Outline I

1. What is Refinement?
   - What does refinement guarantee?
   - Some things you can do in refinement
   - Reducing nondeterminism: examples
   - Weakening the precondition
   - Refinement is not equivalence
   - Refining the state
   - The Array machine
   - A refinement of the Array machine
   - A function machine
   - A refinement of the function machine
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;
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;
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;
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;
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.
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.
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.
**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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
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 a refined operation is specified on the assumption of the precondition, the refinement can do anything outside of the precondition.
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.
Reducing nondeterminism: examples

Specification

\[
\begin{align*}
\text{result} & \leftarrow \text{SimpleChoice} \ni \\
\text{result} & \in \{3, 6, 9, 12\}
\end{align*}
\]

Refinement

\[
\begin{align*}
\text{result} & \leftarrow \text{SimpleChoice} \ni \\
\text{result} & := 6
\end{align*}
\]
Reducing nondeterminism: examples

**Specification**

\[
\text{result} \leftarrow \text{SimpleChoice} \triangleq \\
\text{result} \in \{3, 6, 9, 12\};
\]

**Refinement**

\[
\text{result} \leftarrow \text{SimpleChoice} \triangleq \\
\text{result} := 6;
\]
Reducing nondeterminism: examples

**Specification**

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

**Refinement**

\[
\text{result} \leftarrow \text{SimpleChoice} \triangleq \\
\text{result} := 6;
\]
Weakening the precondition

**Specification**

\[
\text{result} \leftarrow \text{Divide} \ (n1, n2) \\
\text{pre} \quad n1 \in \mathbb{N} \land n2 \in \mathbb{N} \land n2 \neq 0 \quad \text{then} \quad \text{result} := \frac{n1}{n2} \\
\text{end} ;
\]

**Refinement**

\[
\text{result} \leftarrow \text{Divide} \ (n1, n2) \\
\quad \text{if} \quad n2 \neq 0 \quad \text{then} \quad \text{result} := \frac{n1}{n2} \\
\quad \text{else} \quad \text{result} := 42 \\
\text{end} ;
\]
Weakening the precondition

**Specification**

\[
result \leftarrow \text{Divide} ( n1, n2 ) \\
\text{pre} \quad n1 \in \mathbb{N} \land n2 \in \mathbb{N} \land n2 \neq 0 \quad \text{then} \\
\quad result := n1 / n2 \\
\text{end} ;
\]

**Refinement**

\[
result \leftarrow \text{Divide} ( n1, n2 ) \\
\text{if} \quad n2 \neq 0 \\
\text{then} \quad result := n1 / n2 \\
\text{else} \quad result := 42 \\
\text{end} ;
\]
Weakening the precondition

**Specification**

`result ← Divide (n1, n2) ≜ `  
`pre n1 ∈ N ∧ n2 ∈ N ∧ n2 ≠ 0 then `  
`result := n1 / n2 `  
`end ;`

**Refinement**

`result ← Divide (n1, n2) ≜ `  
`if n2 ≠ 0 then result := n1 / n2 `  
`else result := 42 `  
`end ;`
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

\[ \text{coin} \leftarrow \text{Flip} \mathrel{\hat{\equiv}} \text{coin} : \in \text{COIN} \]

Refinement

\[ \text{coin} \leftarrow \text{Flip} \mathrel{\hat{\equiv}} \text{coin} : \in \text{COIN} \]

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.
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

\[
\text{SETS } \text{COIN} = \{ \text{Head}, \text{Tail} \}
\]

Specification

\[
\text{coin} \leftarrow \text{Flip} \triangleq \text{coin} : \in \text{COIN}
\]

Refinement

\[
\text{coin} \leftarrow \text{Flip} \triangleq \text{coin} : \in \text{COIN}
\]

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.
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

\textbf{SETS} \ COIN = \{ \textit{Head}, \textit{Tail} \}

\begin{itemize}
\item \textbf{Specification}
\begin{align*}
\textit{coin} & \leftarrow \textit{Flip} \triangleq \\
\textit{coin} & \in \text{COIN}
\end{align*}
\item \textbf{Refinement}
\begin{align*}
\textit{coin} & \leftarrow \textit{Flip} \triangleq \\
\textit{coin} & \in \text{COIN}
\end{align*}
\end{itemize}

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.
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

\[
\text{SETS } COIN = \{ \text{Head}, \text{Tail} \}
\]

Specification

\[
\text{coin } \leftarrow \text{Flip } \uparrow \leftarrow \\
\text{coin } : \in COIN
\]

Refinement

\[
\text{coin } \leftarrow \text{Flip } \uparrow \\
\text{coin } : \in COIN
\]

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.
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

\[
\text{SETS } \text{COIN} = \{ \text{Head}, \text{Tail} \}
\]

Specification

\[
\text{coin } \leftarrow \text{Flip} \trianglelefteq \\
\text{coin } : \in \text{COIN}
\]

Refinement

\[
\text{coin } \leftarrow \text{Flip} \trianglelefteq \\
\text{coin } : \in \text{COIN}
\]

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

The COIN set

\[ \text{SETS } COIN = \{ \text{Head} , \text{Tail} \} \]

Specification

\[ \text{coin} \leftarrow \text{Flip} \triangleq \text{coin} : \in COIN \]

Refinement

\[ \text{coin} \leftarrow \text{Flip} \triangleq \text{coin} : \in COIN \]

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.
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

\[
\text{SETS } COIN = \{ \text{Head} , \text{Tail} \}
\]

Specification

\[
\text{coin } \leftarrow \text{Flip } \trianglelefteq \\
\text{coin } : \in COIN
\]

Refinement

\[
\text{coin } \leftarrow \text{Flip } \trianglelefteq \\
\text{coin } : \in COIN
\]

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.
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.
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.
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.
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.
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.
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.
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.

\[
\text{MACHINE } \text{Array} \ ( \text{maxarray} , \text{VAL} )\\
\text{CONSTRAINTS } \text{maxarray} \in \mathbb{N}_1\\
\text{VARIABLES } \text{array}\\
\text{INVARIANT } \text{array} \in 1..\text{maxarray} \rightarrow \text{VAL}\\
\text{INITIALISATION } \text{array} := \{\}\\
\]

The Array machine II

OPERATIONS

\[ \text{store} \ (pos, val) \triangleq \]
\[ \text{pre} \ pos \in \mathbb{N} \land val \in \text{VAL} \land pos \in 1 \ldots \text{maxarray} \text{ then} \]
\[ \text{array} (pos) := val \]
end ;

val ← get (pos) \triangleq

\[ \text{pre} \ pos \in \mathbb{N} \land pos \in \text{dom (array)} \text{ then} \]
\[ \text{val} := \text{array} (pos) \]
end
END
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.
A refinement of the Array machine II

**REFINEMENT** $ArrayR$
**REFINES** $Array$
**VARIABLES** $arrayr$
**INVARIANT**

\[
arrayr \in 1 \ldots \text{maxarray} \rightarrow \text{VAL} \wedge
\]

Refinement relation

\[
array \subseteq arrayr
\]

**INITIALISATION** $arrayr :\in 1 \ldots \text{maxarray} \rightarrow \text{VAL}$
A refinement of the Array machine III

OPERATIONS

store (pos, val) \equiv

arrayr (pos) := val ;

val ← get (pos) \equiv

val := arrayr (pos)

END
A function machine I

MACHINE Function ( maxdom )
CONSTRAINTS maxdom ∈ \( \mathbb{N}_1 \)
SETS DOM ; RAN
PROPERTIES card ( DOM ) = maxdom
VARIABLES fun
INARIANT fun ∈ DOM ↔ RAN
INITIALISATION fun := {}
A function machine II

OPERATIONS

\textbf{update} ( dval , rval ) \triangleq
\begin{align*}
\text{pre} & \quad dval \in \text{DOM} \land rval \in \text{RAN} \quad \text{then} \\
& \quad \text{fun} ( dval ) := rval
\end{align*}
\text{end ;}

rval \leftarrow \textbf{fetch} ( dval ) \triangleq
\begin{align*}
\text{pre} & \quad dval \in \text{DOM} \land dval \in \text{dom} ( \text{fun} ) \quad \text{then} \\
& \quad rval := \text{fun} ( dval )
\end{align*}
\text{end}

END
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.
Outline

Objectives of this Lecture

What is Refinement?

What does refinement guarantee?

Some things you can do in refinement

Reducing nondeterminism: examples

Weakening the precondition

Refinement is not equivalence

Refining the state

The Array machine

A refinement of the Array machine

A function machine

A refinement of the function machine

REFINEMENT  FunctionR

REFINES  Function
VARIABLES  fundom, funran

INARIANT

fundom ∈ iseq (DOM) ∧
funran ∈ seq (RAN) ∧
size (fundom) = size (funran) ∧

Refinement relation

fun = (fundom⁻¹ ; funran)

ASSERTIONS

dom (fundom) = dom (funran) ∧
dom (fun) = ran (fundom) ∧
ran (fun) = ran (funran) ∧
ran (fun) = ran (fundom⁻¹)
dom (funran) = ran (fundom⁻¹)

INITIALISATION  fundom, funran := [], []
OPERATIONS

update (dval, rval) ≜
begin
  if dval ∉ ran (fundom)
  then fundom := fundom ← dval
  end;
  funran (fundom⁻¹(dval)) := rval
end;

rval ← fetch (dval) ≜
rval := funran (fundom⁻¹(dval))

END
Towards a formal understanding of refinement

- Coin flip machine and refinement
- Operation refinement
- Simple refinement intuition
- Checking our intuition
- The effect of nondeterminism
- A revised formalisation of refinement
- Isolating the problem
- Final formulation
- Validating Flip under the new formulation
Outline II

- Notes on the conjugate weakest precondition
- Refinement and feasibility
- Avoiding the infeasible
- Formality does not guarantee feasibility
- Proving feasibility
- The Infeasible cannot be made Feasible
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.
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.
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.
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.
Towards a formal understanding of refinement

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* ∧  
Refinement relation  
*coinC* = *coinA*

**INITIALISATION**  
*coinC* :∈ *COIN*

**OPERATIONS**  
*Flip* \( \triangleq *coinC* :∈ *COIN*

**END**
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.

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>$v_A$</td>
</tr>
<tr>
<td></td>
<td>$v_C$</td>
</tr>
<tr>
<td>Invariant</td>
<td>$I_A$</td>
</tr>
<tr>
<td></td>
<td>$I_C$</td>
</tr>
<tr>
<td>Refinement relation</td>
<td>$R$</td>
</tr>
<tr>
<td>Operation</td>
<td>$Op_A$</td>
</tr>
<tr>
<td>Precondition</td>
<td>$P$</td>
</tr>
<tr>
<td>Operation body</td>
<td>$B_A$</td>
</tr>
<tr>
<td></td>
<td>$Op_C$</td>
</tr>
<tr>
<td></td>
<td>$B_C$</td>
</tr>
</tbody>
</table>

Without any loss of generality we will assume that the states $v_A$ and $v_C$ are disjoint.
A simple intuition about refinement is shown above.

This leads to the following mathematical formulation of refinement

\[
l_A \land l_C \land R \land P \Rightarrow [B_C ; B_A](l_A \land l_C \land R)
\]  (1)
Towards a formal understanding of refinement

Coin flip machine and refinement

Operation refinement

Simple refinement intuition

Checking our intuition

The effect of nondeterminism

A revised formalisation of refinement

Isolating the problem

Final formulation

Validating Flip under the new formulation

Notes on the conjugate weakest precondition

Refinement and feasibility

Avoiding the infeasible

Formality does not guarantee feasibility

Proving feasibility

The Infeasible cannot be made Feasible

Checking our intuition

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

We have to prove:

\[ \text{coin}A \in \text{COIN} \land \text{coin}C \in \text{COIN} \land \text{coin}C = \text{coin}A \]

\[ \Rightarrow \]

\[ [\text{coin}C \in \text{COIN} ; \text{coin}A \in \text{COIN}] \]

\[ (\text{coin}A \in \text{COIN} \land \text{coin}C \in \text{COIN} \land \text{coin}C = \text{coin}A) \]

To simplify the proof we will omit \(\text{coin}A \in \text{COIN}\) and \(\text{coin}C \in \text{COIN}\)
Checking our intuition

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

We have to prove:

\[
\text{coinA} \in \text{COIN} \land \text{coinC} \in \text{COIN} \land \text{coinC} = \text{coinA} \\
\Rightarrow \\
[\text{coinC} \in \text{COIN} ; \text{coinA} \in \text{COIN}] \\
(\text{coinA} \in \text{COIN} \land \text{coinC} \in \text{COIN} \land \text{coinC} = \text{coinA})
\]

To simplify the proof we will omit \(\text{coinA} \in \text{COIN}\) and \(\text{coinC} \in \text{COIN}\)
\[ coinC = coinA \]

\[ \Rightarrow [coinC \in COIN ; coinA \in COIN](coinC = coinA) \]
\[ \Rightarrow [coinC \in COIN][coinA \in COIN](coinC = coinA) \]
\[ \Rightarrow [coinC \in COIN][[coinA := \text{Head}](coinC = coinA) \land [coinA := \text{Tail}](coinC = coinA)] \]
\[ \Rightarrow [coinC \in COIN][(coinC = \text{Head}) \land (coinC = \text{Tail})] \]
\[ \Rightarrow [coinC := \text{Head}][(coinC = \text{Head}) \land (coinC = \text{Tail})] \land [coinC := \text{Tail}][(coinC = \text{Head}) \land (coinC = \text{Tail})] \]
\[ \Rightarrow (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \]
\[ \Rightarrow (\text{Head} = \text{Tail}) \]
\[ \Rightarrow \text{false} \]

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

What went wrong?
Thus, it appears that our initial intuition doesn’t work.

What went wrong?
$\text{coinC} = \text{coinA}$

\[ \Rightarrow [\text{coinC} : \in \text{COIN} ; \text{coinA} : \in \text{COIN}] (\text{coinC} = \text{coinA}) \]

\[ \Rightarrow [\text{coinC} : \in \text{COIN}] [\text{coinA} : \in \text{COIN}] (\text{coinC} = \text{coinA}) \]

\[ \Rightarrow [\text{coinC} : \in \text{COIN}] ([\text{coinA} := \text{Head}] (\text{coinC} = \text{coinA}) \land [\text{coinA} := \text{Tail}] (\text{coinC} = \text{coinA})) \]

\[ \Rightarrow [\text{coinC} : \in \text{COIN}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \]

\[ \Rightarrow [\text{coinC} := \text{Head}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \land [\text{coinC} := \text{Tail}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \]

\[ \Rightarrow (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \]

\[ \Rightarrow (\text{Head} = \text{Tail}) \]

\[ \Rightarrow \text{false} \]

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

What went wrong?
Towards a formal understanding of refinement

Coin flip machine and refinement
Operation refinement
Simple refinement intuition
Checking our intuition
The effect of nondeterminism
A revised formalisation of refinement
Isolating the problem
Final formulation
Validating Flip under the new formulation
Notes on the conjugate weakest precondition
Refinement and feasibility
Avoiding the infeasible
Formality does not guarantee feasibility
Proving feasibility
The Infeasible cannot be made Feasible

\[ \text{coinC} = \text{coinA} \]

\[ \Rightarrow \ [\text{coinC} \in \text{COIN} ; \ \text{coinA} \in \text{COIN}] (\text{coinC} = \text{coinA}) \]
\[ \Rightarrow \ [\text{coinC} \in \text{COIN}] [\text{coinA} \in \text{COIN}] (\text{coinC} = \text{coinA}) \]
\[ \Rightarrow \ [\text{coinC} \in \text{COIN}] ([\text{coinA} := \text{Head}] (\text{coinC} = \text{coinA}) \land \]
\[ [\text{coinA} := \text{Tail}] (\text{coinC} = \text{coinA})) \]
\[ \Rightarrow \ [\text{coinC} \in \text{COIN}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \]
\[ \Rightarrow \ [\text{coinC} := \text{Head}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \land \]
\[ [\text{coinC} := \text{Tail}] ((\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \]
\[ \Rightarrow \ (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \]
\[ \Rightarrow \ (\text{Head} = \text{Tail}) \]
\[ \Rightarrow \ \text{false} \]

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

What went wrong?
\[
\begin{align*}
\text{coinC} &= \text{coinA} \\
\Rightarrow & \quad [\text{coinC} \in \text{COIN} ; \text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA}) \\
\Rightarrow & \quad [\text{coinC} \in \text{COIN}][\text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA}) \\
\Rightarrow & \quad [\text{coinC} \in \text{COIN}][\text{coinA} := \text{Head}](\text{coinC} = \text{coinA}) \land \\
& \quad [\text{coinA} := \text{Tail}](\text{coinC} = \text{coinA}) \\
\Rightarrow & \quad [\text{coinC} \in \text{COIN}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \\
\Rightarrow & \quad [\text{coinC} := \text{Head}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \land \\
& \quad [\text{coinC} := \text{Tail}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \\
\Rightarrow & \quad (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail}) \\
\Rightarrow & \quad (\text{Head} = \text{Tail}) \\
\Rightarrow & \quad \text{false}
\end{align*}
\]

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

What went wrong?
Towards a formal understanding of refinement

Coin flip machine and refinement

Operation refinement

Simple refinement intuition

Checking our intuition

The effect of nondeterminism

A revised formalisation of refinement

Isolating the problem

Final formulation

Validating Flip under the new formulation

Notes on the conjugate weakest precondition

Refinement and feasibility

Avoiding the infeasible

Formality does not guarantee feasibility

Proving feasibility

The Infeasible cannot be made Feasible

\[\text{coinC} = \text{coinA}\]

\[\Rightarrow [\text{coinC} : \in \text{COIN} ; \text{coinA} : \in \text{COIN}](\text{coinC} = \text{coinA})\]

\[\Rightarrow [\text{coinC} : \in \text{COIN}][\text{coinA} : \in \text{COIN}](\text{coinC} = \text{coinA})\]

\[\Rightarrow [\text{coinC} : \in \text{COIN}][[\text{coinA} := \text{Head}](\text{coinC} = \text{coinA}) \land [\text{coinA} := \text{Tail}](\text{coinC} = \text{coinA})]\]

\[\Rightarrow [\text{coinC} : \in \text{COIN}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})]\]

\[\Rightarrow [\text{coinC} := \text{Head}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})] \land [\text{coinC} := \text{Tail}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})]\]

\[\Rightarrow (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail})\]

\[\Rightarrow (\text{Head} = \text{Tail})\]

\[\Rightarrow \text{false}\]

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

What went wrong?
$coinC = coinA$

$\Rightarrow [coinC : \in COIN ; \ coinA : \in COIN](coinC = coinA)$

$\Rightarrow [coinC : \in COIN][coinA : \in COIN](coinC = coinA)$

$\Rightarrow [coinC : \in COIN][(coinA := \text{Head})(coinC = coinA) \land$

$\quad [coinA := \text{Tail})(coinC = coinA))$

$\Rightarrow [coinC : \in COIN]((coinC = \text{Head}) \land (coinC = \text{Tail}))$

$\Rightarrow [coinC := \text{Head}]((coinC = \text{Head}) \land (coinC = \text{Tail})) \land$

$\quad [coinC := \text{Tail}]((coinC = \text{Head}) \land (coinC = \text{Tail}))$

$\Rightarrow (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail})$

$\Rightarrow (\text{Head} = \text{Tail})$

$\Rightarrow false$

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

What went wrong?
\[\text{coinC} = \text{coinA}\]

\[\Rightarrow [\text{coinC} \in \text{COIN} ; \text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA})\]

\[\Rightarrow [\text{coinC} \in \text{COIN}][\text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA})\]

\[\Rightarrow [\text{coinC} \in \text{COIN}][(\text{coinA} := \text{Head})(\text{coinC} = \text{coinA}) \land
[\text{coinA} := \text{Tail}](\text{coinC} = \text{coinA}))\]

\[\Rightarrow [\text{coinC} \in \text{COIN}][(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail}))\]

\[\Rightarrow [(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail})) \land
[(\text{coinC} := \text{Tail})(\text{coinC} = \text{Head}) \land (\text{coinC} = \text{Tail}))\]

\[\Rightarrow (\text{Head} = \text{Head}) \land (\text{Head} = \text{Tail}) \land (\text{Tail} = \text{Head}) \land (\text{Tail} = \text{Tail})\]

\[\Rightarrow (\text{Head} = \text{Tail})\]

\[\Rightarrow false\]

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

What went wrong?
\[ \text{coinC} = \text{coinA} \]

\[
\Rightarrow [\text{coinC} \in \text{COIN} ; \text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA})
\Rightarrow [\text{coinC} \in \text{COIN}][\text{coinA} \in \text{COIN}](\text{coinC} = \text{coinA})
\Rightarrow [\text{coinC} \in \text{COIN}][[\text{coinA} := \text{Head}](\text{coinC} = \text{coinA}) \wedge
[\text{coinA} := \text{Tail}](\text{coinC} = \text{coinA}))
\Rightarrow [\text{coinC} \in \text{COIN}][((\text{coinC} = \text{Head}) \wedge (\text{coinC} = \text{Tail})))
\Rightarrow [\text{coinC} := \text{Head}][((\text{coinC} = \text{Head}) \wedge (\text{coinC} = \text{Tail}))) \wedge
[\text{coinC} := \text{Tail}][((\text{coinC} = \text{Head}) \wedge (\text{coinC} = \text{Tail})))
\Rightarrow (\text{Head} = \text{Head}) \wedge (\text{Head} = \text{Tail}) \wedge (\text{Tail} = \text{Head}) \wedge (\text{Tail} = \text{Tail})
\Rightarrow (\text{Head} = \text{Tail})
\Rightarrow \text{false}
\]

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

What went wrong?
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 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.
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 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.
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 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.
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 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.
A revised formalisation of refinement

Figure: Accommodation of nondeterminism
A revised formalisation of refinement

Figure: Accommodation of nondeterminism
Consider any final state of the concrete operation, contained in the \textit{blue} set. Since $R$ is, in general, a relation the blue set of concrete states will be related to a the \textit{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 is that it requires \textit{all} reachable abstract states, the \textit{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.
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 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 the above diagram is as follows:

- from a single initial concrete state, $\bullet$, the operation $Op_C$ 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 $Op_A$ 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 $Op_C$ to be a refinement of $Op_A$, the green set must be contained in the red set. This needs to be true for all initial concrete states.
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 \(P\).
¬ [S]¬ P is sometimes called the \textit{conjugate weakest precondition} of S with respect to P.

We can now recast 1 as:

\[ I_A \land I_C \land R \land P \Rightarrow [B_C]¬ [B_A](¬ (I_C \land R)) \] (2)
Currently we have ignored operation results; we now have to take them into account. Suppose that the abstract operation is

\[
\text{result} \leftarrow Op_A = P | B_A
\]

and the concrete operation is

\[
\text{result} \leftarrow Op_C = P | B_C
\]

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 \(\text{result}'\) and let

\[
B'_C = [\text{result} := \text{result}'] B_C
\]

then the general refinement condition becomes

\[
I_A \land I_C \land R \land P \Rightarrow [B'_C] \neg [B_A](\neg (\text{result}' = \text{result} \land I_C \land R)) \quad (3)
\]
Currently we have ignored operation results; we now have to take them into account. Suppose that the abstract operation is

\[ result \leftarrow Op_A = P \mid B_A \]

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

\[ result \leftarrow Op_A = P \mid B_A \]

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

\[ B'_C = [\text{result} := \text{result'}]B_C \]

then the general refinement condition becomes

\[ I_A \land I_C \land R \land P \Rightarrow [B'_C] \neg [B_A](\neg (\text{result'} = \text{result} \land I_C \land R)) \]
Final formulation

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

\[ \text{result} \leftarrow Op_A = P \mid B_A \]

and the concrete operation is

\[ \text{result} \leftarrow Op_C = P \mid B_C \]

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 \( \text{result}' \) and let

\[ B'_C = [\text{result} := \text{result}'] B_C \]

then the general refinement condition becomes

\[ I_A \land I_C \land R \land P \Rightarrow [B'_C] \neg [B_A] (\neg (result' = result \land I_C \land R)) \quad (3) \]
Final formulation

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

\[
\text{result} \leftarrow \text{Op}_A = P \mid B_A
\]

and the concrete operation is

\[
\text{result} \leftarrow \text{Op}_C = P \mid B_C
\]

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 \(\text{result}'\) and let

\[
B'_C = [\text{result} := \text{result}'] B_C
\]

then the general refinement condition becomes

\[
I_A \land I_C \land R \land P \Rightarrow [B'_C] \neg [B_A](\neg (\text{result}' = \text{result} \land I_C \land R)) \quad (3)
\]
Final formulation

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

\[
\text{result} \leftarrow Op_A = P \mid B_A
\]

and the concrete operation is

\[
\text{result} \leftarrow Op_C = P \mid B_C
\]

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 \(\text{result}'\) and let

\[
B'_C = [\text{result} := \text{result}']B_C
\]

then the general refinement condition becomes

\[
I_A \land I_C \land R \land P \Rightarrow [B'_C] \neg [B_A](\neg (\text{result}' = \text{result} \land I_C \land R)) \quad (3)
\]
Validating Flip under the new formulation

\[ \text{coin}C = \text{coin}A \]

\[ \Rightarrow [\text{coin}C : \in \text{COIN}] \rightarrow [\text{coin}A : \in \text{COIN}] \rightarrow (\text{coin}C = \text{coin}A) \]

\[ \Rightarrow [\text{coin}C : \in \text{COIN}] \rightarrow [\text{coin}A : \in \text{COIN}] (\text{coin}C \neq \text{coin}A) \]

\[ \Rightarrow [\text{coin}C : \in \text{COIN}] (\text{(coin}C \neq \text{Hea}d) \land (\text{coin}C \neq \text{Tail})) \]

\[ \Rightarrow [\text{coin}C : \in \text{COIN}] (\text{(coin}C = \text{Hea}d) \lor (\text{coin}C = \text{Tail})) \]

\[ \Rightarrow ((\text{Hea}d = \text{Hea}d) \lor (\text{Hea}d = \text{Tail})) \land ((\text{Tail} = \text{Hea}d) \lor (\text{Tail} = \text{Tail})) \]

\[ \Rightarrow \text{true} \]
Validating Flip under the new formulation

\[ coinC = coinA \]
\[ \Rightarrow [coinC \in COIN] \rightarrow [coinA \in COIN] \rightarrow (coinC = coinA) \]
\[ \Rightarrow [coinC \in COIN] \rightarrow [coinA \in COIN] (coinC \neq coinA) \]
\[ \Rightarrow [coinC \in COIN] \rightarrow ((coinC \neq \text{Head}) \land (coinC \neq \text{Tail})) \]
\[ \Rightarrow [coinC \in COIN] ((coinC = \text{Head}) \lor (coinC = \text{Tail})) \]
\[ \Rightarrow ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land ((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail})) \]
\[ \Rightarrow \text{true} \]
Towards a formal understanding of refinement

Validating Flip under the new formulation

\[ \text{coinC} = \text{coinA} \]
\[ \Rightarrow [\text{coinC} \in \text{COIN}] \Rightarrow [\text{coinA} \in \text{COIN}] \Rightarrow (\text{coinC} = \text{coinA}) \]
\[ \Rightarrow [\text{coinC} \in \text{COIN}] \Rightarrow [\text{coinA} \in \text{COIN}] \Rightarrow (\text{coinC} \neq \text{coinA}) \]
\[ \Rightarrow [\text{coinC} \in \text{COIN}] \Rightarrow ((\text{coinC} \neq \text{Head}) \land (\text{coinC} \neq \text{Tail})) \]
\[ \Rightarrow [\text{coinC} \in \text{COIN}] \Rightarrow ((\text{coinC} = \text{Head}) \lor (\text{coinC} = \text{Tail})) \]
\[ \Rightarrow ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land ((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail})) \]
\[ \Rightarrow \text{true} \]
Validating Flip under the new formulation

\[
\begin{align*}
\text{coinC} &= \text{coinA} \\
\Rightarrow & \quad [\text{coinC} \in \text{COIN}] \land [\text{coinA} \in \text{COIN}] \land (\text{coinC} = \text{coinA}) \\
\Rightarrow & \quad [\text{coinC} \in \text{COIN}] \land [\text{coinA} \in \text{COIN}] (\text{coinC} \neq \text{coinA}) \\
\Rightarrow & \quad [\text{coinC} \in \text{COIN}] ((\text{coinC} \neq \text{Head}) \land (\text{coinC} \neq \text{Tail})) \\
\Rightarrow & \quad [\text{coinC} \in \text{COIN}] ((\text{coinC} = \text{Head}) \lor (\text{coinC} = \text{Tail})) \\
\Rightarrow & \quad ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land ((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail})) \\
\Rightarrow & \quad \text{true}
\end{align*}
\]
Validating Flip under the new formulation

\[ \text{coinC} = \text{coinA} \]

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

\[ \Rightarrow [\text{coinC} \in \text{COIN}] \neg [\text{coinA} \in \text{COIN}] (\text{coinC} \neq \text{coinA}) \]

\[ \Rightarrow [\text{coinC} \in \text{COIN}] ((\text{coinC} \neq \text{Head}) \land (\text{coinC} \neq \text{Tail})) \]

\[ \Rightarrow [\text{coinC} \in \text{COIN}] ((\text{coinC} = \text{Head}) \lor (\text{coinC} = \text{Tail})) \]

\[ \Rightarrow ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land ((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail})) \]

\[ \Rightarrow \text{true} \]
Validating Flip under the new formulation

\[
\text{coinC} = \text{coinA}
\]

\[
\Rightarrow [\text{coinC} : \in \text{COIN}] \neg [\text{coinA} : \in \text{COIN}] \neg (\text{coinC} = \text{coinA})
\]

\[
\Rightarrow [\text{coinC} : \in \text{COIN}] \neg [\text{coinA} : \in \text{COIN}] (\text{coinC} \neq \text{coinA})
\]

\[
\Rightarrow [\text{coinC} : \in \text{COIN}] \neg ((\text{coinC} \neq \text{Head}) \land (\text{coinC} \neq \text{Tail}))
\]

\[
\Rightarrow [\text{coinC} : \in \text{COIN}] ((\text{coinC} = \text{Head}) \lor (\text{coinC} = \text{Tail}))
\]

\[
\Rightarrow ((\text{Head} = \text{Head}) \lor (\text{Head} = \text{Tail})) \land ((\text{Tail} = \text{Head}) \lor (\text{Tail} = \text{Tail}))
\]

\[
\Rightarrow \text{true}
\]
Notes on the conjugate weakest precondition

If $S$ is deterministic then $\neg [S] \rightarrow P = [S]P$.

The behaviour of $\neg [S] \rightarrow P$ can be demonstrated as follows:

1. Assume that $S$ has the form $v : s^1$, then
   
   $\neg [S] \rightarrow P = \neg [v : s] \rightarrow P$  
   = $\neg \forall xx. (xx : s \Rightarrow \{v := xx\}(\neg P))$  
   = $\exists xx. (xx : s \land \neg \{v := xx\}(\neg P))$  
   = $\exists xx. (xx : s \land \{v := xx\}P)$

   Existential quantification captures the arbitrary choice from nondeterminism during refinement. In contrast,

   $[S]P = \forall xx. (xx : s \Rightarrow \{v := xx\}P)$

2. Assume that $S$ has the form $v := e$
   
   $\neg [S] \rightarrow P = \neg [v := e] \rightarrow P$  
   = $\neg (\neg [v := e]P)$  
   = $[v := e]P$  
   = $[S]P$

1 more generally we could use $S = S_1 || \ldots || S_n$
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 $v : \in s^1$, then
   $$\neg[S]\neg P = \neg[v : \in s]\neg P$$
   $$= \neg v \ xx. (xx : s \Rightarrow [v := xx]|P) \quad \text{(semantics of } v : \in s)$$
   $$= \exists xx. (xx : s \land \neg ([v := xx]|\neg P)) \quad \text{(} \neg \exists \neg \Rightarrow \exists \neg \land \neg \Rightarrow \exists$$
   $$= \exists xx. (xx : s \land [v := xx]|P) \quad \text{(} \neg \text{ distributes through simple substitution)}$$

Existential quantification captures the arbitrary choice from nondeterminism during refinement. In contrast,

$[S]P = \forall xx. (xx : s \Rightarrow [v := xx]|P)$

2. Assume that $S$ has the form $v := e$
   $$\neg[S]\neg P = \neg[v := e]\neg P$$
   $$= \neg(\neg[v := e]|P)$$
   $$= [v := e]|P$$
   $$= [S]P$$

\[1^\text{more generally we could use } S = S_1 \parallel \ldots \parallel S_n\]
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 $v : \in s^1$, then
   \[
   \neg [S] \neg P = \neg [v : \in s] \neg P \\
   = \neg \forall xx. (xx : s \Rightarrow [v := xx](\neg P)) \{ \text{semantics of } v : \in s \} \\
   = \exists xx. (xx : s \land \neg ([v := xx](\neg P))) \{ \neg \forall z. (P \Rightarrow Q) = \exists z. (P \land \neg Q) \} \\
   = \exists xx. (xx : s \land [v := xx]P) \{ \neg \text{ distributes through simple substitution} \}
   \]
   Existential quantification captures the arbitrary choice from nondeterminism during refinement. In contrast,
   $[S] P = \forall xx. (xx : s \Rightarrow [v := xx]P)$

2. Assume that $S$ has the form $v := e$
   \[
   \neg [S] \neg P = \neg [v := e] \neg P \\
   = \neg (\neg [v := e]P) \\
   = [v := e]P \\
   = [S] P
   \]

\[\text{more generally we could use } S = S_1 \parallel \ldots \parallel S_n\]
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 $v : \in s^1$, then

   \[
   \neg [S] \neg P = \neg [v : \in s] \neg P \\
   = \neg \forall xx. (xx : s \Rightarrow [v := xx](\neg P)) \{\text{semantics of } v : \in s\} \\
   = \exists xx. (xx : s \land \neg ([v := xx](\neg P))) \{\neg \forall z. (P \Rightarrow Q) = \exists z. (P \land \neg Q)\} \\
   = \exists xx. (xx : s \land [v := xx]P) \{\neg \text{ distributes through simple substitution}\}
   \]

Existential quantification captures the arbitrary choice from nondeterminism during refinement. In contrast,

$[S] P = \forall xx. (xx : s \Rightarrow [v := xx]P)$

2. Assume that $S$ has the form $v := e$

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

\[1\text{more generally we could use } S = S_1 \mid \ldots \mid S_n \]
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 $v : \in s^1$, then

   $\neg [S] \neg P = \neg [v : \in s] \neg P$
   $= \neg \forall xx. (xx : s \Rightarrow [v := xx](\neg P))$ \{semantics of $v : \in s$\}
   $= \exists xx. (xx : s \land \neg ([v := xx](\neg P)))$ \{ $\forall z. (P \Rightarrow Q) = \exists z. (P \land \neg Q)$ \}
   $= \exists xx. (xx : s \land [v := xx]P)$ \{ $\neg$ distributes through simple substitution \}

   Existential quantification captures the arbitrary choice from nondeterminism during refinement. In contrast,

   $[S]P = \forall xx. (xx : s \Rightarrow [v := xx]P)$

2. Assume that $S$ has the form $v := e$

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

1 more generally we could use $S = S_1 \ [\ldots \ [S_n$
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 $v : s^1$, then
   
   $\neg [S] \neg P = \neg [v : s] \neg P$
   
   $= \neg \forall xx. (xx : s \Rightarrow [v := xx](\neg P)) \{ \text{semantics of } v : s \}$
   
   $= \exists xx. (xx : s \land \neg ([v := xx](\neg P))) \{ \neg \forall z. (P \Rightarrow Q) = \exists z. (P \land \neg Q) \}$
   
   $= \exists xx. (xx : s \land [v := xx] P) \{ \neg \text{ distributes through simple substitution} \}$

   Existential quantification captures the arbitrary choice from nondeterminism during refinement. In contrast,
   
   $[S] P = \forall xx. (xx : s \Rightarrow [v := xx] P)$

2. Assume that $S$ has the form $v := e$

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

---

$^1$more generally we could use $S = S_1 \parallel \ldots \parallel S_n$
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 $v : \in s^1$, then
   
   $\neg [S] \neg P = \neg [v : \in s] \neg P$
   
   $= \neg \ \forall xx. \ (xx : s \Rightarrow [v := xx] (\neg P)) \{\text{semantics of } v : \in s\}$
   
   $= \exists xx. \ (xx : s \land \neg ([v := xx] (\neg P))) \\{\neg \ \forall z. (P \Rightarrow Q) = \exists z. (P \land \neg Q)\}$
   
   $= \exists xx. \ (xx : s \land [v := xx] P) \{\neg \ \text{distributes through simple substitution}\}$

   Existential quantification captures the arbitrary choice from nondeterminism during refinement. In contrast,

   $[S] P = \forall xx. \ (xx : s \Rightarrow [v := xx] P)$

2. Assume that $S$ has the form $v := e$
   
   $\neg [S] \neg P = \neg [v := e] \neg P$
   
   $= \neg (\neg [v := e] P)$
   
   $= [v := e] P$
   
   $= [S] P$

$^1$more generally we could use $S = S_1 \ [ \ldots \ [ S_n$
Refinement and feasibility

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

\begin{itemize}
\item for all $P$, $\lceil \text{Abort} \rceil P = \text{false}$, and
\item for all $P$, $\lceil \text{Magic} \rceil P = \text{true}$
\end{itemize}

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

\[ \text{Abort} \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots R_n \sqsubseteq \text{Magic} \]

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.

\[ ^2 \text{The ordering, in general, is not linear as shown here, but a lattice.} \]
Refinement and feasibility

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

1. for all $P$, $\lbrack\text{Abort}\rbrack P = \text{false}$, and
2. for all $P$, $\lbrack\text{Magic}\rbrack P = \text{true}$

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

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

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 The ordering, in general, is not linear as shown here, but a lattice.
Refinement and feasibility

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

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:\(^2\)

$$Abort \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots R_n \sqsubseteq Magic$$

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

\(^2\)The ordering, in general, is not linear as shown here, but a lattice.
Refinement and feasibility

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

1. for all $P$, $[\text{Abort}]P = \text{false}$, and
2. for all $P$, $[\text{Magic}]P = \text{true}$

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

$$Abort \sqsubseteq R_0 \sqsubseteq R_1 \sqsubseteq \cdots R_n \sqsubseteq Magic$$

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

---

\[2\] The ordering, in general, is not linear as shown here, but a lattice.
Refinement and feasibility

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

1. for all $P$, $[\text{Abort}]P = \text{false}$, and
2. for all $P$, $[\text{Magic}]P = \text{true}$

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

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

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.

---

$^2$The ordering, in general, is not linear as shown here, but a lattice.
Refinement and feasibility

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

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:\(^2\)

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

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.

\(^2\)The ordering, in general, is not linear as shown here, but a lattice.
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.
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.
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.
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.
Formality does not guarantee feasibility I

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.

\[
\begin{align*}
\text{MACHINE} & \quad \text{Fermat} \\
\text{CONSTANTS} & \quad \text{EXP} \\
\text{PROPERTIES} & \\
\text{EXP} & \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \land \\
\forall \ xx \ . \ ( xx \in \mathbb{N} \Rightarrow \text{EXP} ( xx , 0 ) = 1 ) \land \\
\forall ( xx , nn ) . ( xx \in \mathbb{N} \land nn \in \mathbb{N}_1 \Rightarrow \\
& \quad \text{EXP} ( xx , nn ) = xx \times \text{EXP} ( xx , nn - 1 ) )
\end{align*}
\]
Formality does not guarantee feasibility II

```
OPERATIONS
  aa, bb, cc ← Fermat ( nn ) ≜
  pre  nn ∈ \mathbb{N}
  then
    any  xx, yy, zz
    where xx ∈ \mathbb{N} ∧ yy ∈ \mathbb{N} ∧ zz ∈ \mathbb{N} ∧
      \text{EXP} ( xx, nn ) + \text{EXP} ( yy, nn ) = \text{EXP} ( zz, nn )
    then  aa, bb, cc := xx, yy, zz
  end
end
END
```
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.
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.
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.
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.
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.
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.
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.
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

\[
I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \neg [v_A : \in s] \neg (I_A \land I_C \land v_A = v_C)
\]

\[
= I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \neg [v_A : \in s] (\neg I_A \lor (v_A \neq v_C))
\]

\[
= I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \neg (\forall xx. (xx : s \Rightarrow [v_A := xx] (\neg I_A \lor (v_A \neq v_C))))
\]

\[
= I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \exists xx. (xx : s \land I_A \land I_C \land xx = v_C)
\]

\[
= I_A \land I_C \land v_A = v_C \Rightarrow \exists xx. (xx : s \land I_A \land I_C \land xx = e)
\]

\[
= false, \quad \text{if } s \text{ is empty}
\]

This demonstrates the sting in the tail of magic: it is truly impossible to implement.
Towards a formal understanding of refinement

Coin flip machine and refinement

Operation refinement

Simple refinement intuition

Checking our intuition

The effect of nondeterminism

A revised formalisation of refinement

Isolating the problem

Final formulation

Validating Flip under the new formulation

Notes on the conjugate weakest precondition

Refinement and feasibility

Avoiding the infeasible

Formality does not guarantee feasibility

Proving feasibility

The Infeasible cannot be made Feasible

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

\[
I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \neg [v_A :\in s] \neg (I_A \land I_C \land v_A = v_C)
\]

\[
= I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \neg [v_A :\in s] (\neg I_A \lor (v_A \neq v_C))
\]

\[
= I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \neg (\forall xx. (xx : s \Rightarrow [v_A := xx](\neg I_A \lor (v_A \neq v_C))))
\]

\[
= I_A \land I_C \land v_A = v_C \Rightarrow [v_C := e] \exists xx. (xx : s \land I_A \land I_C \land xx = v_C)
\]

\[
= I_A \land I_C \land v_A = v_C \Rightarrow \exists xx. (xx : s \land I_A \land I_C \land xx = e)
\]

= \textit{false}, \hspace{1em} \text{if } s \text{ is empty}

This demonstrates the sting in the tail of \textit{magic}: it is truly impossible to implement.
Towards a formal understanding of refinement

Coin flip machine and refinement

Operation refinement

Simple refinement intuition

Checking our intuition

The effect of nondeterminism

A revised formalisation of refinement

Isolating the problem

Final formulation

Validating Flip under the new formulation

Notes on the conjugate weakest precondition

Refinement and feasibility

Avoiding the infeasible

Formality does not guarantee feasibility

Proving feasibility

The Infeasible cannot be made Feasible

---

Now suppose that we claim to refine the body of that operation to $v_C := e$, i.e., a deterministic refinement, with invariant $I_B$ and refinement relation $v_A = v_C$,

then we would have to prove

$$l_A \land l_C \land v_A = v_C \Rightarrow [v_C := e] \neg [v_A : s] \neg (l_A \land l_C \land v_A = v_C)$$

$$= l_A \land l_C \land v_A = v_C \Rightarrow [v_C := e] \neg [v_A : s] (\neg l_A \lor (v_A \neq v_C))$$

$$= l_A \land l_C \land v_A = v_C \Rightarrow [v_C := e] \neg (\forall xx.(xx : s \Rightarrow [v_A := xx](\neg l_A \lor (v_A \neq v_C))))$$

$$= l_A \land l_C \land v_A = v_C \Rightarrow [v_C := e] \exists xx.(xx : s \land l_A \land l_C \land xx = v_C)$$

$$= l_A \land l_C \land v_A = v_C \Rightarrow \exists xx.(xx : s \land l_A \land l_C \land xx = e)$$

$$= false, \quad \text{if } s \text{ is empty}$$

This demonstrates the sting in the tail of magic: it is truly impossible to implement.
Now suppose that we claim to refine the body of that operation to
\( v_C := e \), i.e., a deterministic refinement, with invariant \( I_B \) and refinement relation \( v_A = v_C \),

then we would have to prove

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

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

- A Simple Complete Development
- A second complete implementation
- Adding an interface to Queue.mch
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;
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;
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;
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;
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;
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;
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;
We will present complete developments of simple stateless machines. All refinements are algorithmic refinements.

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

- **Sqr** is an operation with a single natural number argument that computes the numerical square of the value of the argument.
- **ApproxSqrt**(num) returns the largest natural number that does not exceed $\sqrt{\text{num}}$, the real mathematical square root of *num*. 
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.
We will present complete developments of simple stateless machines. All refinements are algorithmic refinements.

Consider the following two machines, \textit{Square}, and \textit{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.
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 $num$. 
A square operation

MACHINE  *Square*

OPERATIONS

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

end

END
A square root operation

MACHINE  \textit{SquareRoot}

OPERATIONS
\[
\textit{sqrt} \leftarrow \textit{ApproxSqrt} \ ( \textit{num} ) \ 
\text{pre} \quad \textit{num} \in \mathbb{N} \ \text{then}
\begin{align*}
\text{any} & \quad \textit{approx} \quad \text{where} \\
\text{approx} & \in \mathbb{N} \land \\
\text{approx} \times \text{approx} & \leq \textit{num} \land \\
\textit{num} & < ( \text{approx} + 1 ) \times ( \text{approx} + 1 )
\end{align*}
\text{then} \quad \textit{sqrt} := \textit{approx}
\end{align*}
\]
end
end
END
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.
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.
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*. 
Implementation of Square

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

\begin{verbatim}
IMPLEMENTATION   SquareI
REFINES     Square

OPERATIONS
  \text{sqr} \leftarrow \text{Sqr} ( \text{num} ) \equiv
  \begin{align*}
  \text{begin} & \\
  \text{sqr} := \text{num} \times \text{num} & \\
  \text{end} & \\
  \end{align*}
END
\end{verbatim}
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:

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

It is relatively easy to choose a value of \text{approx} to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, \text{low} and \text{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 \text{SquareRootI}.

Notice that this machine \text{IMPORTS} a new machine \text{SqrtUtility}.
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:

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

It is relatively easy to choose a value of \( \text{approx} \) to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, \( \text{low} \) and \( \text{high} \) 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 \text{SquareRootI}.

Notice that this machine \text{IMPORTS} a new machine \text{SqrtUtility}. 
Implementing $\text{ApproxSqrt}$

The problem with the specification of $\text{ApproxSqrt}$ is that we have to choose the value of a single variable that satisfies two conjuncts:

$$\text{approx} \times \text{approx} \leq \text{num} \quad \text{and} \quad \text{num} < (\text{approx} + 1) \times (\text{approx} + 1)$$

It is relatively easy to choose a value of $\text{approx}$ to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, $\text{low}$ and $\text{high}$, 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 $\text{SquareRootI}$. Notice that this machine $\text{IMPORTS}$ a new machine $\text{SqrtUtility}$. 
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:

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

It is relatively easy to choose a value of \text{approx} to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, \text{low} and \text{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 \text{SquareRootI}.

Notice that this machine IMPORTS a new machine \text{SqrtUtility}. 
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:

$$approx \times approx \leq num \text{ and } num < (approx + 1) \times (approx + 1)$$

It is relatively easy to choose a value of $approx$ to satisfy either conjunct, but not both.

Thus we refine the operation by using two variables, $low$ and $high$ with

$$low \times low \leq num$$
$$num < high \times high$$
$$low + 1 \leq high$$

as shown in the refinement, and implementation machine \textbf{SquareRootI}.

Notice that this machine \textit{IMPORTS} a new machine \textbf{SqrtUtility}.
IMPLEMENTATION  SquareRootI
REFINES  SquareRoot
IMPORTS  SqrtUtility

OPERATIONS
\( \text{sqrt} \leftarrow \text{ApproxSqrt} \left( \text{num} \right) \triangleq \)

\begin{verbatim}
var low, high in
low, high ← ChooseInitApprox (num);
while low + 1 ≠ high do
  low, high ← ChooseBetterApprox (num, low, high)
end

invariant
low ∈ \( \mathbb{N} \) ∧ high ∈ \( \mathbb{N} \) ∧
low + 1 ≤ high ∧
low × low ≤ num ∧
num < high × high

variant
high - low

end;

sqrt := low

end

END
\end{verbatim}
The **SqrtUtility** machine has operations for choosing an initial approximation for the variables *low* and *high*, and an operation for *improving* the values of those variables.
SqrtUtility II

MACHINE SqrtUtility

OPERATIONS

\[ \text{low}, \text{high} \leftarrow \text{ChooseInitApprox} \left( \text{num} \right) \]

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

\[ \text{any} \quad xx, yy \quad \text{where} \]

\[ xx \in \mathbb{N} \land yy \in \mathbb{N} \land \]

\[ xx + 1 \leq yy \land \]

\[ xx \times xx \leq \text{num} \land \]

\[ \text{num} < yy \times yy \]

\[ \text{then} \quad \text{low}, \text{high} := xx, yy \]

\[ \text{end} \]

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

pre num ∈ ℕ ∧ oldlow ∈ ℕ ∧ oldhigh ∈ ℕ ∧
oldlow × oldlow ≤ num ∧ num < oldhigh × oldhigh ∧
oldlow + 1 < oldhigh then
any xx, yy where
xx ∈ ℕ ∧ yy ∈ ℕ ∧
xx + 1 ≤ yy ∧
xx × xx ≤ num ∧
num < yy × yy ∧
yy − xx < oldhigh − oldlow
then low, high := xx, yy
end
end
END
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**.
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**.
The first refinement \textit{SqrtUtilityR I}

\begin{align*}
\text{REFINEMENT} & \quad \textit{SqrtUtilityR} \\
\text{REFINES} & \quad \textit{SqrtUtility} \\
\text{OPERATIONS} & \quad \text{ChooseInitApprox} ( \textit{num} ) \\
& \quad \text{low, high} \leftarrow \text{ChooseInitApprox} ( \textit{num} ) \\
& \quad \text{low, high} := 0 , ( \textit{num} + 1 ) / 2 + 1 ;
\end{align*}
The first refinement \texttt{SqrtUtility}\texttt{R II}

\[
\text{low, high} \leftarrow \text{ChooseBetterApprox}(\text{num, oldlow, oldhigh}) \\
\text{any mid where} \\
\text{mid} \in \mathbb{N} \land \text{oldlow} < \text{mid} \land \text{mid} < \text{oldhigh} \text{ then} \\
\text{select mid} \times \text{mid} \leq \text{num} \text{ then} \\
\text{low, high} := \text{mid, oldhigh} \\
\text{when num} < \text{mid} \times \text{mid} \text{ then} \\
\text{low, high} := \text{oldlow, mid} \\
\text{end} \\
\text{end} \\
\text{END}
\]
The second refinement \textit{SqrtUtilityRR I}

\begin{verbatim}
REFINEMENT \textit{SqrtUtilityRR} \\
REFINES \textit{SqrtUtilityR}

OPERATIONS \\
\begin{array}{l}
\text{low}, \text{high} \leftarrow \text{ChooseInitApprox} (\text{num}) \\
\text{begin} \quad \text{low} := 0 ; \quad \text{high} := (\text{num} + 1) / 2 + 1 \quad \text{end} ;
\end{array}
\end{verbatim}
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
Finally the implementation SqrtUtilityRRI I

IMPLEMENTATION SqrtUtilityRRI
REFINES SqrtUtilityRR

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

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

\begin{verbatim}
var mid in
  mid := (oldlow + oldhigh) / 2;
  if mid \leq \text{num} / \text{mid}
    then low := mid; high := oldhigh
  else low := oldlow; high := mid
end
end
END
\end{verbatim}

Notice that the one change in the last refinement step is to replace
the guard \( mid \times mid \leq \text{num} \) by the equivalent \( mid \leq \text{num} / mid \). The
former is logically the condition we want, but it could overflow in a
finite integer implementation; the latter is equivalent, but avoids
overflow.
Having produced an implementation machine, *SqrtUtilityRRI*, we can produce code by selecting the `trl` button in the *Translators* environment of the B-Toolkit.

The C code is shown in the following frames.
Generating code

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

The C code is shown in the following frames.
#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;
}
#include "SqrtUtility.h"

void INI_SqrtUtility()
{
    ;
}

void ChooseInitApprox(_low , _high,_num)
int *__low,*_high,_num;
{
    *__low = 0;
    *__high = (_num+1)/2+1;
}
void
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;
    }
}
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.
Generating an interface

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

The interface can be run by selecting the \texttt{exe} (execute) button in the \textit{Translators} environment.

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

- `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.

- `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`. 

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`. 
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:

\[\text{queueid} \leftarrow \text{Enqueue}(item)\] 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.

\[\text{item} \leftarrow \text{Dequeue}\] an operation that returns the item that is at the head of the queue.

\[\text{Unqueue}(qid)\] removes the item from the queue identified by the queue identifier, \(qid\).
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:

\[
\text{queueid} \leftarrow \text{Enqueue}(\text{item}) \quad \text{an operation that places an item on the end of the queue. The operation returns a token that uniquely identifies this instance of the item in the queue. A queue can contain multiple instances of the same item value.}
\]

\[
\text{item} \leftarrow \text{Dequeue} \quad \text{an operation that returns the item that is at the head of the queue.}
\]

\[
\text{Unqueue}(\text{qid}) \quad \text{removes the item from the queue identified by the queue identifier, qid.}
\]
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:

\[
\text{queueid} \leftarrow \text{Enqueue}(\text{item}) \quad \text{an operation that places an item on the end of the queue. The operation returns a token that uniquely identifies this instance of the item in the queue. A queue can contain multiple instances of the same item value.}
\]

\[
\text{item} \leftarrow \text{Dequeue} \quad \text{an operation that returns the item that is at the head of the queue.}
\]

\[
\text{Unqueue(qid)} \quad \text{removes the item from the queue identified by the queue identifier, qid.}
\]
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.

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.
The Queue machine I

MACHINE Queue_ctx
SETS QUEUE
END

MACHINE Queue (ITEM, maxqueue)
CONSTRAINTS maxqueue ∈ N
SEES Queue_ctx
VARIABLES
  queuetokens,
  queue,
  queueitem
INVARIANT
  queuetokens ⊆ QUEUE ∧
  queue ∈ iseq (queuetokens) ∧
  size (queue) ≤ maxqueue ∧
The Queue machine II

\[
\text{queuetokens} = \text{ran (queue)} \land
\]
\[
\text{queueitem} \in \text{queuetokens} \rightarrow \text{ITEM}
\]

\text{ASSERTIONS} \quad \text{card (queuetokens)} = \text{size (queue)}

\text{INITIALISATION} \quad \text{queuetokens}, \text{queue}, \text{queueitem} := \{\}, \{\}, \{\}
The Queue machine III

OPERATIONS

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

\[ \text{enqueueid} \leftarrow \text{Enqueue}(\text{item}) \triangleq \]
\[ \text{pre \hspace{1em}} \text{item} \in \text{ITEM} \land \text{size}(\text{queue}) \neq \text{maxqueue} \text{ \hspace{1em} then} \]
\[ \text{any \hspace{1em}} \text{qid} \text{ \hspace{1em} where} \hspace{1em} \text{qid} \in \text{QUEUE} - \text{queuetokens} \text{ \hspace{1em} then} \]
\[ \text{queuetokens} := \text{queuetokens} \cup \{ \text{qid} \} \parallel \]
\[ \text{queue} := \text{queue} \leftarrow \text{qid} \parallel \]
\[ \text{queueitem}(\text{qid}) := \text{item} \parallel \]
\[ \text{enqueueid} := \text{qid} \]
\[ \text{end} \]
\[ \text{end} ; \]
Dequeue returns the item at the head of the queue

\[
\text{item} \leftarrow \text{Dequeue} \quad \triangleq \\
\quad \text{pre} \quad 0 < \text{size (queue)} \quad \text{then} \\
\quad \quad \text{any} \quad qfirst \quad \text{where} \quad qfirst \in \text{ran (queue)} \land qfirst = \text{queue (1)} \quad \text{then} \\
\quad \quad \quad \text{item} := \text{queueitem (qfirst)} \parallel \text{queue} := \text{tail (queue)} \parallel \\
\quad \quad \quad \text{queuetokens} := \text{queuetokens} - \{ qfirst \} \parallel \\
\quad \quad \quad \text{queueitem} := \{ qfirst \} \triangleleft \text{queueitem} \\
\quad \quad \text{end} \\
\quad \text{end} ;
\]
The Queue machine V

Unqueue(qid) removes an item from the queue

Unqueue ( qid ) ≜
  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 \} \leftarrow queueitem
  end
end
END
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;
- \texttt{qsize} the size of the queue.

Additionally, the refinement has variables \texttt{queuetokens} and \texttt{queueitem}, which have the same role as they did in the 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.

Refinements may not use variables of the refined machine except in invariants. Complete hiding is enforced.
Refining the Queue machine

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

- **qfirst**: a pointer to the first element on the queue;
- **qlast**: a pointer to the last element on the queue;
- **qnext**: a pointer that links the elements of the queue—relevant only to queues with more than one item;
- **qsize**: the size of the queue.

Additionally, the refinement has variables *queuetokens* and *queueitem*, which have the same role as they did in the *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.

Refinements may not use variables of the refined machine except in invariants. *Complete hiding* is enforced.
Refining the Queue machine

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

- **qfirst**: a pointer to the first element on the queue;
- **qlast**: a pointer to the last element on the queue;
- **qnext**: a pointer that links the elements of the queue—relevant only to queues with more than one item;
- **qsize**: the size of the queue.

Additionally, the refinement has variables **queuetokens** and **queueitem**, which have the same role as they did in the 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.

Refinements may not use variables of the refined machine except in invariants. **Complete hiding** is enforced.
Refining the Queue machine

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

- **qfirst**: a pointer to the first element on the queue;
- **qlast**: a pointer to the last element on the queue;
- **qnext**: a pointer that links the elements of the queue —relevant only to queues with more than one item;
- **qsize**: the size of the queue.

Additionally, the refinement has variables `queuetokens` and `queueitem`, which have the same role as they did in the 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.

Refinements may not use variables of the refined machine except in invariants. *Complete hiding* is enforced.
A Simple Complete Development

A second complete implementation

Adding an interface to Queue.mch

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;
- \texttt{qsize} the size of the queue.

Additionally, the refinement has variables \texttt{queuetokens} and \texttt{queueitem}, which have the same role as they did in the \texttt{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.

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

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

qfirst  a pointer to the first element on the queue;
qlast   a pointer to the last element on the queue;
qnext   a pointer that links the elements of the queue—relevant only to queues with more than one item;
qsize   the size of the queue.

Additionally, the refinement has variables queuetokens and queueitem, which have the same role as they did in the 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.

Refinements may not use variables of the refined machine except in invariants. Complete hiding is enforced.
Refining the Queue machine

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

- **qfirst**  a pointer to the first element on the queue;
- **qlast**   a pointer to the last element on the queue;
- **qnext**  a pointer that links the elements of the queue —relevant only to queues with more than one item;
- **qsize**  the size of the queue.

Additionally, the refinement has variables **queuetokens** and **queueitem**, which have the same role as they did in the *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.

Refinements may not use variables of the refined machine except in invariants. *Complete hiding* is enforced.
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*}
q_{\text{first}} & \quad \text{gives the identity of the first item in the list} \\
q_{\text{next}}(q_{\text{first}}) & \quad \text{gives the identity of the second item in the list} \\
(q_{\text{next}} ; q_{\text{next}})(q_{\text{first}}) & \quad \text{gives the identity of the third item in the list} \\
\ldots & \quad \text{etc}
\end{align*}

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

If \( r \in X \leftrightarrow X \), then \( r^0 = id(X) \) and \( r^{n+1} = r^n ; r \).
Relational composition and iteration

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

\[
q_{first} \quad \text{gives the identity of the first item in the list}
\]

\[
q_{next}(q_{first}) \quad \text{gives the identity of the second item in the list}
\]

\[
(q_{next} \circ q_{next})(q_{first}) \quad \text{gives the identity of the third item in the list}
\]

\[
\ldots
\]

etc

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

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

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

\[
\begin{align*}
\texttt{qfirst} & \quad \text{gives the identity of the first item in the list} \\
\texttt{qnext(qfirst)} & \quad \text{gives the identity of the second item in the list} \\
(q\texttt{next }; \texttt{qnext})(\texttt{qfirst}) & \quad \text{gives the identity of the third item in the list} \\
& \quad \text{etc}
\end{align*}
\]

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

If \( r \in X \leftrightarrow X \), then \( r^0 = id(X) \) and \( r^{n+1} = r^n ; r \).
Relational composition and iteration

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

$q_{first}$ gives the identity of the first item in the list
$qnext(q_{first})$ gives the identity of the second item in the list
$(qnext; qnext)(q_{first})$ gives the identity of the third item in the list
...

etc

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

If $r \in X \leftrightarrow X$, then $r^0 = id(X)$ and $r^{n+1} = r^n \; r$. 
Relational composition and iteration

Since we are modelling a list structure by the use of the \texttt{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 & \quad \text{gives the identity of the first item in the list} \\
  qnext(qfirst) & \quad \text{gives the identity of the second item in the list} \\
  (qnext \circ qnext)(qfirst) & \quad \text{gives the identity of the third item in the list} \\
  \ldots & \quad \text{etc}
\end{align*}

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

If \( r \in X \leftrightarrow X \), then \( r^0 = id(X) \) and \( r^{n+1} = r^n \circ r \).
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*}
\text{qfirst} & \quad \text{gives the identity of the first item in the list} \\
\text{qnext(qfirst)} & \quad \text{gives the identity of the second item in the list} \\
(q\text{next} ; q\text{next})(q\text{first}) & \quad \text{gives the identity of the third item in the list} \\
\ldots & \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.

If \( r \in X \leftrightarrow X \), then \( r^0 = id(X) \) and \( r^{n+1} = r^n ; r \).
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*}
\text{\texttt{qfirst}} & \quad \text{gives the identity of the first item in the list} \\
\text{\texttt{qnext(qfirst)}} & \quad \text{gives the identity of the second item in the list} \\
(\text{\texttt{qnext ; qnext}})(\text{\texttt{qfirst}}) & \quad \text{gives the identity of the third item in the list} \\
\ldots & \quad \text{etc}
\end{align*}

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

If \( r \in X \leftrightarrow X \), then \( r^0 = \text{id}(X) \) and \( r^{n+1} = r^n ; r \).
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*}
q_{\text{first}} & \quad \text{gives the identity of the first item in the list} \\
q_{\text{next}}(q_{\text{first}}) & \quad \text{gives the identity of the second item in the list} \\
(q_{\text{next}} ; q_{\text{next}})(q_{\text{first}}) & \quad \text{gives the identity of the third item in the list} \\
\ldots & \quad \text{etc}
\end{align*}

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

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

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

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

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

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

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.

\[†\) It should be clear that continuous composition of a relation with itself will eventually reach a stationary relation.\]
Closure

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

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

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

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

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.

\[\dagger\] It should be clear that continuous composition of a relation with itself will eventually reach a stationary relation.
Relation composition of functions

It should be clear that if $f$ is a function then $f; f$ 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.
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.
Relation composition of functions

It should be clear that if $f$ is a function then $f; f$ 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.
Relation composition of functions

It should be clear that if \( f \) is a function then \( f ; f \) 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.
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.
Relation composition of functions

It should be clear that if $f$ is a function then $f; f$ 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.
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,

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

We should be able to prove as an assertion

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

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

\[ qnext^*[\{qfirst\}] = queuetokens \]
The invariant of QueueR

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

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

and there are no loops,

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

We should be able to prove as an assertion

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

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

\[ qnext^*[\{qfirst\}] = \text{queue tokens} \]
The invariant of QueueR

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

$$qnext \in \text{QUEUE} \mapsto \text{QUEUE}$$

and there are no loops,

$$qnext \land id(\text{QUEUE}) = \{\}$$

We should be able to prove as an assertion

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

If $queue\text{tokens}$ is the set of tokens currently in the list representation of the queue then,

$$qnext^*[\{qfirst\}] = queue\text{tokens}$$
The invariant of QueueR

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

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

and there are no loops,

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

We should be able to prove as an assertion

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

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

\[ qnext^*[\{qfirst\}] = \text{queuetokens} \]
The invariant of QueueR

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

\[ qnext \in QUEUE \mapsto QUEUE \]

and there are no loops,

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

We should be able to prove as an assertion

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

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

\[ qnext^*[\{qfirst\}] = queuetokens \]
The invariant of QueueR

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

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

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

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

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

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

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

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

for non-empty lists.
The invariant of QueueR

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

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

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

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

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

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

If you take $qsize - 1$ steps along the list for the first node you will reach the last node,

$qnext^{qsize-1}(qfirst) = qlast$

for non-empty lists.
The invariant of QueueR

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

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

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

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

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

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

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

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

for non-empty lists.
The invariant of QueueR

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

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

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

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

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

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

If you take $qsize - 1$ steps along the list for the first node you will reach the last node,

$$qnext^{qsize - 1}(qfirst) = qlast$$

for non-empty lists.
The refinement relation of QueueR

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,

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

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

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,

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

REFINEMENT QueueR
REFINES Queue
SEES Queue_ctx
VARIABLES
  queuetokens, queueitem,
  qfirst, qlast, qnext, qsize
The QueueR machine II

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

\[ \text{queueitem} \in \text{queuetokens} \rightarrow \text{ITEM} \land \]

\[ \text{qfirst} \in \text{QUEUE} \land \text{qlast} \in \text{QUEUE} \land \]

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

\[ (0 < \text{qsize} \Rightarrow \text{qfirst} \in \text{queuetokens}) \land \]

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

\[ (0 < \text{qsize} \Rightarrow \text{qlast} \in \text{queuetokens}) \land \]

the list is linear and connected with no loops; hence qnext is injective
The QueueR machine III

\[ q_{next} \in \text{QUEUE} \mapsto \text{QUEUE} \land \]

the queue has no loops

\[ q_{next} \cap \text{id ( QUEUE )} = \{\} \land \]

\[ \bigcup nn \cdot ( nn \in \mathbb{N}_1 \mid \text{qnext}^{nn} ) \cap \text{id ( QUEUE )} = \{\} \land \]

All of the tokens in the queue are in queuetokens

(\( 0 < qsize \Rightarrow \text{queuetokens} = \text{qnext}^* [ \{ \text{qfirst} \} ] \) ) \land

\[ qsize \in \mathbb{N} \land \]

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

(\( 1 < qsize \Rightarrow \forall nn . ( nn \in \mathbb{N} \land nn \in 0 .. qsize - 2 \Rightarrow \text{qnext}^{nn} ( \text{qfirst} ) \in \text{dom ( qnext )} ) ) \land \]

all items in the queue, except the first, have a predecessor
The QueueR machine IV

\[ 1 < \text{qsize} \Rightarrow \forall nn . \ ( nn \in \mathbb{N} \land nn \in 1 \ldots \text{qsize} - 1 \Rightarrow \]
\[ \text{qnext}^{nn} ( \text{qfirst} ) \in \text{ran} ( \text{qnext} ) ) ) \land \]

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

\[ \text{qfirst} \notin \text{ran} ( \text{qnext} ) \land \]

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

\[ ( \text{qsize} = 1 \Rightarrow \text{qlast} = \text{qfirst} ) \land \]

the last item does not have a next item

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

taking qsize-1 steps from qfirst reaches qlast

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

each queue item follows at most one other item
The QueueR machine \( V \)

\[ \text{qnext} \in \text{queuetokens} \mapsto \text{queuetokens} \land \]
\[ (0 < \text{qsize} \Rightarrow \text{card}(\text{qnext}) = \text{qsize} - 1) \land \]

empty and singleton queues have no next items

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

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

\[ \text{qsize} = \text{card}(\text{queuetokens}) \land \]

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

\[ \forall \, nn \cdot (\, nn \in \mathbb{N} \land nn \in 1 \ldots \text{qsize} - 1 \Rightarrow \]
\[ \text{qnext}^{nn} \in \text{queuetokens} \mapsto \text{queuetokens} \land \]

Refinement relation

the number of items in the linked version is the same as the length of the sequence
The QueueR machine VI

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

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} \left( \text{queue} \right) \Rightarrow \\
qnext^{pos - 1} \left( \text{qfirst} \right) = \text{queue} \left( pos \right) \) \]
The QueueR machine VII

ASSERTIONS

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

\[
\text{ran}(\text{queue}) = \bigcup pos . ( pos \in \text{dom}(\text{queue}) \mid \{ \text{queue}(pos) \}) \land \\
(1 < \text{qsize} \Rightarrow \text{queuetokens} = \text{ran}(\text{queue})) \land
\]

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

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

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

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

Specialisation of the refinement relation

\[
(0 < \text{qsize} \Rightarrow \text{qfirst} = \text{first}(\text{queue})) \land \\
(0 < \text{qsize} \Rightarrow \text{qlast} = \text{last}(\text{queue}))
\]
The QueueR machine VIII

INITIALISATION

\[
\text{queuetokens} \ , \ \text{queueitem} := \{\} \ , \ \{\} \ \| \\
\text{qsize} \ , \ \text{qnext} := 0 \ , \ \{\} \ \| \ \text{qfirst} \in \text{QUEUE} \ \| \ \text{qlast} \in \text{QUEUE}
\]

Note that \textit{qfirst} and \textit{qlast} always have a value, even when the list is empty.
The QueueR machine IX

**OPERATIONS**

\[
\text{queueid} \leftarrow \text{Enqueue}(\text{item}) \equiv
\]

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

\[
\text{queuetokens} := \text{queuetokens} \cup \{ \text{qid} \} \quad ||
\]

\[
\text{qlast} := \text{qid} \quad ||
\]

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

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

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

\[
\text{end} \quad ||
\]

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

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

\[
\text{end} ;
\]
The QueueR machine X

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

\begin{verbatim}
  begin
    item := queueitem( qfirst ) \parallel
    if 1 < qsize then
      qfirst := qnext( qfirst ) \parallel qnext := { qfirst } \lhd qnext
    end \parallel
    queueitem := { qfirst } \lhd queueitem \parallel
    queuetokens := queuetokens - { qfirst } \parallel
    qsize := qsize - 1
  end ;
\end{verbatim}
The QueueR machine XI

Unqueue ( qid ) ⊆
begin
  if 1 < qsize then
    if qid = qfirst then
      qfirst := qnext ( qid ) || qnext := { qid } ⊄ qnext
    elsif 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
DEFINITIONS qprev ⊆ qnext −1
END
The Design of the Implementation

Implementing queue tokens

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 ⊆ 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.
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 ⊆ 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.
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 ⊆ 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 $\text{prev}$ for free because $\text{qnext}$ was an injective function, so $\text{prev}$ by simply inverting $\text{qnext}$. In the implementation we have no such luxury. Here we implement $\text{prev}$ 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 $\text{qprev}$.
The QueueRI implementation

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 )
The QueueRI implementation II

**PROPERTIES**

\( \text{QUEUE} = 0 \ldots \text{maxqueue} \)

**INVARIANT**

Now the **refinement relation**

\[
\begin{align*}
\text{queuetokens} & \subseteq 1 \ldots \text{maxqueue} \land \\
\text{queuetokens} \cup \text{freetokens_sset} & = 1 \ldots \text{queuetokens_Nvar} \land \\
\text{queuetokens} \cap \text{freetokens_sset} & = \{\} \land \\
\text{queueitem_Vfnc} & = \text{queueitem} \land \\
\text{qnext} & \subseteq \text{qnext_Nfnc} \land \\
(0 < \text{qsize_Nvar} \implies \text{qfirst_Nvar} = \text{qfirst}) & \land \\
(0 < \text{qsize_Nvar} \implies \text{qlast_Nvar} = \text{qlast}) & \land \\
\text{queuetokens_Nvar} - \text{card}(\text{freetokens_sset}) & = \text{qsize} \land \\
\text{qsize_Nvar} & = \text{qsize}
\end{align*}
\]
The QueueRI implementation III

**ASSERTIONS**

\[
\text{freetokens\_sset} \subseteq 1 \ldots \text{queuetokens\_Nvar} \land
\]

\[
\text{queuetokens\_Nvar} - \text{card (freetokens\_sset)} = \text{card (queuetokens)} \land
\]

\[
(\text{freetokens\_sset = \{} \Rightarrow \text{queuetokens = 1} \ldots \text{queuetokens\_Nvar})
\]

**OPERATIONS**
The QueueRI implementation IV

\[
\begin{align*}
\text{queueid} & \leftarrow \textbf{Enqueue} (\text{item}) \equiv \\
\textbf{var} & \quad \text{bb, qid, lst in} \\
\quad & \quad \text{bb} \leftarrow \text{freetokens}_\text{EMP}_\text{SET} ; \\
\quad & \quad \text{if } \quad \text{bb} = \text{TRUE} \quad \text{then} \\
\quad & \quad \quad \text{queuetokens}_\text{INC}_\text{NVAR} ; \ qid \leftarrow \text{queuetokens}_\text{VAL}_\text{NVAR} \\
\quad & \quad \text{else} \\
\quad & \quad \quad \ qid \leftarrow \text{freetokens}_\text{ANY}_\text{SET} ; \ \text{freetokens}_\text{RMV}_\text{SET} (\ qid ) \end{align*}
\]

\[
\begin{align*}
\text{end} ; \\
\quad & \quad \text{bb} \leftarrow \text{qsize}_\text{GTR}_\text{NVAR} (\ 0 ) ; \\
\quad & \quad \text{if } \quad \text{bb} = \text{TRUE} \quad \text{then} \\
\quad & \quad \quad \text{lst} \leftarrow \text{qlast}_\text{VAL}_\text{NVAR} ; \ \text{qnext}_\text{STO}_\text{NFNC} (\ lst, qid ) \\
\quad & \quad \text{else} \\
\quad & \quad \quad \ \text{qfirst}_\text{STO}_\text{NVAR} (\ qid ) \\
\quad & \quad \text{end} ; \\
\quad & \quad \text{qlast}_\text{STO}_\text{NVAR} (\ qid ) ; \ \text{queueitem}_\text{STO}_\text{FNC} (\ qid, \text{item} ) ; \\
\quad & \quad \text{qsize}_\text{INC}_\text{NVAR} ; \ \text{queueid} := \ qid \\
\text{end} ;
\end{align*}
\]
The QueueRI implementation VI

\[
\begin{align*}
\text{item} & \leftarrow \text{Deque} \\
\text{var} & \quad bb, qfst, qnxt \text{ in} \\
qfst & \leftarrow \text{qfirst}_\text{VAL}_\text{NVAR} \; ; \text{item} \leftarrow \text{queueitem}_\text{VAL}_\text{FNC} \; (qfst) \; ; \\
b & \leftarrow \text{qsize}_\text{GTR}_\text{NVAR} \; (1) \; ; \\
\text{if} & \quad bb = \text{TRUE} \quad \text{then} \\
qnxt & \leftarrow \text{qnext}_\text{VAL}_\text{NFNC} \; (qfst) \; ; \\
qfst & \text{STO}_\text{NVAR} \; (qnxt) \\
\text{end} ; \\
\text{freetokens}_\text{ENT}_\text{SET} \; (qfst) \; ; \text{qsize}_\text{DEC}_\text{NVAR} \\
\text{end} ; 
\end{align*}
\]
The QueueRI implementation VII

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

\[ \text{var } \ bb, \ qnxt, \ qprv \ \text{in} \]

\[ bb \leftarrow \text{qsize}_\text{GTR}_\text{NVAR} (1); \]

\[ \text{if } bb = \text{TRUE} \text{ then} \]

\[ bb \leftarrow \text{qfirst}_\text{EQL}_\text{NVAR} (qid); \]

\[ \text{if } bb = \text{TRUE} \text{ then} \]

\[ qnxt \leftarrow \text{qnext}_\text{VAL}_\text{NFNC} (qid); \text{qfirst}_\text{STO}_\text{NVAR} (qnxt); \]

\[ \text{else} \]
The QueueRI implementation VIII

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.

\[
qprv \leftarrow q\text{first}_\text{VAL}_{\text{NVAR}} ; qnxt \leftarrow q\text{next}_\text{VAL}_{\text{NFNC}} (qprv) ;
\]

\[
\text{while } qnxt \neq qid \text{ do}
\]

\[
qprv := qnxt ; qnxt \leftarrow q\text{next}_\text{VAL}_{\text{NFNC}} (qprv)
\]

\text{invariant}

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

\[
\text{qnext}_{\text{Nfnc}} = \text{qnext} \land \text{q\first}_{\text{Nvar}} = \text{q\first} \land
\]

\[
\text{q\last}_{\text{Nvar}} = \text{q\last} \land \text{qsize}_{\text{Nvar}} = \text{qsize} \land
\]

\[
qprv \in \text{dom} ( q\text{next} ) \land qnxt = q\text{next} (qprv)
\]

\text{variant} \quad \text{queue}^{-1}(qid) \rightarrow \text{queue}^{-1}(qnxt)

\text{end} ;

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

\[
\neg(qnxt \neq qid) \land qnxt = q\text{next}(qprv) \ldots \equiv q\text{next}(qnxt) = qid \ldots
\]
The QueueRI implementation IX

\[ bb \leftarrow qlast\_EQL\_NVAR( qid ) ; \]
\begin{verbatim}
if \ bb = TRUE  then
    qlast\_STO\_NVAR( qprv ) ; qnext\_RMV\_NFNC( qprv )
else
    qnxt \leftarrow qnext\_VAL\_NFNC( qid ) ;
    qnext\_STO\_NFNC( qprv , qnxt )
end
end

qsize\_DEC\_NVAR ;
freetokens\_ENT\_SET( qid ) ;
queueitem\_RMV\_FNC( qid )
end
END
Notes on While Invariant . . .

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.

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.
Notes on While Invariant . . .

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.

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 main conjuncts of the invariant are:

\[ qprv \in \text{dom}(qnext) \land qnxt = qnext(qprv) \]

which describes the relationship between the changing variables.

The other parts of the invariant:

- \( queueitem_{Vfnc} = queueitem \land qnext_{Nfnc} = qnext \land qfirst_{Nvar} = qfirst \land qlast_{Nvar} = qlast \land qsize_{Nvar} = qsize \)

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

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 $queue^{-1}$ to map queuetokens to position in the sequence and

$$queue^{-1}(qid) - queue^{-1}(qnxt)$$

is a decreasing expression bounded below by 0.
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 $\text{queue}^{-1}$ to map queuetokens to position in the sequence and

$$\text{queue}^{-1}(qid) - \text{queue}^{-1}(qnxt)$$

is a decreasing expression bounded below by 0.
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 $queue^{-1}$ to map queuetokens to position in the sequence and

$$queue^{-1}(qid) - queue^{-1}(qnxt)$$

is a decreasing expression bounded below by 0.
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.
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
MACHINE QueueAPI (maxqueue)
CONSTRAINTS maxqueue ∈ N₁
SEES Response_ctx, Queue_ctx, Item_ctx
INCLUDES Queue (ITEM, maxqueue)

OPERATIONS
status, queueid ← EnqueueAPI(item) ≡

pre item ∈ ITEM then

select size(queue) ≠ maxqueue then

status := OK || queueid ← Enqueue(item)

else status := QueueFull || queueid ∈ QUEUE

end

end ;
\[
\text{status}, \text{item} \leftarrow \text{DequeueAPI} \equiv \\
\text{select } 1 \leq \text{size} (\text{queue}) \text{ then} \\
\quad \text{status} := \text{OK} \quad \| \quad \text{item} \leftarrow \text{Dequeue} \\
\text{else} \quad \text{status} := \text{QueueEmpty} \quad \| \quad \text{item} \in \text{ITEM} \\
\text{end} ;
\]
status ← UnqueueAPI ( qid ) ≜
  pre qid ∈ QUEUE then
    select 0 = size ( queue ) then
      status := QueueEmpty
    when qid ∉ queuetokens then
      status := NotInQueue
    else status := OK || Unqueue ( qid )
  end
end
END
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.
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.
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*. 
Implementation strategies

There are principally two implementation strategies:

1. Simply implement the whole machine, in this case \textit{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 \textit{QueueAPI} by essentially importing the \textit{Queue} machine; followed by implementation of \textit{Queue}.

However, due to complete hiding, it will not be possible to evaluate the guards. To facilitate this a machine \textit{QueuePreAPI}, which augments the \textit{Queue} operations, by auxiliary operations that will be required by the implementation of \textit{QueueAPI}.
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*. 
MACHINE QueuePreAPI ( maxqueue )
CONSTRANTS maxqueue ∈ N
SEES Bool_TYPE, Queue_ctx, Item_ctx
EXTENDS Queue ( ITEM, maxqueue )

OPERATIONS
anyqueue ← anyQUEUE ≜ anyqueue :∈ QUEUE ;
anyitem ← anyITEM ≜ anyitem :∈ ITEM ;
ok ← isQueueEmpty ≜
    ok := bool ( size ( queue ) = 0 ) ;
ok ← isQueueFull ≜
    ok := bool ( size ( queue ) = maxqueue ) ;
ok ← isItemInQueue ( qid ) ≜
    pre qid ∈ QUEUE then
        ok := bool ( qid ∈ queuetokens )
    end
END
Implementing QueueAPI

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

The purpose of the auxiliary operations will be clear from an inspection of the code for the guards of the robust operations.
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  QueueAPIII
REFINES  QueueAPI
SEES  Bool_TYPE, Response_ctx, Queue_ctx, Item_ctx
IMPORTS  QueuePreAPI ( maxqueue )

OPERATIONS
$$\text{status} \ , \ \text{queueid} \leftarrow \text{EnqueueAPI} \left( \text{item} \right) \triangleq$$

\[
\begin{align*}
\text{var} \ & \ bb \ \text{in} \\
bb & \leftarrow \text{isQueueFull} \\
\text{if} \ & \ bb = \text{FALSE} \ \text{then} \\
\text{status} & := \text{OK} \ ; \ \text{queueid} \leftarrow \text{Enqueue} \left( \text{item} \right) \\
\text{else} \\
\text{status} & := \text{QueueFull} \ ; \ \text{queueid} \leftarrow \text{anyQUEUE} \\
\text{end} \\
\text{end} ;
\end{align*}
\]
\begin{verbatim}
status, item ← DequeueAPI ≜
  var bb in
  bb ← isQueueEmpty;
  if bb = FALSE then
    status := OK; item ← Dequeue
  else
    status := QueueEmpty; item ← anyITEM
  end
end;
\end{verbatim}
status ← UnqueueAPI ( qid ) ≜

var bb in

bb ← isQueueEmpty ;
status := QueueEmpty ;
if bb = FALSE then
    status := NotInQueue ;
    bb ← isItemInQueue ( qid ) ;
if bb = TRUE then
    status := OK ; Unqueue ( qid )
end
end
end

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

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

In turn, the refinement will be implemented following QueueRI.

Since QueuePreAPIR and QueuePreAPIRI are so similar to QueueR and QueueRI they are moved to the appendix.
Refining QueuePreAPI

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

QueuePreAPI\textsubscript{RI} follows Queue\textsubscript{R}, with the addition of the auxiliary operations.

In turn, the refinement will be implemented following Queue\textsubscript{RI}.

Since QueuePreAPI\textsubscript{RI} and QueuePreAPI\textsubscript{RI}\textsubscript{RI} are so similar to Queue\textsubscript{RI} and Queue\textsubscript{RI} they are moved to the appendix.
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 QueuePreAPIRI are so similar to QueueR and QueueRI they are moved to the appendix.
Refining QueuePreAPI

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

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

In turn, the refinement will be implemented following QueueRI.

Since QueuePreAPIR and QueuePreAPIRI are so similar to QueueR and QueueRI they are moved to the appendix.
REFINEMENT QueuePreAPIR
REFINES QueuePreAPIR
SEES Bool_TYPE, Queue_ctx, Item_ctx
VARIABLES
    queuetokens, queueitem,
    qfirst, qlast, qnext, qsize
IN Variant

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

\[
\text{queuetokens} \subseteq \text{QUEUE} \land \text{queuetokens} = \text{ran (queue)} \land
\]

queueitem links an item’s token to the item

\[
\text{queueitem} \in \text{queuetokens} \rightarrow \text{ITEM} \land
\]

\[
\text{qfirst} \in \text{QUEUE} \land \text{qlast} \in \text{QUEUE} \land
\]

if the queue length is greater than 1 then qfirst is one of the current tokens

\[
(0 < qsize \Rightarrow \text{qfirst} \in \text{queuetokens}) \land
\]

if the queue length is greater than 1 then qlast is one of the current tokens

\[
(0 < qsize \Rightarrow \text{qlast} \in \text{queuetokens}) \land
\]
QueuePreAPIR III

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

\[ qnext \in QUEUE \mapsto QUEUE \land \]

there are no loops

\[ qnext \cap \text{id}(QUEUE) = \{\} \land \]
\[ \bigcup nn \cdot (nn \in \mathbb{N} \mid qnext^{nn}) \cap \text{id}(QUEUE) = \{\} \land \]
\[ qsize \in \mathbb{N} \land \]

All of the tokens in the queue are in queuetokens

\[ (1 < qsize \Rightarrow \text{queuetokens} = qnext^* [\{qfirst\}] ) \land \]

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

\[ (1 < qsize \Rightarrow \forall nn \cdot (nn \in \mathbb{N} \land nn \in 0 .. qsize - 2 \Rightarrow qnext^{nn}(qfirst) \in \text{dom}(qnext)) ) \land \]

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

\[(1 < qsize \Rightarrow \forall \, \text{nn} \cdot (\, \text{nn} \in \mathbb{N} \land \text{nn} \in 1 \ldots qsize - 1 \Rightarrow \, qnext^{\text{nn}} (\, qfirst ) \in \text{ran} (\, qnext ) ) ) \land \]

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

\[qfirst \notin \text{ran} (\, qnext ) \land \]

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

\[(qsize = 1 \Rightarrow qlast = qfirst ) \land \]

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
QueuePreAPIR V

\[ qnext \in \text{queuetokens} \mapsto \text{queuetokens} \land \\
( 0 < qsize \Rightarrow \text{card ( } qnext \text{ ) } = qsize - 1 ) \land \\
\text{empty and singleton queues have no } next \text{ items} \\
( qsize \leq 1 \Rightarrow qnext = \{\} ) \land \\
\text{the number of queue tokens is the same as the size of the queue} \\
qsize = \text{card ( queuetokens )} \land \\
\text{composing the qnext function with itself any number of times produces an injective function} \\
\forall nn . ( nn \in \mathbb{N} \land nn \in 1 \ldots qsize - 1 \Rightarrow \\
qnext^{nn} \in \text{queuetokens} \mapsto \text{queuetokens} ) \land \\
\text{Refinement relation} \\
\text{the number of items in the linked version is the same as the length of the sequence} \]
\[ qsize = \text{size}(\text{queue}) \wedge \]

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

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

\( ( 1 < q\text{size} \Rightarrow q\text{first} \in \text{dom} ( q\text{next} ) ) \land \)

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

\( ( 1 < q\text{size} \Rightarrow q\text{last} \in \text{ran} ( q\text{next} ) ) \land \)

Specialisation of the refinement relation

\( ( 0 < q\text{size} \Rightarrow q\text{first} = \text{first} ( \text{queue} ) ) \land \)
\( ( 0 < q\text{size} \Rightarrow q\text{last} = \text{last} ( \text{queue} ) ) \)
INITIALISATION

\[
\text{queuetokens} , \text{queueitem} := \{\} , \{\} \parallel \\
\text{qsize} , \text{qnext} := 0 , \{\} \parallel \text{qfirst} \in \text{QUEUE} \parallel \text{qlast} \in \text{QUEUE}
\]

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

\[ queueid \leftarrow \text{Enqueue} (\text{item}) \equiv \]

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

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

\[ qlast := qid \]

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

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

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

\[ \text{end } \]

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

\[ qsize := qsize + 1 \]

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

\[ \text{end} ; \]
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 ;
QueuePreAPIR XI

Unqueue(\( qid \)) ⊆
begin
    if \( 1 < qsize \) then
        if \( qid = qfirst \) then
            \( qfirst := qnext(qid) \) || \( qnext := \{ qid \} \ll qnext \)
        elsif \( qid = qlast \) then
            \( qlast := qprev(qid) \) || \( qnext := qnext \gg \{ qid \} \)
        else
            \( qnext := \{ qid \} \ll (qnext \ll \{ qprev(qid) \rightarrow qnext(qid) \}) \)
        end
    end ||
    \( qsize := qsize - 1 \) ||
    \( queuetokens := queuetokens - \{ qid \} \) ||
    \( queueitem := \{ qid \} \ll queueitem \)
end;
\( \text{anyqueue} \leftarrow \text{anyQUEUE} \triangleq \text{anyqueue} : \in \text{QUEUE} ; \)
\( \text{anyitem} \leftarrow \text{anyITEM} \triangleq \text{anyitem} : \in \text{ITEM} ; \)
\( \text{ok} \leftarrow \text{isQueueEmpty} \triangleq \text{ok} := \text{bool} ( qsize = 0 ) ; \)
QueuePreAPIR XII

\[ \text{ok} \leftarrow \text{isQueueFull} \triangleq \text{ok} := \text{bool} \ (qsize = \text{maxqueue}) ; \]
\[ \text{ok} \leftarrow \text{isItemInQueue} \ (qid) \triangleq \text{ok} := \text{bool} \ (qid \in \text{queuetokens}) \]

DEFINITIONS \[ qprev \triangleq qnext^{-1} \]
END
IMPLEMENTATION QueuePreAPIRI
REFINES QueuePreAPIR
SEES Bool_TYPE, Queue_ctx, Item_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 ∧
    ITEM = 0 .. 100
IN Variant

Now the refinement relation

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

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

OPERATIONS
QueuePreAPIR IV

\[ \text{queueid} \leftarrow \text{Enqueue} \ ( \text{item} ) \triangleq \]

\begin{verbatim}
var bb, qid, lst in
    bb ← freetokens_EMP_SET ;
    if \( bb = \text{TRUE} \) 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 ;
\end{verbatim}
item ← Dequeue

var bb, qfst, qnxt in
qfst ← qfirst_VAL_NVAR ; item ← queueitem_VAL_FNC ( qfst ) ;
bb ← qsize_GTR_NVAR ( 1 ) ;
if bb = TRUE then
qnxt ← qnext_VAL_NFNC ( qfst ) ;
qfirst_STO_NVAR ( qnxt )
end ;
freetokens_ENT_SET ( qfst ) ; qsize_DEC_NVAR
end ;
Unqueue ( qid ) ≜

var bb, qnxt, qprv in

bb ← qsize\_GTR\_NVAR ( 1 ) ;
if bb = TRUE then
  bb ← qfirst\_EQL\_NVAR ( qid ) ;
if bb = TRUE then
  qnxt ← qnext\_VAL\_NFNC ( qid ) ; qfirst\_STO\_NVAR ( qnxt )
else
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.

\[
q_{prv} \leftarrow q_{first}_{\text{VAL\_NVAR}} \; ; \; q_{nxt} \leftarrow q_{next}_{\text{VAL\_NFNC}} ( q_{prv} ) ;
\]

\textbf{while } q_{nxt} \neq q_{id} \textbf{ do}

\[
q_{prv} := q_{nxt} \; ; \; q_{nxt} \leftarrow q_{next}_{\text{VAL\_NFNC}} ( q_{prv} )
\]

\textbf{invariant}

\[
\text{queueitem} \_\text{Vfnc} = \text{queueitem} \land \\
q_{next} \_\text{Nfnc} = q_{next} \land q_{first} \_\text{Nvar} = q_{first} \land \\
q_{last} \_\text{Nvar} = q_{last} \land q_{size} \_\text{Nvar} = q_{size} \land \\
q_{prv} \in \text{dom} ( q_{next} ) \land q_{nxt} = q_{next} ( q_{prv} )
\]

\textbf{variant} \quad q_{size} = q_{size}^{-1} ( q_{id} )

\textbf{end} ;

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

\[
\neg ( q_{nxt} \neq q_{id} ) \land q_{nxt} = q_{next}(q_{prv}) \ldots \equiv q_{next}(q_{nxt}) = q_{id} \ldots
\]
\begin{verbatim}
bb \leftarrow qlast\_EQL\_NVAR ( qid ) ;
if \ bb = TRUE \ then
    qlast\_STO\_NVAR ( qprv ) ; qnext\_RMV\_NFNC ( qprv )
else
    qnxt \leftarrow qnext\_VAL\_NFNC ( qid ) ;
    qnext\_STO\_NFNC ( qprv , qnxt )
end
end
end ;
qsize\_DEC\_NVAR ;
freetokens\_ENT\_SET ( qid ) ;
queueitem\_RMV\_FNC ( qid )
end ;
\end{verbatim}
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

END
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.
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**\_\_\_Nfc 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.
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.
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.
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.
MACHINE  queuetokens_Nvar ( maxint )
CONSTRAINTS  maxint ≤ 2147483646
SEES  Bool_TYPE
VARIABLES  queuetokens_Nvar
INVARIANT  queuetokens_Nvar ∈ 0 .. maxint
INITIALISATION  queuetokens_Nvar := 0

OPERATIONS
  vv ←− queuetokens.VAL_NVAR  ≡
    begin
      vv := queuetokens_Nvar
    end ;
  queuetokens.STO_NVAR ( vv ) ≡
    pre
      vv ∈ 0 .. maxint
    then
\texttt{queuetokens\_Nvar.mch II}

\begin{verbatim}
\textit{queuetokens\_Nvar} := vv
end ;
\textit{uu} ← \textit{queuetokens\_MIN\_NVAR} ( vv ) ≡
\textit{pre}
    \begin{itemize}
    \item \textit{vv} ∈ 0 . . maxint
\end{itemize}
\textit{then}
    \begin{itemize}
    \item \textit{uu} := \textit{min} ( \{ \textit{queuetokens\_Nvar} , \textit{vv} \} )
\end{itemize}
\textit{end} ;
\textit{uu} ← \textit{queuetokens\_MAX\_NVAR} ( vv ) ≡
\textit{pre}
    \begin{itemize}
    \item \textit{vv} ∈ 0 . . maxint
\end{itemize}
\textit{then}
    \begin{itemize}
    \item \textit{uu} := \textit{max} ( \{ \textit{queuetokens\_Nvar} , \textit{vv} \} )
\end{itemize}
\textit{end} ;
\textit{bb} ← \textit{queuetokens\_PRE\_INC\_NVAR} ≡
\begin{itemize}
\item \textit{begin}
    \begin{itemize}
    \item \textit{bb} := \textit{bool} ( \textit{queuetokens\_Nvar} < \textit{maxint} )
    \end{itemize}
\end{itemize}
\textit{end} ;
\end{verbatim}
queue_tokens_Nvar.mch III

queue_tokens_INC_NVAR \triangleq
  \textbf{pre}
  \begin{align*}
  &\text{queue_tokens}_Nvar + 1 \in 0 \ldots \text{maxint} \\
  \textbf{then}
  &\text{queue_tokens}_Nvar := \text{queue_tokens}_Nvar + 1
  \end{align*}
\textbf{end ;}

\text{bb} \leftarrow \text{queue_tokens}_P\text{RE\_DEC\_NVAR} \triangleq
  \begin{align*}
  \textbf{begin}
  &\text{bb} := \text{bool} (\text{queue_tokens}_Nvar > 0) \\
  \textbf{end ;}
\end{align*}

\text{queue_tokens}_DE\text{C\_NVAR} \triangleq
  \textbf{pre}
  \begin{align*}
  &\text{queue_tokens}_Nvar \in 1 \ldots \text{maxint} \\
  \textbf{then}
  &\text{queue_tokens}_Nvar := \text{queue_tokens}_Nvar - 1
  \end{align*}
\textbf{end ;}

\text{bb} \leftarrow \text{queue_tokens}_P\text{RE\_ADD\_NVAR} (vv) \triangleq
  \textbf{pre}
**queuetokens_Nvar.mch IV**

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

then

\[ \text{bb} := \text{bool} \left( \text{queuetokens}_\text{Nvar} + \text{vv} \leq \text{maxint} \right) \]

end ;

\text{queuetokens}_\text{ADD}_\text{NVAR} \left( \text{vv} \right) \equiv

pre

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

\[ \text{queuetokens}_\text{Nvar} + \text{vv} \leq \text{maxint} \]

then

\[ \text{queuetokens}_\text{Nvar} := \text{queuetokens}_\text{Nvar} + \text{vv} \]

end ;

\[ \text{bb} \leftarrow \text{queuetokens}_\text{PRE}_\text{MUL}_\text{NVAR} \left( \text{vv} \right) \equiv \]

pre

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

then

\[ \text{bb} := \text{bool} \left( \text{queuetokens}_\text{Nvar} \times \text{vv} \leq \text{maxint} \right) \]

end ;

\text{queuetokens}_\text{MUL}_\text{NVAR} \left( \text{vv} \right) \equiv
pre
\[ \begin{align*}
vv & \in 0 \ldots \text{maxint} \land \\
\text{queuetokens\_Nvar} \times vv & \leq \text{maxint}
\end{align*} \]
then
\[ \begin{align*}
\text{queuetokens\_Nvar} & := \text{queuetokens\_Nvar} \times vv
\end{align*} \]
end ;
\[ bb \leftarrow \text{queuetokens\_PRE\_SUB\_NVAR} (vv) \triangleq \]
pre
\[ \begin{align*}
vv & \in 0 \ldots \text{maxint}
\end{align*} \]
then
\[ \begin{align*}
bb & := \text{bool} (\text{queuetokens\_Nvar} - vv \geq 0)
\end{align*} \]
end ;
\[ \text{queuetokens\_SUB\_NVAR} (vv) \triangleq \]
pre
\[ \begin{align*}
vv & \in 0 \ldots \text{maxint} \land \\
\text{queuetokens\_Nvar} - vv & \geq 0
\end{align*} \]
then
\[ \begin{align*}
\text{queuetokens\_Nvar} & := \text{queuetokens\_Nvar} - vv
\end{align*} \]


```
queuetokens_Nvar.mch VI

end ;
bb ←→ queuetokens_PRE_DIV_NVAR ( vv ) ≜

pre
  vv ∈ 0 .. maxint
then
  bb := bool ( vv > 0 )
end ;
queuetokens_DIV_NVAR ( vv ) ≜

pre
  vv ∈ 1 .. maxint
then
  queuetokens_Nvar := queuetokens_Nvar / vv
end ;
bb ←→ queuetokens_PRE_MOD_NVAR ( vv ) ≜

pre
  vv ∈ 0 .. maxint
then
  bb := bool ( vv > 0 )
```
end;
queuetokens\_MOD\_NVAR ( \( vv \) ) \( \equiv \)
pre
\( vv \in 1..\text{maxint} \)
then
\[ \text{queuetokens\_Nvar} := \text{queuetokens\_Nvar} - vv \times (\text{queuetokens\_Nvar} / vv) \]
end;
\( bb \leftarrow \text{queuetokens\_EQL\_NVAR} ( \( vv \) ) \( \equiv \)
pre
\( vv \in 0..\text{maxint} \)
then
\( bb := \text{bool} ( \text{queuetokens\_Nvar} = vv ) \)
end;
\( bb \leftarrow \text{queuetokens\_NEQ\_NVAR} ( \( vv \) ) \( \equiv \)
pre
\( vv \in 0..\text{maxint} \)
then
\( bb := \text{bool} ( \text{queuetokens\_Nvar} \neq vv ) \)
end;

\( bb \leftarrow \text{queuetokens}\_\text{GTR}\_\text{NVAR} ( vv ) \)
\begin{align*}
\text{pre} \quad & \quad vv \in 0 \ldots \text{maxint} \\
\text{then} \quad & \quad bb := \text{bool} ( \text{queuetokens}\_\text{Nvar} > vv )
\end{align*}
end;

\( bb \leftarrow \text{queuetokens}\_\text{GEQ}\_\text{NVAR} ( vv ) \)
\begin{align*}
\text{pre} \quad & \quad vv \in 0 \ldots \text{maxint} \\
\text{then} \quad & \quad bb := \text{bool} ( \text{queuetokens}\_\text{Nvar} \geq vv )
\end{align*}
end;

\( bb \leftarrow \text{queuetokens}\_\text{SMR}\_\text{NVAR} ( vv ) \)
\begin{align*}
\text{pre} \quad & \quad vv \in 0 \ldots \text{maxint} \\
\text{then} \quad & \quad bb := \text{bool} ( \text{queuetokens}\_\text{Nvar} < vv )
\end{align*}
\[ \text{end ;} \]

\[ bb \leftarrow \text{queuetokens\_LEQ\_NVAR} ( vv ) \triangleq \]

\[ \text{pre} \]

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

\[ \text{then} \]

\[ bb := \text{bool} ( \text{queuetokens\_Nvar} \leq vv ) \]

\[ \text{end ;} \]

\[ \text{queuetokens\_SAV\_NVAR} \triangleq \]

\[ \text{begin} \]

\[ \text{skip} \]

\[ \text{end ;} \]

\[ \text{queuetokens\_RST\_NVAR} \triangleq \]

\[ \text{begin} \]

\[ \text{queuetokens\_Nvar} \in 0 \ldots \text{maxint} \]

\[ \text{end ;} \]

\[ \text{queuetokens\_SAVN\_NVAR} \triangleq \]

\[ \text{begin} \]

\[ \text{skip} \]
end;
queuetokens_RSTN_NVAR ≜ begin
    queuetokens_Nvar :∈ 0 .. maxint
end
END
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_IDX_SET ( ii ) ≜
   pre
freetokens_set.mch II

\[ ii \in 1 \ldots \text{maxcrd} \]

then

\[ bb \leftarrow \text{bool} \left( ii \in 1 \ldots \text{card} \left( \text{freetokens}\_\text{sset} \right) \right) \]

end ;

\[ nn \leftarrow \text{freetokens}\_\text{CRD}\_\text{SET} \ni \]

begin

\[ nn \leftarrow \text{card} \left( \text{freetokens}\_\text{sset} \right) \]

end ;

\[ vv \leftarrow \text{freetokens}\_\text{VAL}\_\text{SET} \left( ii \right) \ni \]

pre

\[ ii \in 1 \ldots \text{card} \left( \text{freetokens}\_\text{sset} \right) \]

then

\[ vv \leftarrow \text{freetokens}\_\text{ordn} \left( ii \right) \]

end ;

\[ vv \leftarrow \text{freetokens}\_\text{ANY}\_\text{SET} \ni \]

pre

\[ \neg \left( \text{freetokens}\_\text{sset} = \{\} \right) \]

then
freetokens_set.mch III

\[ vv \in \text{freetokens\_sset} \]

end;

freetokens_CLR_SET \equiv

begin

freetokens\_sset := \{ \} \parallel
freetokens\_ordn := []

end;

freetokens_ENT_SET ( vv ) \equiv

pre

vv \in VALUE \land

card ( freetokens\_sset ) < maxcrd

then

freetokens\_sset := freetokens\_sset \cup \{ vv \} \parallel
freetokens\_ordn := \text{perm} ( freetokens\_sset \cup \{ vv \} )

end;

freetokens_RMV_SET ( vv ) \equiv

pre

vv \in VALUE
then
  \( \text{freetokens\_sset} := \text{freetokens\_sset} - \{ \text{vv} \} \ || \)
  \( \text{freetokens\_ordn} \in \text{perm} ( \text{freetokens\_sset} - \{ \text{vv} \} ) \)
end ;
bb \leftarrow \text{freetokens\_MBR\_SET} ( \text{vv} ) \equiv
  \text{pre}
  \( \text{vv} \in \text{VALUE} \)
then
  bb := \text{bool} ( \text{vv} \in \text{freetokens\_sset} )
end ;
bb \leftarrow \text{freetokens\_EMP\_SET} \equiv
  \text{begin}
  bb := \text{bool} ( \text{freetokens\_sset} = \{\} )
  \text{end} ;
\text{freetokens\_SAV\_SET} \equiv \text{skip} ;
\text{freetokens\_RST\_SET} \equiv
  \text{any } rset , rseq \text{ where}
  rset \in \mathbb{F} ( \text{VALUE} ) \wedge
freetokens_set.mch

\[
\begin{align*}
\text{rseq} & \in \text{perm ( rset ) } \land \\
\text{card ( rset )} & \leq \text{maxcrd} \\
\text{then} & \\
\text{freetokens_sset} & := \text{rset} \parallel \\
\text{freetokens_ordn} & := \text{rseq} \\
\end{align*}
\]
end ;
\[
\begin{align*}
\text{freetokens_SAVN_SET} & \triangleq \text{skip} ; \\
\text{freetokens_RSTN_SET} & \triangleq \\
\text{any} & \text{ rset, rseq where} \\
\text{rset} & \in \mathbb{F} ( \text{VALUE} ) \land \\
\text{rseq} & \in \text{perm ( rset ) } \land \\
\text{card ( rset )} & \leq \text{maxcrd} \\
\text{then} & \\
\text{freetokens_sset} & := \text{rset} \parallel \\
\text{freetokens_ordn} & := \text{rseq} \\
\end{align*}
\]
end
END
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 := {} 

OPERATIONS

bb ← qnext_TST_FLD_NFNC ( ff ) =

pre

ff ∈ N

then

bb := bool ( ff ∈ 1 .. maxfld )

end ;
\[ bb \leftarrow \text{qnext\_DEF\_NFNC}(ff) \equiv \]

\text{pre}
\[
ff \in 1 .. \text{maxfld}
\]

\text{then}
\[
bb := \text{bool}(ff \in \text{dom}(\text{qnext\_Nfnc}))
\]

\text{end} ;

\[ bb, dd \leftarrow \text{qnext\_FREE\_NFNC} \equiv \]

\text{if} \ (1 .. \text{maxfld}) - \text{dom}(\text{qnext\_Nfnc}) \neq \{\} \text{ then}
\[
bb := \text{TRUE} \ ||
\]
\[
dd :\in (1 .. \text{maxfld}) - \text{dom}(\text{qnext\_Nfnc})
\]

\text{else}
\[
bb := \text{FALSE} \ ||
\]
\[
dd :\in 1 .. \text{maxfld}
\]

\text{end} ;

\[ \text{qnext\_STO\_NFNC}(ff, vv) \equiv \]

\text{pre}
\[
ff \in 1 .. \text{maxfld} \land
\]
\[
vv \in 0 .. \text{maxint}
\]
then
    \( qnext_{Nfnc} ( ff ) := vv \)
end;
qnext\_RMV\_NFNC ( ff ) \equiv
pre
    ff \in \text{dom} ( qnext\_Nfnc )
then
    qnext\_Nfnc := \{ ff \} \leftarrow qnext\_Nfnc
end;
qnext\_ADD\_NFNC ( ff , vv ) \equiv
pre
    vv \in 0 \ldots \text{maxint} \land
    ff \in \text{dom} ( qnext\_Nfnc ) \land
    qnext\_Nfnc ( ff ) + vv \leq \text{maxint}
then
    qnext\_Nfnc ( ff ) := qnext\_Nfnc ( ff ) + vv
end;
qnext\_MUL\_NFNC ( ff , vv ) \equiv
\begin{verbatim}
qnext_Nfnc.mch IV

pre
  \begin{align*}
  vv & \in 0 \ldots \text{maxint} \land \\
  ff & \in \text{dom} \left( qnext_Nfnc \right) \land \\
  qnext_Nfnc \left( ff \right) \times vv & \leq \text{maxint}
  \end{align*}
then
  qnext_Nfnc \left( ff \right) \coloneqq qnext_Nfnc \left( ff \right) \times vv
end;

qnext_SUB_NFNC \left( ff, vv \right) \triangleq
pre
  \begin{align*}
  vv & \in 0 \ldots \text{maxint} \land \\
  ff & \in \text{dom} \left( qnext_Nfnc \right) \land \\
  qnext_Nfnc \left( ff \right) & \geq vv
  \end{align*}
then
  qnext_Nfnc \left( ff \right) \coloneqq qnext_Nfnc \left( ff \right) - vv
end;

qnext_DIV_NFNC \left( ff, vv \right) \triangleq
pre
  \begin{align*}
  vv & \in 1 \ldots \text{maxint} \land
  \end{align*}
\end{verbatim}
\[
ff \in \text{dom} ( \text{qnext}_Nfnc )
\]
then
\[
\text{qnext}_Nfnc ( \ff ) := \text{qnext}_Nfnc ( \ff ) / \vv
\]
end ;
\[
\text{qnext}_\text{MOD}_N\text{FNC} ( \ff , \vv ) \triangleq
\]
pre
\[
\vv \in 1 \ldots \text{maxint} \land
\ff \in \text{dom} ( \text{qnext}_Nfnc )
\]
then
\[
\text{qnext}_Nfnc ( \ff ) := \text{qnext}_Nfnc ( \ff ) - \vv \times ( \text{qnext}_Nfnc ( \ff ) / \vv )
\]
end ;
\[
\vv \leftarrow \text{qnext}_\text{VAL}_N\text{FNC} ( \ff ) \triangleq
\]
pre
\[
\ff \in \text{dom} ( \text{qnext}_Nfnc )
\]
then
\[
\vv := \text{qnext}_Nfnc ( \ff )
\]
end ;
\[
\bb \leftarrow \text{qnext}_\text{EQL}_N\text{FNC} ( \ff , \vv ) \triangleq
\]
qnext_Nfnc.mch VI

pre

\[ vv \in 0 \ldots \text{maxint} \land \]
\[ ff \in \text{dom} ( qnext\_Nfnc ) \]

then

\[ bb := \text{bool} \ ( qnext\_Nfnc \ ( ff ) = vv ) \]

end ;

\[ bb \leftarrow qnext\_NEQ\_NFNC \ ( ff , vv ) \]

pre

\[ vv \in 0 \ldots \text{maxint} \land \]
\[ ff \in \text{dom} ( qnext\_Nfnc ) \]

then

\[ bb := \text{bool} \ ( qnext\_Nfnc \ ( ff ) \neq vv ) \]

end ;

\[ bb \leftarrow qnext\_GTR\_NFNC \ ( ff , vv ) \]

pre

\[ vv \in 0 \ldots \text{maxint} \land \]
\[ ff \in \text{dom} ( qnext\_Nfnc ) \]

then
\( \text{bb} := \text{bool} \left( \text{qnext\_Nfnc} \left( \text{ff} \right) > \text{vv} \right) \)

\( \text{end} ; \)
\( \text{bb} \leftarrow \text{qnext\_GEQ\_NFNC} \left( \text{ff} , \text{vv} \right) \equiv \)

\( \text{pre} \)
\( \text{vv} \in 0 .. \text{maxint} \land \)
\( \text{ff} \in \text{dom} \left( \text{qnext\_Nfnc} \right) \)

\( \text{then} \)
\( \text{bb} := \text{bool} \left( \text{qnext\_Nfnc} \left( \text{ff} \right) \geq \text{vv} \right) \)

\( \text{end} ; \)
\( \text{bb} \leftarrow \text{qnext\_SMR\_NFNC} \left( \text{ff} , \text{vv} \right) \equiv \)

\( \text{pre} \)
\( \text{vv} \in 0 .. \text{maxint} \land \)
\( \text{ff} \in \text{dom} \left( \text{qnext\_Nfnc} \right) \)

\( \text{then} \)
\( \text{bb} := \text{bool} \left( \text{qnext\_Nfnc} \left( \text{ff} \right) < \text{vv} \right) \)

\( \text{end} ; \)
\( \text{bb} \leftarrow \text{qnext\_LEQ\_NFNC} \left( \text{ff} , \text{vv} \right) \equiv \)

\( \text{pre} \)
\begin{verbatim}

  vv ∈ 0 . . maxint ∧ 
  ff ∈ dom ( qnext_Nfnc )
  then
    bb := bool ( qnext_Nfnc ( ff ) ≤ vv )
  end ;
qnext_SAV_NFNC ≜ begin skip end ;
qnext_RST_NFNC ≜
  begin  qnext_Nfnc :∈ 1 . . maxfld → 0 . . maxint end ;
qnext_SAVN_NFNC ≜ begin skip end ;
qnext_RSTN_NFNC ≜
  begin  qnext_Nfnc :∈ 1 . . maxfld → 0 . . maxint end

END
\end{verbatim}