \text{ran}(qnext)) \land \)

Specialisation of the refinement relation

\((0 < qsize \Rightarrow qfirst = \text{first}(\text{queue})) \land \)

\((0 < qsize \Rightarrow qlast = \text{last}(\text{queue})) \land \)

**INITIALISATION**

\(\text{queue}tokens, \text{queue}item := \{\}, \{\} \parallel \)

\(qsize, qnext := 0, \{\} \parallel qfirst : \text{QUEUE} \parallel qlast : \text{QUEUE} \)

27
Note that $qfirst$ and $qlast$ always have a value, even when the list is empty.

OPERATIONS

$$\text{queueid} \leftarrow \text{Enqueue} (\text{item}) \triangleq$$

\[
\text{any } qid \text{ where } qid \in \text{QUEUE} - \text{queuetokens} \text{ then}
\]

\[
\text{queuetokens} := \text{queuetokens} \cup \{ \text{qid} \} \parallel
\]

\[
\text{qlast} := qid \parallel
\]

\[
\text{if } 0 < \text{qsize} \text{ then}
\]

\[
\text{qnext} (\text{qlast}) := qid
\]

\[
\text{else } qfirst := qid
\]

\[
\text{end} \parallel
\]

\[
\text{queueitem} (\text{qid}) := \text{item} \parallel \text{qsize} := \text{qsize} + 1 \parallel
\]

\[
\text{queueid} := \text{qid}
\]

\[
\text{end} ;
\]

\[
\text{item} \leftarrow \text{Dequeue} \triangleq
\]

\[
\text{begin}
\]

\[
\text{item} := \text{queueitem} (\text{qfirst}) \parallel
\]

\[
\text{if } 1 < \text{qsize} \text{ then}
\]

\[
\text{qfirst} := \text{qnext} (\text{qfirst}) \parallel \text{qnext} := \{ \text{qfirst} \} \leftarrow \text{qnext}
\]

\[
\text{end} \parallel
\]

\[
\text{queueitem} := \{ \text{qfirst} \} \leftarrow \text{queueitem} \parallel
\]

\[
\text{queuetokens} := \text{queuetokens} - \{ \text{qid} \} \parallel
\]

\[
\text{qsize} := \text{qsize} - 1
\]

\[
\text{end} ;
\]

\[
\text{Unqueue} (\text{qid}) \triangleq
\]

\[
\text{begin}
\]

\[
\text{if } 1 < \text{qsize} \text{ then}
\]

\[
\text{if } \text{qid} = \text{qfirst} \text{ then}
\]

\[
\text{qfirst} := \text{qnext} (\text{qid}) \parallel \text{qnext} := \{ \text{qid} \} \leftarrow \text{qnext}
\]

\[
\text{elsif } \text{qid} = \text{qlast} \text{ then}
\]

\[
\text{qlast} := \text{qprev} (\text{qid}) \parallel \text{qnext} := \text{qnext} \Rightarrow \{ \text{qid} \}
\]

\[
\text{else}
\]

\[
\text{qnext} := \{ \text{qid} \} \leftarrow (\text{qnext} \Leftarrow \{ \text{qprev} (\text{qid}) \mapsto \text{qnext} (\text{qid}) \})
\]

\[
\text{end}
\]

\[
\text{end} \parallel
\]

\[
\text{qsize} := \text{qsize} - 1 \parallel
\]

\[
\text{queuetokens} := \text{queuetokens} - \{ \text{qid} \} \parallel
\]
queueitem := \{ qid \} \leq queueitem
end
DEFINITIONS qprev \equiv qnext^{-1}
END

3.2.7 Implementation of Queue

The Design of the Implementation

Implementing queuetokens

A common tactic for implementing a set of unique identifiers is to simply use a counter, which is advanced each time a new token is required. This only works when tokens are not reclaimed. In the case of the queue tokens are taken out of use by DeQueue and UnQueue. We could simply lose them: this would be correct, but means we will run out of tokens more quickly; or we could reclaim them. We choose to do the latter. Our implementation uses a counter augmented by a set of allocated but unused tokens.

Implementing qnext

We implement qnext, which remember is a partial injection, by using a partial function machine qnext_Nfnc with the refinement relation: qnext \subseteq qnext_Nfnc. This is closer to a real pointer implementation; the important property is that within the domain of qnext the function is injective. Notice that in the implementation of DeQueue and UnQueue, qnext_Nfnc is left defined for the deleted item, but that item is no longer accessible from the head of the queue.

Implementing prev

Until now we got prev for free because qnext was an injective function, so prev by simply inverting qnext. In the implementation we have no such luxury. Here we implement prev by using a loop to search from the beginning of the queue (list). This, of course, is inefficient. If efficiency is important, we could implement a doubly linked list, ie implement qprev.

IMPLEMENTATION QueueRI
REFINES QueueR
SEES Bool_TYPE, Queue_ctx
IMPORTS
queuetokens_Nvar (maxqueue),
queueitem_Vfnc (ITEM, maxqueue),
qnext_Nfnc (maxqueue, maxqueue),
freetokens_set (QUEUE, maxqueue),
qfirst_Nvar (maxqueue),
qlast_Nvar (maxqueue),
qsize_Nvar (maxqueue)

PROPERTIES QUEUE = 0..maxqueue
INVARIANT
Now the refinement relation

\[
\text{queuetokens} \subseteq 1 \ldots \text{maxqueue} \land \\
\text{queuetokens} \cup \text{freetokens}\_\text{sset} = 1 \ldots \text{queuetokens}\_\text{Nvar} \land \\
\text{queuetokens} \cap \text{freetokens}\_\text{sset} = \emptyset \land \\
\text{queueitem}\_\text{Vfnc} = \text{queueitem} \land \\
\text{qnext} \subseteq \text{qnext}\_\text{Nfnc} \land \\
(0 < \text{qsize}\_\text{Nvar} \Rightarrow \text{qfirst}\_\text{Nvar} = \text{qfirst}) \land \\
(0 < \text{qsize}\_\text{Nvar} \Rightarrow \text{qlast}\_\text{Nvar} = \text{qlast}) \land \\
\text{queuetokens}\_\text{Nvar} - \text{card( freetokens}\_\text{sset}) = \text{qsize} \land \\
\text{qsize}\_\text{Nvar} = \text{qsize}
\]

**Assertions**

\[
\text{freetokens}\_\text{sset} \subseteq 1 \ldots \text{queuetokens}\_\text{Nvar} \land \\
\text{queuetokens}\_\text{Nvar} - \text{card( freetokens}\_\text{sset}) = \text{card( queuetokens)} \land \\
(\text{freetokens}\_\text{sset} = \emptyset \Rightarrow \text{queuetokens} = 1 \ldots \text{queuetokens}\_\text{Nvar})
\]

**Operations**

\[
\text{queueid} \leftarrow \text{Enqueue( item )} \equiv \\
\text{var bb, qid, lst in} \\
\text{bb} \leftarrow \text{freetokens}\_\text{EMP}\_\text{SET}; \\
\text{if bb = TRUE then} \\
\text{queuetokens}\_\text{INC}\_\text{NVAR}; \text{qid} \leftarrow \text{queuetokens}\_\text{VAL}\_\text{NVAR} \\
\text{else} \\
\text{qid} \leftarrow \text{freetokens}\_\text{ANY}\_\text{SET}; \text{freetokens}\_\text{RMV}\_\text{SET}( \text{qid} ) \\
\text{end}; \\
\text{bb} \leftarrow \text{qsize}\_\text{GTR}\_\text{NVAR}(0); \\
\text{if bb = TRUE then} \\
\text{lst} \leftarrow \text{qlast}\_\text{VAL}\_\text{NVAR}; \text{qnext}\_\text{STO}\_\text{NFNC}( \text{lst}, \text{qid} ) \\
\text{else} \\
\text{qfirst}\_\text{STO}\_\text{NVAR}( \text{qid} ) \\
\text{end}; \\
\text{qlast}\_\text{STO}\_\text{NVAR}( \text{qid} ); \text{queueitem}\_\text{STO}\_\text{FNC}( \text{qid}, \text{item} ); \\
\text{qsize}\_\text{INC}\_\text{NVAR}; \text{queueid} := \text{qid} \\
\text{end}; \\
\]

\[
\text{item} \leftarrow \text{Dequeue} \equiv \\
\text{var bb, qfst, qxnt in} \\
\text{qfst} \leftarrow \text{qfirst}\_\text{VAL}\_\text{NVAR}; \text{item} \leftarrow \text{queueitem}\_\text{VAL}\_\text{FNC}( \text{qfst} ); \\
\text{bb} \leftarrow \text{qsize}\_\text{GTR}\_\text{NVAR}(1); \\
\text{if bb = TRUE then}
\]

30
Unqueue \( (\textit{qid}) \) \( \triangleq \)

\[
\begin{align*}
\text{var } & \text{ } bb, qnxt, qprv \text{ in} \\
& bb \leftarrow qsize \text{}_\rangle \text{NVAR} (1) \\
& \text{if } bb = \text{TRUE} \text{ then} \\
& \quad bb \leftarrow qfirst \text{}_= \text{NVAR} (\textit{qid}) \\
& \text{if } bb = \text{TRUE} \text{ then} \\
& \quad qnxt \leftarrow qnext \text{VAL} \text{NFNC} (\textit{qid}) ; qfirst \text{STO} \text{NVAR} (qnxt) \\
& \text{else} \\
\end{align*}
\]

We now need the predecessor of the item to be deleted. We will find this by searching from the beginning of the queue until the next item is the item being deleted.

\[
\begin{align*}
\text{aprv} & \leftarrow qfirst \text{VAL} \text{NVAR} ; qnxt \leftarrow qnext \text{VAL} \text{NFNC} (\textit{aprv}) \\
\text{while } & qnxt \neq \textit{qid} \text{ do} \\
& \text{aprv} := qnxt ; qnxt \leftarrow qnext \text{VAL} \text{NFNC} (\textit{aprv}) \\
\text{invariant} & \\
& \text{queueitem} \text{Vfnc} = \text{queueitem} \land \\
& \text{qnext} \text{Nfnc} = \text{qnext} \land \text{qfirst} \text{Nvar} = \text{qfirst} \land \\
& \text{qlast} \text{Nvar} = \text{qlast} \land \text{qsize} \text{Nvar} = \text{qsize} \land \\
& \text{aprv} \in \text{dom}(\text{qnext}) \land qnxt = qnext(\textit{aprv}) \\
\text{variant} & \text{queue}^{-1}(\textit{qid}) = \text{queue}^{-1}(\text{qnxt}) \\
\text{end} \\
\end{align*}
\]

The consequence of the loop is the negation of the guard conjuncted with the invariant, ie

\[
\neg (qnxt \neq \textit{qid}) \land qnxt = qnext(\textit{aprv}) \ldots \equiv qnext(qnxt) = \textit{qid} \ldots
\]

\[
\begin{align*}
bb & \leftarrow qlast \text{EQL} \text{NVAR} (\textit{qid}) \\
\text{if } & bb = \text{TRUE} \text{ then} \\
& qlast \text{STO} \text{NVAR} (\textit{qprv}) ; qnext \text{RMV} \text{NFNC} (\textit{qprv}) \\
\text{else} & \\
& qnxt \leftarrow qnext \text{VAL} \text{NFNC} (\textit{qid}) ; \\
& qnext \text{STO} \text{NFNC} (\textit{qprv}, qnxt) \\
\text{end} \\
\text{end} \\
\text{end} \\
\text{end} \\
\text{end} \\
\text{end} \\
\text{qsize} \text{DEC} \text{NVAR} ;
\]
3.2.8 Notes on While Invariant and Variant

The main conjuncts of the invariant are:

\[ q_{prv} \in \text{dom}(q_{next}) \land q_{nxt} = q_{next}(q_{prv}) \]

which describes the relationship between the changing variables.

\[ \text{queueitem}_{Vfnc} = \text{queueitem} \land \]

The other parts of the invariant:

\[ q_{next}_{Nfnc} = q_{next} \land q_{first}_{Nvar} = q_{first} \land \]
\[ q_{last}_{Nvar} = q_{last} \land q_{size}_{Nvar} = q_{size} \]

describe refinement relations between variables that do not change within the loop.

The variant is interesting because at first sight there doesn’t seem to be anything obvious that is decreasing in traversing a list structure.

It is important to realise all variables are visible within abstract expressions like the invariant and the variant. In particular, traversing the list version of the queue is exactly modelled by the corresponding traversing of the sequence in the specification. This means that we can use \( q_{queue}^{-1} \) to map queuetokens to position in the sequence and

\[ q_{queue}^{-1}(qid) - q_{queue}^{-1}(q_{nxt}) \]

is a decreasing expression bounded below by 0.

3.3 Adding an interface to Queue.mch

An API machine, whose operations are all robust, will now be defined.

The parametric set \( ITEM \) for \( Queue \) will be instantiated by a set a context machine. This is particularly necessary as \( QueueAPI \) will have to choose an arbitrary element from \( ITEM \). This is not possible from a parametric set.

Response_ctx and Item_ctx

MACHINE Response_ctx
SETS
\[ RESPONSE = \]
\{ OK , QueueFull , QueueEmpty , NotInQueue \} END
MACHINE Item_ctx
SETS ITEM
END

QueueAPI.mch

MACHINE QueueAPI ( maxqueue )
CONSTRAINTS maxqueue ∈ \mathbb{N}_1
SEES Response_ctx, Queue_ctx, Item_ctx
INCLUDES Queue ( ITEM, maxqueue )

OPERATIONS

status, queueid ← EnqueueAPI ( item ) ≜
  \text{pre } \text{ item } \in \text{ITEM} \text{ then}
  \text{select } \text{size ( queue )} \neq \text{maxqueue} \text{ then}
  \text{status} := \text{OK} \parallel \text{queueid} ← \text{Enqueue ( item )}
  \text{else } \text{status} := \text{QueueFull} \parallel \text{queueid} : \in \text{QUEUE}
  \text{end}
  \text{end}

\text{status, item} ← \text{DequeueAPI } ≜
  \text{select } 1 \leq \text{size ( queue )} \text{ then}
  \text{status} := \text{OK} \parallel \text{item} ← \text{Dequeue}
  \text{else } \text{status} := \text{QueueEmpty} \parallel \text{item} : \in \text{ITEM}
  \text{end}
\text{end}

\text{status} ← \text{UnqueueAPI ( qid ) } ≜
  \text{pre } \text{qid} \in \text{QUEUE} \text{ then}
  \text{select } 0 = \text{size ( queue )} \text{ then}
  \text{status} := \text{QueueEmpty}
  \text{when } \text{qid} \notin \text{queuetokens} \text{ then}
  \text{status} := \text{NotInQueue}
  \text{else } \text{status} := \text{OK} \parallel \text{Unqueue ( qid )}
  \text{end}
  \text{end}
END
3.3.1 Implementation strategies

There are principally two implementation strategies:

1. Simply implement the whole machine, in this case QueueAPI.
2. Use an import structure that mimics the inclusion structure of the specification combined with separate implementation of the imported machines. In this case this would mean implementing QueueAPI by essentially importing the Queue machine; followed by implementation of Queue.

However, due to complete hiding, it will not be possible to evaluate the guards. To facilitate this a machine QueuePreAPI, which augments the Queue operations, by auxiliary operations that will be required by the implementation of QueueAPI.

3.3.2 QueuePreAPI.mch

QueuePreAPI.mch

MACHINE QueuePreAPI ( maxqueue )
CONSTRAINTS maxqueue ∈ \mathbb{N}_1
SEES Bool_TYPE , Queue_ctx , Item_ctx
EXTENDS Queue ( ITEM , maxqueue )

OPERATIONS

\begin{align*}
\text{anyqueue} & \leftarrow \text{anyQUEUE} \triangleq \text{anyqueue} :\subseteq \text{QUEUE} ; \\
\text{anyitem} & \leftarrow \text{anyITEM} \triangleq \text{anyitem} :\subseteq \text{ITEM} ; \\
\text{ok} & \leftarrow \text{isQueueEmpty} \triangleq \\
& \quad \text{ok} := \text{bool} ( \text{size} ( \text{queue} ) = 0 ) ; \\
\text{ok} & \leftarrow \text{isQueueFull} \triangleq \\
& \quad \text{ok} := \text{bool} ( \text{size} ( \text{queue} ) = \text{maxqueue} ) ; \\
\text{ok} & \leftarrow \text{isItemInQueue} ( \text{qid} ) \triangleq \\
& \quad \text{pre } qid \in \text{QUEUE} \text{ then} \\
& \quad \quad \text{ok} := \text{bool} ( \text{qid} \in \text{queuetokens} ) \\
\end{align*}

END

3.3.3 Implementing QueueAPI

Given QueuePreAPI we can give the implementation of QueueAPI, QueueAPII.

The purpose of the auxiliary operations will be clear from an inspection of the code for the guards of the robust operations.

IMPLEMENTATION QueueAPII
REFINES QueueAPI
SEES  Bool_TYPE, Response_ctxt, Queue_ctxt, Item_ctxt
IMPORTS QueuePreAPI (maxqueue)

OPERATIONS

\[
\begin{align*}
\text{status, queueid} &\leftarrow \text{EnqueueAPI (item)} \triangleq \\
\text{var } bb \text{ in } \\
& bb \leftarrow \text{isQueueFull} ; \\
& \text{if } bb = \text{FALSE} \text{ then} \\
& \text{status} := \text{OK} ; \text{queueid} \leftarrow \text{Enqueue (item)} \\
& \text{else} \\
& \text{status} := \text{QueueFull} ; \text{queueid} \leftarrow \text{anyQUEUE} \\
& \text{end} \\
& \text{end} ; \\
\end{align*}
\]

\[
\begin{align*}
\text{status, item} &\leftarrow \text{DequeueAPI} \triangleq \\
\text{var } bb \text{ in } \\
& bb \leftarrow \text{isQueueEmpty} ; \\
& \text{if } bb = \text{FALSE} \text{ then} \\
& \text{status} := \text{OK} ; \text{item} \leftarrow \text{Dequeue} \\
& \text{else} \\
& \text{status} := \text{QueueEmpty} ; \text{item} \leftarrow \text{anyITEM} \\
& \text{end} \\
& \text{end} ; \\
\end{align*}
\]

\[
\begin{align*}
\text{status} &\leftarrow \text{UnqueueAPI (qid)} \triangleq \\
\text{var } bb \text{ in } \\
& bb \leftarrow \text{isQueueEmpty} ; \\
& \text{status} := \text{QueueEmpty} ; \\
& \text{if } bb = \text{FALSE} \text{ then} \\
& \text{status} := \text{NotInQueue} ; \\
& bb \leftarrow \text{isItemInQueue (qid)} ; \\
& \text{if } bb = \text{TRUE} \text{ then} \\
& \text{status} := \text{OK} ; \text{Unqueue (qid)} \\
& \text{end} \\
& \text{end} \\
& \text{end} \\
& \text{END}
\end{align*}
\]
3.3.4 Refining QueuePreAPI

QueuePreAPI now needs to be implemented and we start by refining as we did for the implementation of Queue.

QueuePreAPI follows QueueR, with the addition of the auxiliary operations.

In turn, the refinement will be implemented following QueueRI

Since QueuePreAPI and QueuePreAPIR are so similar to QueueR and QueueRI they are moved to the appendix.
A Appendix

A.1 QueuePreAPIR

QueuePreAPIR

REFINEMENT QueuePreAPIR
REFINES QueuePreAPI
SEES Bool_TYPE, Queue_ctx, Item_ctx

VARIABLES
queuetokens, queueitem,
afirst, qlast, qnext, qsize

INVARIANT

quequetokens is the set of identifiers for items currently in the queue
queuetokens ⊆ QUEUE ∧ queuetokens = ran (queue ) ∧

queueitem links an item’s token to the item
queueitem ∈ queuetokens → ITEM ∧
afirst ∈ QUEUE ∧ qlast ∈ QUEUE ∧

if the queue length is greater than 1 then qfirst is one of the current tokens
(0 < qsize ⇒ qfirst ∈ queuetokens) ∧

if the queue length is greater than 1 then qlast is one of the current tokens
(0 < qsize ⇒ qlast ∈ queuetokens) ∧

the list is linear and connected with no loops; hence qnext is injective
qnext ∈ QUEUE ↾ Q U E U E ∧

there are no loops
qnext ∩ id (QUEUE ) = {} ∧

∪ nn . (nn ∈ N ∧ qnextmn ) ∩ id (QUEUE ) = {} ∧
qsize ∈ N ∧

All of the tokens in the queue are in queuetokens
(1 < qsize ⇒ queuetokens = qnextn [ {qfirst } ] ) ∧

all items in the queue, except the last, have a successor
(1 < qsize ⇒ ∀ nn . (nn ∈ N ∧ nn ∈ 0..qsize − 2 ⇒
qnextnn (a first) ∈ dom (qnext)) ) ∧

all items in the queue, except the first, have a predecessor
(1 < qsize ⇒ ∀ nn . (nn ∈ N ∧ nn ∈ 1..qsize − 1 ⇒
qnextnn (qfirst) ∈ ran (qnext)) ) ∧

the first item in the queue is not the next item of any item
a first ∉ ran (qnext) ∧

for a singleton queue the first item is also the last item
(qsize = 1 ⇒ qlast = qfirst) ∧

the last item does not have a next item

(0 < qsize ⇒ qlast ∉ dom (qnext)) ∧

taking qsize-1 steps from qfirst reaches qlast

(0 < qsize ⇒ qnextqsize − 1 (qfirst) = qlast) ∧

each queue item follows at most one other item

qnext ∈ queuetokens ↦ queuetokens ∧

(0 < qsize ⇒ card (qnext) = qsize − 1) ∧

empty and singleton queues have no next items

(qsize ≤ 1 ⇒ qnext = { }) ∧

the number of queue tokens is the same as the size of the queue

qsize = card (queuetokens) ∧

composing the qnext function with itself any number of times produces an injective function

∀ nn . (nn ∈ N ∧ nn ∈ 1 . . . qsize − 1 ⇒ qnext^nn ∈ queuetokens ↦ queuetokens) ∧

Refinement relation

the number of items in the linked version is the same as the length of the sequence

qsize = size (queue) ∧

the item at position pos in the queue can be recovered by following qnext pos − 1 steps from the first item

∀ pos . (pos ∈ dom (queue) ⇒ qnext^pos − 1 (qfirst) = queue (pos))

ASSERTIONS

if the queue has more than 1 item then the first item has a successor

(1 < qsize ⇒ qfirst ∈ dom (qnext)) ∧

if the queue has more than 1 item then the last item has a predecessor

(1 < qsize ⇒ qlast ∈ ran (qnext)) ∧

Specialisation of the refinement relation

(0 < qsize ⇒ qfirst = first (queue)) ∧

(0 < qsize ⇒ qlast = last (queue))

INITIALISATION

queuetokens, queueitem := {}, {} ||

qsize, qnext := 0, {} || qfirst ∈ QUEUE || qlast ∈ QUEUE

Note that qfirst and qlast always have a value, even when the list is empty.

38
OPERATIONS

queueid ← Enqueue (item) ⇐

any qid where qid ∈ QUEUE − queuetokens then
queuetokens := queuetokens ∪ {qid} ∥
qlast := qid ∥
if 0 < qsize then
  qnext (qlast) := qid
else qfirst := qid
end ∥
queueitem (qid) := item ∥ qsize := qsize + 1 ∥
queueid := qid
end ;

item ← Dequeue ⇐

begin
  item := queueitem (qfirst) ∥
  if 1 < qsize then
    qfirst := qnext (qfirst) ∥ qnext := {qfirst} ← qnext
  end ∥
  queueitem := {qfirst} ← queueitem ∥
  queuetokens := queuetokens − {qfirst} ∥
  qsize := qsize − 1
end ;

Unqueue (qid) ⇐

begin
  if 1 < qsize then
    if qid = qfirst then
      qfirst := qnext (qid) ∥ qnext := {qid} ← qnext
    elseif qid = qlast then
      qlast := qprev (qid) ∥ qnext := qnext ▷ {qid}
    else
      qnext := {qid} ▷ (qnext ▷ {qprev (qid)} ▷ qnext (qid))
    end
  end ∥
  qsize := qsize − 1 ∥
  queuetokens := queuetokens − {qid} ∥
  queueitem := {qid} ← queueitem
end ;

anyqueue ← anyQUEUE ⇐ anyqueue ∈ QUEUE ;
anyitem ← anyITEM ⇐ anyitem ∈ ITEM ;
ok ← isQueueEmpty ⇐ ok := bool (qsize = 0) ;
ok ← isQueueFull ⇐ ok := bool (qsize = maxqueue) ;
ok ← isItemInQueue (qid) ⇐ ok := bool (qid ∈ queuetokens)
DEFINITIONS \( q_{prev} \equiv q_{next}^{-1} \)
END
A.2 QueuePreAPIR

QueuePreAPIR

IMPLEMENTATION QueuePreAPIR
REFINES QueuePreAPIR
SEES Bool_TYPE, Queue_ctx, Item_ctx
IMPORTS
  queuetokens_Nvar (maxqueue),
  queueitem_Vfunc (ITEM, maxqueue),
  qnext_Nfunc (maxqueue, maxqueue),
  freetokens_set (QUEUE, maxqueue),
  qfirst_Nvar (maxqueue),
  qlast_Nvar (maxqueue),
  qsize_Nvar (maxqueue)
PROPERTIES
  QUEUE = 0 .. maxqueue ∧
  ITEM = 0 .. 100

INVARIANT

Now the refinement relation

  queuetokens ⊆ 1 .. maxqueue ∧
  queuetokens ∪ freetokens_sset = 1 .. queuetokens_Nvar ∧
  queuetokens ∩ freetokens_sset = {} ∧
  queueitem_Vfunc = queueitem ∧
  qnext ⊆ qnext_Nfunc ∧
  ( 0 < qsize_Nvar ⇒ qfirst_Nvar = qfirst ) ∧
  ( 0 < qsize_Nvar ⇒ qlast_Nvar = qlast ) ∧
  queuetokens_Nvar − card (freetokens_sset) = qsize ∧
  qsize_Nvar = qsize

ASSERTIONS

  freetokens_sset ⊆ 1 .. queuetokens_Nvar ∧
  queuetokens_Nvar − card (freetokens_sset) = card (queuetokens) ∧
  (freetokens_sset = {} ⇒ queuetokens = 1 .. queuetokens_Nvar) ∧
  (freetokens_sset = {} ⇒ queuetokens_Nvar = qsize)

OPERATIONS

  queueid ← Enqueue (item) ≜
  var bb, qid, lst in
  bb ← freetokens_EMP_SET ;
  if bb = TRUE then
    queuetokens_INC_NVAR ; qid ← queuetokens_VAL_NVAR
  else
qid ← freetokens\_ANY\_SET ; freetokens\_RMV\_SET ( qid )
end ;

bb ← qsize\_GTR\_NVAR ( 0 ) ;
if \( bb = \text{TRUE} \) then
    lst ← qlast\_VAL\_NVAR ; qnext\_STO\_NFNC ( lst , qid )
else
    qfirst\_STO\_NVAR ( qid )
end ;
qlast\_STO\_NVAR ( qid ) ; queueitem\_STO\_FNC ( qid , item ) ;
qsize\_INC\_NVAR ; queueid := qid
end ;

item ← Dequeue \( \equiv \)
var \( bb , qfst , qnxt \) in
qfst ← qfirst\_VAL\_NVAR ; item ← queueitem\_VAL\_FNC ( qfst ) ;
bb ← qsize\_GTR\_NVAR ( 1 ) ;
if \( bb = \text{TRUE} \) then
    qnxt ← qnext\_VAL\_NFNC ( qfst ) ;
    qfirst\_STO\_NVAR ( qnxt )
end ;
freetokens\_ENT\_SET ( qfst ) ; qsize\_DEC\_NVAR
end ;

Unqueue ( qid ) \( \equiv \)
var \( bb , qnxt , qprv \) in
bn ← qsize\_GTR\_NVAR ( 1 ) ;
if \( bb = \text{TRUE} \) then
    bn ← qfirst\_EQ\_NVAR ( qid ) ;
    if \( bb = \text{TRUE} \) then
        qnxt ← qnext\_VAL\_NFNC ( qid ) ; qfirst\_STO\_NVAR ( qnxt )
    else
        freeitems ( qid )
    end ;
end ;

We now need the predecessor of the item to be deleted. We will find this by searching from the beginning of
the queue until the next item is the item being deleted.
qprv ← qfirst\_VAL\_NVAR ; qnxt ← qnext\_VAL\_NFNC ( qprv ) ;
while \( qnxt \neq \text{id} \) do
    qprv := qnxt ; qnxt ← qnext\_VAL\_NFNC ( qprv )
end ;

invariant
queueitem\_Vfnc = queueitem \∧
qnext\_Nfnc = qnxt \∧ qfirst\_Nvar = qfirst \∧
qlast\_Nvar = qlast \∧ qsize\_Nvar = qsize \∧
qprv \in \text{dom} ( qnxt ) \∧ qnxt = qnext ( qprv )

variant \( qsize = \text{queue}^-1 ( qid ) \)
end ;
The consequence of the loop is the negation of the guard conjuncted with the invariant, ie
\( \neg ( qnxt \neq qid ) \land qnxt = qnext ( qprv ) \ldots \equiv qnext ( qnxt ) = qid \ldots \)
bb ← qlast_EQL_NVAR ( qid ) ;  
if bb = TRUE then  
qlast_STO_NVAR ( qprv ) ; qnext_RM_VNFNC ( qprv )  
else  
qnext ← qnext_VAL_NFNC ( qid ) ;  
qnext_STO_NFNC ( qprv , qnext )  
end  
end ;  
qsize_DEC_NVAR ;  
freetokens_ENT_SET ( qid ) ;  
queueitem_RM_FNC ( qid )  
end ;

anyqueue ← anyQUEUE ≜ anyqueue := 0 ;  
anyitem ← anyITEM ≜ anyitem := 0 ;  
ok ← isQueueEmpty ≜ ok ← qsize_EQL_NVAR ( 0 ) ;  
ok ← isQueueFull ≜ ok ← qsize_GEQ_NVAR ( maxqueue ) ;  
ok ← isItemInQueue ( qid ) ≜  
var bb in  
ok := FALSE ;  
if qid ≠ 0 then  
bb ← queuetokens_GEQ_NVAR ( qid ) ;  
if bb = TRUE then  
bb ← freetokens_MBR_SET ( qid ) ;  
if bb = FALSE then  
ok := TRUE  
end  
end  
end  
END
B  Imported system library machines

The following renameable machines are imported into the Queue implementations:

- **queuetokens\_Nvar** a natural number variable machine that we will use as a partial implementation of *queuetokens*. The variable provides a counter used to produce the next *unallocated* queue token.

- **queueitems\_Vfnc** a general type function machine used to implement the *queueitems* function.

- **freetokens\_set** a set machine used to save tokens that have been *returned*.

- **qnext\_Nfnc** a Natural number function machine that we use for the implementation of *qnext*.

- **qfirst\_Nvar** implements *qfirst*; **qlast\_Nvar** implements *qlast*; **qsize\_Nvar** implements *qsize*. These machines are the same as **queuetokens\_Nvar**, except for the rename prefixes.
B.1 queuetokens_Nvar.mch

MACHINE queuetokens_Nvar ( maxint )
CONSTRAINTS maxint \leq 2147483646
SEES Bool_TYPE
VARIABLES queuetokens_Nvar
INVARIANT queuetokens_Nvar \in 0 .. maxint
INITIALISATION queuetokens_Nvar := 0

OPERATIONS
vv ← queuetokens_VAL_NVAR ≜
begin
vv := queuetokens_Nvar
end ;
queuetokens_STO_NVAR ( vv ) ≜
pre
vv \in 0 .. maxint
then
queuetokens_Nvar := vv
end ;
uu ← queuetokens_MIN_NVAR ( vv ) ≜
pre
vv \in 0 .. maxint
then
uu := \min ( \{ queuetokens_Nvar , vv \} )
end ;
uu ← queuetokens_MAX_NVAR ( vv ) ≜
pre
vv \in 0 .. maxint
then
uu := \max ( \{ queuetokens_Nvar , vv \} )
end ;
bb ← queuetokens_PRE_INC_NVAR ≜
begin
bb := bool ( queuetokens_Nvar < maxint )
end :
queuetokens_INC_NVAR ≜
pre
queuetokens_Nvar + 1 \in 0 .. maxint
then
queuetokens_Nvar := queuetokens_Nvar + 1
end ;
bb ← queuetokens_PRE_DEC_NVAR ≜
begin
    \( \text{bb} := \text{bool}\left( \text{queuetokens}_{N\text{var}} > 0 \right) \)
end ;
queuetokens_{DEC\_N\text{var}} \triangleq 
pre
    \text{queuetokens}_{N\text{var}} \in 1 .. \text{maxint}
then
    \text{queuetokens}_{N\text{var}} := \text{queuetokens}_{N\text{var}} - 1
end ;
\text{bb} \leftarrow \text{queuetokens\_PRE\_ADD\_N\text{var}}\left( vv \right) \triangleq 
pre
    vv \in 0 .. \text{maxint}
then
    \text{bb} := \text{bool}\left( \text{queuetokens}_{N\text{var}} + vv \leq \text{maxint} \right)
end ;
queuetokens\_ADD\_N\text{var}\left( vv \right) \triangleq 
pre
    vv \in 0 .. \text{maxint} \land 
    \text{queuetokens}_{N\text{var}} + vv \leq \text{maxint}
then
    \text{queuetokens}_{N\text{var}} := \text{queuetokens}_{N\text{var}} + vv
end ;
\text{bb} \leftarrow \text{queuetokens\_PRE\_MUL\_N\text{var}}\left( vv \right) \triangleq 
pre
    vv \in 0 .. \text{maxint}
then
    \text{bb} := \text{bool}\left( \text{queuetokens}_{N\text{var}} \times vv \leq \text{maxint} \right)
end ;
queuetokens\_MUL\_N\text{var}\left( vv \right) \triangleq 
pre
    vv \in 0 .. \text{maxint} \land 
    \text{queuetokens}_{N\text{var}} \times vv \leq \text{maxint}
then
    \text{queuetokens}_{N\text{var}} := \text{queuetokens}_{N\text{var}} \times vv
end ;
\text{bb} \leftarrow \text{queuetokens\_PRE\_SUB\_N\text{var}}\left( vv \right) \triangleq 
pre
    vv \in 0 .. \text{maxint}
then
    \text{bb} := \text{bool}\left( \text{queuetokens}_{N\text{var}} - vv \geq 0 \right)
end ;
queuetokens\_SUB\_N\text{var}\left( vv \right) \triangleq 
pre
vv ∈ 0 .. maxint ∧ queuetokens_Nvar − vv ≥ 0
then
queuetokens_Nvar := queuetokens_Nvar − vv
end;
bubble ← queuetokens_PRE_DIV_NVAR ( vv )
pre
vv ∈ 0 .. maxint
then
bubble := bool ( vv > 0 )
end;
queuetokens_DIV_NVAR ( vv )
pre
vv ∈ 1 .. maxint
then
queuetokens_Nvar := queuetokens_Nvar / vv
end;
bubble ← queuetokens_PRE_MOD_NVAR ( vv )
pre
vv ∈ 0 .. maxint
then
bubble := bool ( vv > 0 )
end;
queuetokens_MOD_NVAR ( vv )
pre
vv ∈ 1 .. maxint
then
queuetokens_Nvar := queuetokens_Nvar − vv × ( queuetokens_Nvar / vv )
end;
bubble ← queuetokens_EQL_NVAR ( vv )
pre
vv ∈ 0 .. maxint
then
bubble := bool ( queuetokens_Nvar = vv )
end;
bubble ← queuetokens_NEQ_NVAR ( vv )
pre
vv ∈ 0 .. maxint
then
bubble := bool ( queuetokens_Nvar ≠ vv )
end;
bubble ← queuetokens_GTR_NVAR ( vv )
pre
\( vv \in 0 \ldots \text{maxint} \)
then
\( bb := \text{bool} ( \text{queuetokens\_Nvar} > vv ) \)
end ;
\( bb \leftarrow \text{queuetokens\_GEQ\_NVAR} ( vv ) \triangleq \)
pre
\( vv \in 0 \ldots \text{maxint} \)
then
\( bb := \text{bool} ( \text{queuetokens\_Nvar} \geq vv ) \)
end ;
\( bb \leftarrow \text{queuetokens\_SMR\_NVAR} ( vv ) \triangleq \)
pre
\( vv \in 0 \ldots \text{maxint} \)
then
\( bb := \text{bool} ( \text{queuetokens\_Nvar} < vv ) \)
end ;
\( bb \leftarrow \text{queuetokens\_LEQ\_NVAR} ( vv ) \triangleq \)
pre
\( vv \in 0 \ldots \text{maxint} \)
then
\( bb := \text{bool} ( \text{queuetokens\_Nvar} \leq vv ) \)
end ;
\text{queuetokens\_SAV\_NVAR} \triangleq 
begin
skip
end ;
\text{queuetokens\_RST\_NVAR} \triangleq 
begin
\text{queuetokens\_Nvar} :\in 0 \ldots \text{maxint}
end ;
\text{queuetokens\_SAVN\_NVAR} \triangleq 
begin
skip
end ;
\text{queuetokens\_RSTN\_NVAR} \triangleq 
begin
\text{queuetokens\_Nvar} :\in 0 \ldots \text{maxint}
end
END
B.2 freetokens_set.mch

MACHINE freetokens_set ( VALUE , maxcrd )
SEES Bool_TYPE
VARIABLES freetokens_sset , freetokens_ordn

INVARIANT
freetokens_sset ∈ F ( VALUE ) ∧
freetokens_ordn ∈ perm ( freetokens_sset ) ∧
card ( freetokens_sset ) ≤ maxcrd

INITIALISATION freetokens_sset , freetokens_ordn := {} , []

OPERATIONS
bb ←− freetokens_FUL_SET ≡
begin
bb := bool ( card ( freetokens_sset ) = maxcrd )
end ;
bb ←− freetokens_XST_IDE_SET ( ii ) ≡
pre
ii ∈ 1 .. maxcrd
then
bb := bool ( ii ∈ 1 .. card ( freetokens_sset ) )
end ;
nn ←− freetokens_CRD_SET ≡
begin
nn := card ( freetokens_sset )
end ;
vv ←− freetokens_VAL_SET ( ii ) ≡
pre
ii ∈ 1 .. card ( freetokens_sset )
then
vv := freetokens_ordn ( ii )
end ;
vv ←− freetokens_ANY_SET ≡
pre
¬ ( freetokens_sset = {} )
then
vv ∈ freetokens_sset
end ;
freetokens_CLR_SET ≡
begin
freetokens_sset := {} ||
freetokens_ordn := []
end ;
freetokens_ENT_SET ( vv ) ≜
  pre
  vv ∈ VALUE ∧
  card ( freetokens_sset ) < maxrd
  then
  freetokens_sset := freetokens_sset ∪ { vv } ∥
  freetokens_ordn :∈ perm ( freetokens_sset ∪ { vv } )
  end :

freetokens_RMV_SET ( vv ) ≜
  pre
  vv ∈ VALUE
  then
  freetokens_sset := freetokens_sset − { vv } ∥
  freetokens_ordn :∈ perm ( freetokens_sset − { vv } )
  end :

bb ← freetokens_MBR_SET ( vv ) ≜
  pre
  vv ∈ VALUE
  then
  bb := bool ( vv ∈ freetokens_sset )
  end :

bb ← freetokens_EMP_SET ≜
  begin
  bb := bool ( freetokens_sset = {} )
  end :

freetokens_SAV_SET ≜ skip ;
freetokens_RST_SET ≜
  any rset , rseq where
  rset ∈ F ( VALUE ) ∧
  rseq ∈ perm ( rset ) ∧
  card ( rset ) ≤ maxcrd
  then
  freetokens_sset := rset ∥
  freetokens_ordn := rseq
  end :

freetokens_SAVN_SET ≜ skip ;
freetokens_RSTN_SET ≜
  any rset , rseq where
  rset ∈ F ( VALUE ) ∧
  rseq ∈ perm ( rset ) ∧
  card ( rset ) ≤ maxcrd
  then
  freetokens_sset := rset ∥
freetokens_ordn := rseq

end

END
B.3 qnext_Nfnc.mch

MACHINE qnext_Nfnc (maxint, maxfld)
CONSTRAINTS
  maxint ≤ 2147483646 ∧ maxfld ≤ 2147483646
SEES Bool_TYPE
VARIABLES qnext_Nfnc
INVARIANT qnext_Nfnc ∈ 1 .. maxfld → 0 .. maxint
INITIALISATION qnext_Nfnc := {}
\( qnext_{Nfnc} := \{ ff \} \triangleleft qnext_{Nfnc} \)

\[ \text{end} ; \]

\( qnext_{\text{ADD\_NFNC}}(ff, vv) \triangleq \]

\[ \text{pre} \]

\[ vv \in 0 \ldots \text{maxint} \land \]

\[ ff \in \text{dom}(qnext_{Nfnc}) \land \]

\[ qnext_{Nfnc}(ff) + vv \leq \text{maxint} \]

\[ \text{then} \]

\[ qnext_{Nfnc}(ff) := qnext_{Nfnc}(ff) + vv \]

\[ \text{end} ; \]

\( qnext_{\text{MUL\_NFNC}}(ff, vv) \triangleq \]

\[ \text{pre} \]

\[ vv \in 0 \ldots \text{maxint} \land \]

\[ ff \in \text{dom}(qnext_{Nfnc}) \land \]

\[ qnext_{Nfnc}(ff) \times vv \leq \text{maxint} \]

\[ \text{then} \]

\[ qnext_{Nfnc}(ff) := qnext_{Nfnc}(ff) \times vv \]

\[ \text{end} ; \]

\( qnext_{\text{SUB\_NFNC}}(ff, vv) \triangleq \]

\[ \text{pre} \]

\[ vv \in 0 \ldots \text{maxint} \land \]

\[ ff \in \text{dom}(qnext_{Nfnc}) \land \]

\[ qnext_{Nfnc}(ff) \geq vv \]

\[ \text{then} \]

\[ qnext_{Nfnc}(ff) := qnext_{Nfnc}(ff) - vv \]

\[ \text{end} ; \]

\( qnext_{\text{DIV\_NFNC}}(ff, vv) \triangleq \]

\[ \text{pre} \]

\[ vv \in 1 \ldots \text{maxint} \land \]

\[ ff \in \text{dom}(qnext_{Nfnc}) \]

\[ \text{then} \]

\[ qnext_{Nfnc}(ff) := qnext_{Nfnc}(ff) / vv \]

\[ \text{end} ; \]

\( qnext_{\text{MOD\_NFNC}}(ff, vv) \triangleq \]

\[ \text{pre} \]

\[ vv \in 1 \ldots \text{maxint} \land \]

\[ ff \in \text{dom}(qnext_{Nfnc}) \]

\[ \text{then} \]

\[ qnext_{Nfnc}(ff) := qnext_{Nfnc}(ff) - vv \times (qnext_{Nfnc}(ff) / vv) \]

\[ \text{end} ; \]

\( vv \leftarrow qnext_{\text{VAL\_NFNC}}(ff) \triangleq \]

\[ \text{pre} \]

\[ \]

53
\[ \text{if } ff \in \text{dom} ( \text{qnext}_Nfnc ) \text{ then} \]
\[ vv := \text{qnext}_Nfnc ( ff ) \]
\[ \text{end ;} \]
\[ bb \leftarrow \text{qnext}_EQLNFNC ( ff, vv ) \triangleq \]
\[ \text{pre} \]
\[ vv \in 0..\text{maxint} \land \]
\[ ff \in \text{dom} ( \text{qnext}_Nfnc ) \text{ then} \]
\[ bb := \text{bool} ( \text{qnext}_Nfnc ( ff ) = vv ) \]
\[ \text{end ;} \]
\[ bb \leftarrow \text{qnext}_{\text{NEQ}}NFNC ( ff, vv ) \triangleq \]
\[ \text{pre} \]
\[ vv \in 0..\text{maxint} \land \]
\[ ff \in \text{dom} ( \text{qnext}_Nfnc ) \text{ then} \]
\[ bb := \text{bool} ( \text{qnext}_Nfnc ( ff ) \neq vv ) \]
\[ \text{end ;} \]
\[ bb \leftarrow \text{qnext}_{\text{GTR}}NFNC ( ff, vv ) \triangleq \]
\[ \text{pre} \]
\[ vv \in 0..\text{maxint} \land \]
\[ ff \in \text{dom} ( \text{qnext}_Nfnc ) \text{ then} \]
\[ bb := \text{bool} ( \text{qnext}_Nfnc ( ff ) > vv ) \]
\[ \text{end ;} \]
\[ bb \leftarrow \text{qnext}_{\text{GEQ}}NFNC ( ff, vv ) \triangleq \]
\[ \text{pre} \]
\[ vv \in 0..\text{maxint} \land \]
\[ ff \in \text{dom} ( \text{qnext}_Nfnc ) \text{ then} \]
\[ bb := \text{bool} ( \text{qnext}_Nfnc ( ff ) \geq vv ) \]
\[ \text{end ;} \]
\[ bb \leftarrow \text{qnext}_{\text{SMR}}NFNC ( ff, vv ) \triangleq \]
\[ \text{pre} \]
\[ vv \in 0..\text{maxint} \land \]
\[ ff \in \text{dom} ( \text{qnext}_Nfnc ) \text{ then} \]
\[ bb := \text{bool} ( \text{qnext}_Nfnc ( ff ) < vv ) \]
\[ \text{end ;} \]
\[ bb \leftarrow \text{qnext}_{\text{LEQ}}NFNC ( ff, vv ) \triangleq \]
\[ \text{pre} \]
\[ vv \in 0..\text{maxint} \land \]
\[ ff \in \text{dom} ( qnext_{Nfnc} ) \]

then

\[ bb := \text{bool} ( qnext_{Nfnc} ( ff ) \leq vv ) \]

end ;

qnext_{SAV\_NFNC} \triangleq \begin{align*}
\text{skip}
\end{align*}

qnext_{RST\_NFNC} \triangleq \begin{align*}
\text{begin} \quad qnext_{Nfnc} : \in 1..\text{maxfld} \rightarrow 0..\text{maxint} \text{ end ;}
\end{align*}

qnext_{SAV\_NFNC} \triangleq \begin{align*}
\text{begin} \quad \text{skip} \quad \text{end ;}
\end{align*}

qnext_{RSTN\_NFNC} \triangleq \begin{align*}
\text{begin} \quad qnext_{Nfnc} : \in 1..\text{maxfld} \rightarrow 0..\text{maxint} \text{ end}
\end{align*}

END