

This module defines the abstract program *IPOFifo_pqs* described in Section 7.6.3 of “A Science of Concurrent Programs” by *Leslie Lamport*. This program is obtained by adding auxiliary variables to program *IPOFifo* defined in module *IPOFifo*. The module also defines the refinement mapping under which *IPOFifo_pqs* implements program *IFifo* of module *IFifo*.

Here are the three auxiliary variables, in the order in which they are added.

Prophecy Variable p

The prophecy sequence variable p is explained in Section 7.6.3.1 of the book. It predicts the order in which the elements of $elts$ will be dequeued—that is, the sequence of elements that will be dequeued by an *EndPODeq* action. A prediction is appended to p when an element is added to $elts$. Note that p is a sequence of data, identifier pairs not just a sequence of data items. It predicts the element that the *EndPODeq* action removes from $elts$, not just the data value the Dequeue operation returns.

Like most prophecy variables, most of the the predictions p can make can’t possibly be right. Sometimes their impossibility can be determined even when the prophecy is made. Logically, that makes no difference. An impossible prediction just prevents dequeue operations from performing their *EndPODeq* actions. Such prophecies would be ruled out by a fairness requirement on the *EndPODeq* action.

History Variable $qBar$

As described in Section 7.6.3.2, $qBar$ is a sequence of datums (elements of $Data \times Ids$ that equals $pg \circ eb$, where pg is the sequence of datums that can be removed from the set $elts$ by dequeue operations even if no further enqueue operations are initiated, and eb is a (hopefully empty) sequence of datums in $elts$ that can never be removed from $elts$ by any dequeue operation.

Here is a description of how the history variable $qBar$ is added that I wrote before writing the book. It’s more detailed than the one in the book, and I’m leaving it here because it might help some readers to understand its definition.

The value of $qBar$ is a sequence of elements that is approximately the value that the refinement mapping assigns to the variable *queue* of *Fifo* (except it contains not just the *Data* values that are in *queue* but also the identifier associated with that data value). It is not exactly the value that implements *queue* because elements aren’t added to it at the right time. (Stuttering variables will be added later to allow defining a refinement mapping that adds the elements to *queue* at the right time.)

To see how to construct $qBar$, let’s pretend we know the entire infinite sequence PP of predictions that will be made by the prophecy variable P . Call PP correct iff it permits a behavior with infinitely many Dequeue operations (and hence infinitely many Enqueue operations). If PP is correct, then p must always be a subsequence of PP of length equal to the number of elements in $elts$. We construct $qBar$ so that, in this case, it is the largest prefix of p all of whose elements are already in $elts$. For example, suppose PP equals $p1, p2, \dots$. Initially $elts = \{\}$ and $p = qBar = \langle \rangle$. Consider the following sequence of *BeginPOEnq* and *EndPODeq* actions, with the values they add or remove from $elts$, and the values of $elts$, p , and $qBar$ they produce. (We can ignore the *EndPOEnq* and *BeginPODeq* actions.)

BeginPOEnq of $p2$: $elts = \{p2\}$, $p = \langle p1 \rangle$, $qBar = \langle \rangle$

BeginPOEnq of $p1$: $elts = \{p1, p2\}$, $p = \langle p1, p2 \rangle$, $qBar = \langle p1, p2 \rangle$

EndPODeq of $p2$: $elts = \{p2\}$, $p = \langle p2 \rangle$, $qBar = \langle p2 \rangle$

BeginPOEnq of $p7$: $elts = \{p2, p7\}$, $p = \langle p2, p3 \rangle$, $qBar = \langle p2 \rangle$

Note that two elements have been appended to $qBar$ at the same time. Since *Fifo* appends elements to *queue* one at a time, we will need a stuttering variable to add an extra step so that under the refinement mapping they are added one at a time to *queue*.

If *PP* is correct, then the elements $p3, \dots, p7$ must be enqueued (not necessarily in that order) before $p7$ can be dequeued. But p controls only the order in which *EndPODeq* $_p$ actions can dequeue elements. It doesn't restrict *EndPODeq* $_p$ actions. Suppose the *EndPOEnq* $_p$ action for the *BeginPOEnq* $_p$ action that enqueued $p7$ is now executed. Since that action has to implement the *EndDeq* action of *Fifo* (because it changes the interface variable *enq* the way the *EndDeq* action does), $p7$ must be appended to $qBar$ before the *EndPODeq* action occurs. (We will use a stuttering variable to add the necessary step.)

At that point, it's still possible for $p2$ to be dequeued, but any further dequeuing should be impossible. Therefore, *PP* is not correct. Let's see why further dequeuing is impossible. Since the predictions of *PP* determine what elements can be dequeued, for further dequeuing to occur, $p3$ must be the next element dequeued. This means that $p3$ must be enqueued and added to $elts$. However, $p7$ is still in $elts$ and its *EndPOEnq* operation has occurred, so $p7 \prec p3$ must hold. This implies that $p7$ must be dequeued before $p3$ is, something that the predictions of *PP* do not predict. So, $p3$ and any elements other than $p2$ (which is in $qBar$) can never be dequeued and hence never removed from $qBar$. So elements can keep being enqueued and appended to $qBar$ by the *EndPOEnq* actions, but once $p2$ is dequeued (if it ever is) no further dequeue operations can finish.

The value of $qBar$ always equals the concatenation of these two sequences (either or both of which could be empty) of elements in $elts$:

Dequeueable: A prefix of the predictions in p that could at that point be dequeued by a sequence of *EndPODeq* actions.

Nondequeueable: A sequence of elements of $elts$ that cannot be dequeued by a sequence of *EndPODeq* actions, and were added to $elts$ by an Enqueue operation that has completed.

A sequence of predictions can be dequeued if it satisfies the following two conditions:

- Each of them predicts an element that is currently in $elts$.
- No two of them predict the same element.
- If $el1$ is in the sequence of predictions and $el2 \prec el1$ for some $el2 \in elts$, then $el2$ must be in the sequence and precede $el1$ in it.

An element e in $elts$ is appended to the *Nondequeueable* part of $qBar$ by the *EndPOEnq* $_p$ action of the enqueue operation that added e to $elts$ if e is not already in $qBar$. (If e is already in $qBar$, it must be in the *Dequeueable* part.) It has to be in $qBar$ at that point because the *EndPOEnq* $_p$ action must implement the *EndEnq* action of *Fifo*, and that value is in *queue* when the *EndEnq* step occurs. (A stuttering step must be added before the *EndPOEnq* action so that, under the refinement mapping, the value is added to *queue* before that step.)

A *BeginPOEnq* action that adds e should append e to $qBar$ iff

- (1) The *Nondequeueable* part of $qBar$ is empty and
- (2) Appending e to $qBar$ makes $qBar$ a prefix of sequence p of predictions that is dequeueable.

Condition (2) holds iff (1) holds and

- (a) e is the prediction in the new value of p (the value after execution of the *BeginPOEnq* action) that comes immediately after the prefix of p that equals $qBar$, and

(b) there is no element ee in elt that is not in $qBar$ such that $ee \prec e$.

However, condition (b) is impossible if (1) holds, since $ee \prec e$ can hold iff the *EndPOEnq* action of the enqueue operation that added ee to $elts$ has already occurred. But since ee is not in $qBar$, it was not in $qBar$ when that action occurred, so the action would have appended ee to the *Nondequeueable* part of $qBar$, contradicting (1). Therefore the *EndPOEnq* action that adds e to $elts$ must append it to the end of $qBar$ iff conditions (1) and (2a) hold.

We saw in the example above that enqueueing $p1$ caused both $p1$ and $p2$ to be appended to $qBar$. In general, when the *Nondequeueable* part of $qBar$ is empty, appending any element e to $qBar$ should also cause the next element in p to be appended to $qBar$ if it is in $elts$. Therefore, if condition (1) holds, a *BeginPOEnq* action should set $qBar$ to $qBar \circ seq$, where seq is the maximal subsequence of p that follows the prefix $qBar$ of p .

Finally, because the dequeueable prefix of $qBar$ can be defined as a state function, there's no need to keep it in a history variable. Instead of adding the history variable $qBar$, we add a history variable h whose value is the nondequeueable suffix of $qBar$. We can then consider $qBar$ to be a state function whose value equals the dequeueable prefix of p and the sequence q .

Stuttering Variable s

As explained in Section 7.6.3.3, we need to add stuttering steps in three places to *POFifo_pq* to be able to define a refinement mapping under which *POFifo* implements *Fifo*:

- Immediately before an *EndPODeq_pq* step we need to add one stuttering step.
- immediately before an *EndPOEnq* step that appends an element to $qBar$ not in pg , we need to execute one stuttering step.
- Immediately after a *BeginPOEnq_pq* step we need to add as one stuttering step for every element added to $qBar$. That is, a number of stuttering steps equal to amount by which the *BeginPOEnq* step increases the length of $qBar$.

We can add the required number of stuttering steps after every *BeginPOEnq_pq* step with a stuttering variable s as follows.

1. The condition $s = 0$ is conjoined to the initial-state predicate.
2. Replace the *BeginPOEnq_pq(e)* action by

$$\begin{aligned} & \vee (s = 0) \wedge \text{BeginPOEnq_pq}(e) \wedge (s' = exp) \\ & \vee (s > 0) \wedge (s' = s - 1) \wedge \text{UNCHANGED } POv_pq \end{aligned}$$

where the expression exp is the number of stuttering steps to be added.

3. Replace each of the other three subactions A of *PONext_pq* by

$$(s = 0) \wedge A \wedge (s' = s)$$

We can add a single stuttering step before every *EndPOEnq_pq(d)* step with a stuttering variable s in the same way, except replacing step 2 by:

2. $\vee \wedge (s = 0) \wedge \text{ENABLED } \text{EndPODeq_pq}(d)$
 $\wedge (s' = 1) \wedge \text{UNCHANGED } POv_pq$
 $\vee \wedge (s = 1)$
 $\wedge \text{EndPODeq_pq}(d) \wedge (s' = 0)$

We need a stuttering step before an $EndPOEnq_pq(e)$ step iff the step appends an element to q . It doesn't hurt always to add the step, since an extra stuttering step does no harm. However, it's a bit more convenient not to do that. We can add the stuttering step iff necessary by replacing step 2 with:

```
2.  $\vee \wedge (s = 0) \wedge \text{ENABLED } EndPOEnq\_pq(e)$ 
    $\wedge \text{IF } \dots \text{ THEN } (s' = 1) \wedge \text{UNCHANGED } POv\_pq \text{ ELSE } EndPOEnq(e) \wedge (s' = s)$ 
 $\vee \wedge (s = 1)$ 
    $\wedge EndPOEnq\_pq(e) \wedge (s' = 0)$ 
```

In order to use those stuttering steps in defining the refinement mapping, we want the stuttering variable to contain information about the action to which it's adding stuttering steps. Those could be added later with another history variable, but they can be put directly into s by making s a tuple one component of which is doing the counting down and the other components are storing the information. One such piece of information is the process taking the stuttering steps.

Instead of adding these three kinds of stuttering steps with three different variables, we use a single stuttering variable with a component that indicates what kind of action it's adding the steps to.

EXTENDS *IPOFifo, Integers, Sequences, TLC*

We will need the assumption on *NoData* to implement the constant of the same name in *Fifo*.

```
CONSTANT NoData
ASSUME NoData  $\notin$  Data
```

```
*****
If I were writing this for myself, I would probably add these three auxiliary variables all at
once. However, for pedagogical reasons, we will add them one at a time to obtain three different
abstract programs. We first add the prophecy variable  $p$  to POFifo to obtain the abstract program
POFifo-p.
*****
```

```
VARIABLE p
POv_p  $\triangleq \langle POv, p \rangle$ 

POTypeOK_p  $\triangleq POTypeOK \wedge (p \in Seq(Datums))$ 

POInit_p  $\triangleq POInit \wedge p = \langle \rangle$ 

BeginPOEnq_p(e)  $\triangleq \wedge BeginPOEnq(e)$ 
                $\wedge \exists el \in Datums : p' = Append(p, el)$ 

EndPOEnq_p(e)  $\triangleq EndPOEnq(e) \wedge (p' = p)$ 

BeginPODeq_p(d)  $\triangleq BeginPODeq(d) \wedge (p' = p)$ 
```

The rules for adding a prophecy sequence variable would require us to define $EndPODeq_p(d)$ by rewriting the definition of $EndPODeq(d)$ to have the form

$$EndPODeq(d) \triangleq \exists el \in Datums : A(el, d)$$

and defining

$$EndPODeq_p(d) \triangleq A(p[1], d) \wedge (p' = Tail(p))$$

We can do this by defining $A(el, d)$ to be

$$EndPODeq(d) \wedge (elts' = elts \setminus \{el\})$$

This definition of $A(el, d)$ yields the following definition of $EndPODeq_p(d)$.

$$EndPODeq_p(d) \triangleq EndPODeq(d) \wedge (elts' = elts \setminus \{p[1]\}) \wedge (p' = Tail(p))$$

$$PONext_p \triangleq \vee \exists e \in EnQers : BeginPOEnq_p(e) \vee EndPOEnq_p(e) \\ \vee \exists d \in DeQers : BeginPODeq_p(d) \vee EndPODeq_p(d)$$

$$POFifo_p \triangleq POInit_p \wedge \Box[PONext_p]_{POv_p}$$

We next add the history variable $qBar$. First we define the dequeueable prefix pg of p . Remember that it is the largest prefix satisfying this condition:

- Each element of it is in $elts$.
- No two of them predict the same element.
- If $el1$ is in the prefix and $el2 \prec el1$ for some $el2 \in elts$, then $el2$ must precede $el1$ in the prefix

It's tricky to get the definition right, so to check it I gave two definitions. The first comes directly from this definition, the second is a recursive definition that is more efficiently executed by *TLC*. (Checking them against one another revealed several errors in both of them before they agreed.) The first definition requires the definition of $SetMax(S)$ to be the maximum of a set of integers, defined to equal 0 if S is the empty set.

$$SetMax(S) \triangleq \text{IF } S = \{\} \text{ THEN } 0 \text{ ELSE CHOOSE } n \in S : \forall m \in S : n \geq m$$

$$prefix(seq, n) \triangleq [j \in 1 \dots n \mapsto seq[j]]$$

$$canDeque(seq) \triangleq \forall i \in 1 \dots Len(seq) : \\ \wedge p[i] \in elts \\ \wedge \forall j \in 1 \dots (i-1) : p[j] \neq p[i] \\ \wedge \forall e \in elts : \\ e \prec p[i] \Rightarrow \exists j \in 1 \dots (i-1) : e = p[j]$$

$$pg \triangleq \text{LET } n \triangleq SetMax(\{m \in 1 \dots Len(p) : canDeque(prefix(p, m))\}) \\ \text{IN } prefix(p, n)$$

The state predicate pgE is defined by the recursive definition of pg that is more efficiently executed by *TLC*. It evaluates $maxdq(n, S)$ recursively, starting with $maxdq(0, \{\})$, and reaching $maxdq(n, S)$ for $n > 0$ iff the first n elements of p are dequeueable, where S is the set of elements in p . The recursion stops when n is the length of p or element $n+1$ of p is not dequeueable.

When model checking the program, we have *TLC* substitute pgE for pg .

$$pgE \triangleq \text{LET RECURSIVE } maxdq(-, -) \\ maxdq(n, S) \triangleq \text{IF } n = Len(p) \\ \text{THEN } n \\ \text{ELSE LET } e \triangleq p[n+1] \\ \text{IN IF } \wedge e \in elts \setminus S \\ \wedge \forall ee \in elts \setminus (S \cup \{e\}) : \\ \neg(ee \prec e) \\ \text{THEN } maxdq(n+1, S \cup \{e\})$$

ELSE n
IN $[i \in 1 \dots \text{maxdq}(0, \{\}) \mapsto p[i]]$

Define $\text{appendToH}(e)$ the formula that, when executing an $\text{EndPOEnq}(e)$ action, is true iff that action should append the datum enqueued by enqueueer e to q . This is the case iff that datum is in elts but is not in the sequence pg .

$$\begin{aligned} \text{appendToH}(e) \triangleq & \wedge \text{adding}[e] \in \text{elts} \\ & \wedge \forall i \in 1 \dots \text{Len}(pg) : pg[i] \neq \text{adding}[e] \end{aligned}$$

VARIABLE $qBar$

$$POv_{pq} \triangleq \langle POv, p, qBar \rangle$$

$$POTypeOK_{pq} \triangleq POTypeOK_p \wedge (qBar \in \text{Seq}(\text{Datums}))$$

$$POInit_{pq} \triangleq POInit_p \wedge (qBar = \langle \rangle)$$

$$\text{BeginPOEnq}_{pq}(e) \triangleq \text{BeginPOEnq}_p(e) \wedge (qBar' = qBar)$$

$$\begin{aligned} \text{EndPOEnq}_{pq}(e) \triangleq & \wedge \text{EndPOEnq}_p(e) \\ & \wedge qBar' = \text{IF } \text{appendToH}(e) \\ & \quad \text{THEN } \text{Append}(qBar, \text{adding}[e]) \\ & \quad \text{ELSE } qBar \end{aligned}$$

$$\text{BeginPODeq}_{pq}(d) \triangleq \text{BeginPODeq}_p(d) \wedge (qBar' = qBar)$$

$$\text{EndPODeq}_{pq}(d) \triangleq \text{EndPODeq}_p(d) \wedge (qBar' = qBar)$$

$$\begin{aligned} PONext_{pq} \triangleq & \vee \exists e \in \text{EnQers} : \text{BeginPOEnq}_{pq}(e) \vee \text{EndPOEnq}_{pq}(e) \\ & \vee \exists d \in \text{DeQers} : \text{BeginPODeq}_{pq}(d) \vee \text{EndPODeq}_{pq}(d) \end{aligned}$$

$$IPOFifo_{pq} \triangleq POInit_{pq} \wedge \square [PONext_{pq}]_{POv_{pq}}$$

We now add stuttering variable s as described above. Instead of writing an expression ENABLED A for the two actions A having a stuttering step added before them, I have “evaluated” that ENABLED expression.

VARIABLE s

$$POv_{pqs} \triangleq \langle POv, p, qBar, s \rangle$$

$$\begin{aligned} POTypeOK_{pqs} \triangleq & \wedge POTypeOK_{pq} \\ & \wedge s \in \{\langle 0 \rangle\} \cup (\text{Nat} \times \{\text{"BeginEnq"}\} \times \text{EnQers}) \\ & \quad \cup (\text{Nat} \times \{\text{"EndEnq"}\} \times \text{EnQers}) \\ & \quad \cup (\text{Nat} \times \{\text{"EndDeq"}\} \times \text{DeQers} \times \text{Datums}) \end{aligned}$$

$$POInit_{pqs} \triangleq POInit_{pq} \wedge (s = \langle 0 \rangle)$$

$$\begin{aligned} \text{BeginPOEnq}_{pqs}(e) \triangleq & \vee \wedge s[1] = 0 \\ & \wedge \text{BeginPOEnq}_{pq}(e) \\ & \wedge s' = \langle \text{Len}(pg') - \text{Len}(pg), \text{"BeginEnq"}, e \rangle \\ & \vee \wedge (s[1] > 0) \wedge (s[2] = \text{"BeginEnq"}) \wedge (s[3] = e) \end{aligned}$$

$$\begin{aligned}
& \wedge (s' = \langle s[1] - 1, \text{"BeginEnq"}, e \rangle) \\
& \wedge \text{UNCHANGED } POv_{pq} \\
EndPOEnq_{pqs}(e) & \triangleq \vee \wedge (s[1] = 0) \wedge (enq[e] \neq Done) \\
& \wedge \text{IF } appendToH(e) \\
& \quad \text{THEN } \wedge s' = \langle 1, \text{"EndEnq"}, e \rangle \\
& \quad \wedge \text{UNCHANGED } POv_{pq} \\
& \quad \text{ELSE } \wedge EndPOEnq_{pq}(e) \\
& \quad \wedge s' = s \\
& \vee \wedge (s[1] = 1) \wedge (s[2] = \text{"EndEnq"}) \wedge (s[3] = e) \\
& \quad \wedge EndPOEnq_{pq}(e) \\
& \quad \wedge s' = \langle 0 \rangle \\
BeginPODeq_{pqs}(d) & \triangleq (s[1] = 0) \wedge BeginPODeq_{pq}(d) \wedge (s' = s) \\
EndPODeq_{pqs}(d) & \triangleq \vee \wedge (s[1] = 0) \wedge (deq[d] = Busy) \wedge (pg \neq \langle \rangle) \\
& \quad \wedge s' = \langle 1, \text{"EndDeq"}, d, Head(pg) \rangle \\
& \quad \wedge \text{UNCHANGED } POv_{pq} \\
& \vee \wedge (s[1] = 1) \wedge (s[2] = \text{"EndDeq"}) \wedge (s[3] = d) \\
& \quad \wedge EndPODeq_{pq}(d) \\
& \quad \wedge s' = \langle 0 \rangle \\
PONext_{pqs} & \triangleq \vee \exists e \in EnQers : BeginPOEnq_{pqs}(e) \vee EndPOEnq_{pqs}(e) \\
& \quad \vee \exists d \in DeQers : BeginPODeq_{pqs}(d) \vee EndPODeq_{pqs}(d) \\
IPOFifo_{pqs} & \triangleq POInit_{pqs} \wedge \square [PONext_{pqs}]_{POv_{pqs}}
\end{aligned}$$

We now define the state function *queueBar*, *deqInnerBar*, and *enqInnerBar* so that *POFifo_{pqs}* implements *Fifo* under a refinement mapping that implements the internal variables *queue*, *deqInner*, and *enqInner* with these state functions.

First, we define the sequence *queueE* of elements whose data values define *queueBar*. It equals *qBar*, except that:

- When adding stuttering steps after *BeginPOEnq_{pq}*, the elements at the end of *pg* have to be appended one at a time by those stuttering steps.
- The head of *pg* has to be removed by the stuttering step added before *EndPODeq_{pq}*.
- The last element of *qBar* has to be appended by a stuttering step added before *EndPOEnq_{pq}*.

$$\begin{aligned}
queueE & \triangleq \\
& (\text{IF } s[1] > 0 \\
& \quad \text{THEN IF } s[2] = \text{"BeginEnq"} \\
& \quad \quad \text{THEN } [i \in 1 \dots (Len(pg) - s[1]) \mapsto pg[i]] \\
& \quad \quad \text{ELSE IF } s[2] = \text{"EndDeq"} \\
& \quad \quad \quad \text{THEN } Tail(pg) \\
& \quad \quad \quad \text{ELSE } pg \\
& \quad \text{ELSE } pg) \\
& \circ \\
& (\text{IF } (s[1] > 0) \wedge (s[2] = \text{"EndEnq"}))
\end{aligned}$$

```

THEN Append(qBar, adding[s[3]])
ELSE qBar)

```

$queueBar \triangleq [i \in 1 \dots Len(queueE) \mapsto queueE[i][1]]$

The state predicates *deqInnerBar* and *enqInnerBar* that are substituted for variables *deq* and *enq* of the *Fifo* program by the refinement mapping are defined so that they make *deq* and *enq* change their values the way they are supposed to when *queueBar* changes. I found it straightforward to write the definitions, but having the model checker to find errors helped.

```

deqInnerBar  $\triangleq$  [d ∈ DeQers  $\mapsto$ 
  IF deq[d] = Busy
  THEN IF (s[1] > 0) ∧ (s[2] = "EndDeq") ∧ (s[3] = d)
    THEN Head(pg)[1]
    ELSE NoData
  ELSE deq[d]]

```

```

enqInnerBar  $\triangleq$  [e ∈ EnQers  $\mapsto$ 
  IF  $\vee enq[e] = Done$ 
   $\vee \exists i \in 1 \dots Len(queueE) : adding[e] = queueE[i]$ 
   $\vee adding[e] \notin elts$ 
   $\vee (s[1] > 0) \wedge (s[2] = "EndDeq") \wedge (adding[e] = Head(p))$ 

```

This disjunct is needed because there is one case in which the stuttering step after *BeginPOEnq*(*e*) that implements the *DoEnq*(*e*) of *IFifo* has occurred, the *EndPOEnq*(*e*) step has not occurred, and *adding*[*e*] is neither in *elts* nor in *queueE*. That's when the stuttering step before the *EndPODeq*(*d*) step that dequeues *adding*[*e*] has occurred, removing *adding*[*e*] from *queueE*, but the following *EndPODeq*(*d*) step has not occurred. In such a state, only this conjunct of the IF condition is true.

```

THEN Done
ELSE Busy]

```

Finally, we define *F!IFifo* to equal *IFifo* WITH *queue* \leftarrow *queueBar*, ... and assert that *IPOFifo_pqs* implies *F!IFifo*.

$F \triangleq$ INSTANCE *IFifo* WITH *queue* \leftarrow *queueBar*, *deqInner* \leftarrow *deqInnerBar*,
enqInner \leftarrow *enqInnerBar*

THEOREM *IPOFifo_pqs* \Rightarrow *F!IFifo*