An Example of a Constructive Specification of a Queue: Preliminary Report

Leonora Benzinger

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois
June, 1985
1. Introduction

The following is an example of the constructive specification of a queue which is done in the style of [Jones 80] using the Vienna Development Method. The basic approach is that of data type refinement. While the techniques we used are not restricted to those used by Jones, particularly with respect to the method for proving properties of the retrieve function for linked lists, the notation is consistent with his.

2. The specification of a Queue

2.1. States and types for the Queue operations

Queue = Element-list

INIT
states: Queue

ENQUEUE
states: Queue
type: Element ->

DEQUEUE
states: Queue
type: -> Element

EMPTY
states: Queue
type: -> Boolean

2.2. Pre- and post-conditions for the Queue operations

post-INIT(q,q') = q' = <>.
post-ENQUEUE(q,e,q') = q' = q || <e>.
pre-DEQUEUE(q) = q ≠ <>.
post-DEQUEUE(q,e,q') = q' = tl(q) and e = hd(q).
post-EMPTY(q,q',b) = q = q' and (b <=> q = <>).
3. A Data Refinement of a Queue in Terms of Linked Lists

3.1. A queue as a linked list

\[
\text{Queue1} = [\text{node}];
\]
\[
\text{node} = \text{record}
\]
\[
\text{E : Element;}
\]
\[
\text{PTR : Queue1}
\]
\[
\text{end;}
\]

3.2. The retrieve function

The retrieve function is a function which maps the linked list representation of a queue into a list representation.

\[
\text{retr : Queue1} \rightarrow \text{Queue}
\]
\[
\text{retr}(q1) \equiv \text{if } q1 = \text{NIL} \text{ then } < > \text{ else } (\langle q1.E \rangle \parallel \text{retr}(q1.PTR)).
\]

The data type invariant for Queue and Queue1 is TRUE.

3.3. Queue1 models Queue

In order to show that Queue1 models Queue the retrieve function must map all of Queue1 into Queue and every member of Queue must be the value of some member of Queue1 under the retrieve mapping. These two conditions are stated more precisely as rules aa and ab in [Jones, 1980, p.187]. In addition to rules aa and ab, the pre- and post-conditions for the operations for Queue1 must imply the pre- and post-conditions for the corresponding operations for Queue for members of Queue1 mapped back to Queue by the retrieve function. These conditions are precisely stated as rules da and ra [Jones, 1980, p.187].

3.3.1. Rules aa and ab are satisfied by the retrieve function

**aa.** \((\forall q1 \in \text{Queue1})(\exists q \in \text{Queue} \text{ such that } q = \text{retr}(q1)).\)

**Proof.** We use structural induction on Queue1. Suppose \(q1 = \text{NIL} \). Then \(\text{retr}(q1) = < > \) and \(< > \in \text{Queue}.\)

Suppose \(q1 \in \text{Queue1} \) and \(q1 \neq \text{NIL} \). Then \(\text{retr}(q1) = \langle q1.E \rangle \parallel \text{retr}(q1.PTR). \) By the induction hypothesis there exists \(q' \in \text{Queue} \) such that \(q' = \text{retr}(q1.PTR). \) Let \(q = \langle q1.E \rangle \parallel q'. \) Clearly, \(q \in \text{Queue} \) and \(q = \text{retr}(q1).\)

**ab.** \((\forall q \in \text{Queue})(\exists q1 \in \text{Queue1} \text{ such that } q = \text{retr}(q1)).\)

**Proof.** We use structural induction on Queue. Suppose that \(q = < > \). If \(q1 = \text{NIL} \) then by the definition of the retrieve function \(\text{retr}(q1) = q.\)

Let \(q \in \text{Queue} \) and suppose that \(q \neq \text{NIL} \). It follows that \(q = \text{hd}(q) \parallel \text{tl}(q) \) where \(\text{tl}(q) \in \text{Queue}. \) By the induction hypothesis, there exists \(q1' \in \text{Queue1} \) such that \(\text{retr}(q1') = \text{tl}(q). \) Define \(q1 \in \text{Queue1} \) as follows:
Then \( \text{retr}(q1) = q \).

### 3.3.2. Specification of the operations on Queue1

To specify the operations on Queue1 in terms of pre- and post- conditions we need an extension of some of the notions introduced by Jones [Jones 80, chapter 9] for lists to linked lists. The queue operations of initialization, enqueue, and empty are straightforward to implement in terms of linked lists. A difficulty occurs in the post-condition for the enqueue operation for a queue implemented on linked lists. If we choose to introduce a new argument, say, tail to describe the element appended at the end of a queue, then tail must be expressed in terms of the new queue. This is because of the form of the post-condition for the enqueue operation at the previous level of abstraction (in terms of lists) is in terms of the new queue which is obtained from the old one by concatenation of a list of a single element to the end of the old queue.

This can be done by the following:

\[
\text{tail} = \langle \text{hd}(\text{rev}(q1)) \rangle \quad \text{for } q1 \in \text{Queue1}
\]

and properly extended notions of \( \text{hd} \), \( \text{rev} \) (the reverse order on lists), and \( \langle \rangle \) to linked lists. If the post-condition for the enqueue operation is stated in terms of tail, it is very awkward to verify rule ra for this operation because the post-condition for the enqueue operation on lists is stated in terms of queues of lists, not "tail ends" of queues. This approach then seems to require a backtracking in the post-condition for the enqueue operation in terms of lists using the notion of tail.

We use another approach, which is to extend the notions used for lists in the post-condition for the enqueue operation of a queue implemented in terms of lists to corresponding notions for linked lists. This has the advantage of making the post-condition for the enqueue operation in terms of linked lists very similar in form to the post-condition for enqueue for queues of lists. This also makes makes rule ra reasonably straightforward to check.

### 3.3.3. Extension of the theory of lists to linked lists

We define the notions of head, tail, and concatenation for linked lists. By an abuse of notation, we use the same names for these notions which are defined for lists [Jones 80, chapter 9].

Let \( \text{llist}, \text{llist1}, \text{llist2} \) be linked lists. Denote by \( \text{hd} \) the head of a linked list. It is defined as follows:

\[
\text{hd}(\text{llist}) \equiv \text{llist.E}.
\]

The tail of a linked list is denoted by \( \text{tl} \). The definition is:

\[
\text{tl}(\text{llist}) \equiv \text{llist.PTR}.
\]

The length of a linked list is denoted by \( \text{len} \). The definition is:

\[
\text{len}(\text{llist}) \equiv \begin{cases} 
0 & \text{if } \text{llist} = \text{NIL} \\
1 + \text{len}(\text{tl}(\text{llist})) & \text{otherwise}
\end{cases}
\]

The index operator extended to linked lists is given by:

\[
\text{llist}(i) \equiv \begin{cases} 
\text{hd}(\text{llist}) & \text{if } i = 1 \\
0 & \text{otherwise}
\end{cases}
\]
else tl(llist)(i - 1).

The concatenation operator extended to linked lists is given by:

\[
\text{llist} \mid\mid \text{llist2} = \text{the unique linked list such that:} \\
\forall i \in \{1, \ldots, \text{len(llist1)}\} (\text{llist}(i) = \text{llist1}(i)) \text{ and} \\
\forall i \in \{1, \ldots, \text{len(llist2)}\} (\text{llist}(i + \text{len(llist1)}) = \text{llist2}(i)).
\]

We observe that llist \mid\mid \text{NIL} = \text{NIL} \mid\mid \text{llist} = \text{llist}.

3.3.4. The retrieve function has an inverse

To define \text{<hd(llist)>} where llist is a linked list, we need the inverse of the retrieve function. We observe that the retrieve function, retr, has a natural extension from Queue1 to List1, the collection of all linked lists, by defining retrieve as follows:

\[
\text{retr} : \text{List1} \rightarrow \text{List} \\
\text{retr}(l) = \text{if } l = \text{NIL} \text{ then } <> \text{ else } (<l.E> \mid\mid \text{retr}(l.PTR)).
\]

The next lemma proves that retr is 1 to 1 and therefore, the inverse exists.

Lemma. Let l1, l2 in List1 and assume that retr(l1) = retr(l2). Then l1 = l2.

Proof. The proof is by structural induction. Suppose l1 = NIL and l2 \neq NIL. Then retr(l1) = <> but retr(l2) = <l2.E> \mid\mid retr(l2.PTR). This contradicts the assumption that retr(l1) = retr(l2).

Next, let l1 \neq NIL and retr(l1) = retr(l2) for some l2 in List1. Furthermore, suppose that for each linked sublist l1' of l1, if retr(l1') = retr(l2'), where l2' is a linked sublist of l2, then l1' = l2'. We note that l2 \neq NIL since l2 = NIL implies that retr(l2) = <>, in which case retr(l2) \neq retr(l1). Therefore retr(l2) = <l2.E> \mid\mid retr(l2.PTR). We also have retr(l1) = <l1.E> \mid\mid retr(l1.PTR). Since retr(l1) = retr(l2), l1.E = l2.E and retr(l1.PTR) = retr(l2.PTR). By the induction hypothesis, l1.PTR = l2.PTR. We conclude that l1 = l2.

We observe that the rules aa and ab hold when applied to linked lists. The proofs carry over by replacing queues implemented in terms of lists and linked lists by arbitrary lists and linked lists. Thus, the function retr is a 1 to 1 mapping onto the set of lists, List.

Let l in List. There exists a unique l1 in List1, by rule ab, such that retr(l1) = l. Define invretr as:

\[
\text{invretr}(l) \equiv l1.
\]

This definition can be restricted in a natural way to hold only for queues implemented in terms of lists and linked lists.

We are now in a position to extend the list notation to linked lists. Let l1 in List1. Then there exists (a unique) l in List such that retr(l1) = l. Assume furthermore that l1 \neq NIL and that l1.E = e. We define the linked list formed from the element l1.E as follows:

\[
< l1.E > \equiv \text{invretr}(< \text{hd}(l1)>).
\]

In particular, \text{<hd(l1)>} = \text{invretr}(< \text{hd}(l1)>). Notice that the list in the term on the left is a linked list, while the list in the term on the right hand side of the equivalence is not a linked list.
3.3.5. States and types for the Queue1 operations

Queue1 = [node];
node = record
  E : Element;
  PTR : Queue1
end;

INIT1
states : Queue1
type : Element —>

ENQUEUE1
states : Queue1
type : Element —>

DEQUEUE1
states : Queue1
type : Element —>

EMPTY1
states : Queue1
type : Boolean

3.3.6. Pre- and post-conditions for the Queue1 operations

post-INIT1(q1,q1') = q1' = NIL.
post-ENQUEUE1(q1,q1',e) = q1' = q1 || <e>.
pre-DEQUEUE1(q1) = q1 \neq NIL.
post-DEQUEUE1(q1,q1',res) = q1' = q1.PTR and res = q1.E.
post-EMPTY1(q1,q1',b) = q1' = q1 and (b <=> q1 = NIL).

3.3.7. The retrieve function is an isomorphism

Lemma. Let <e>, l1 \in List1 and suppose that len(l1) = n for some integer n > 0. Then (l1 || <e>).PTR = l1' || <e> where l1 \in List1 and len(l1) = n - 1.
Proof. Suppose n = 1. Then l1 = <e1> for some e1 \in Element. We have (l1 || <e>).PTR = (<e1> || <e>).PTR = <e1> = NIL || <e>. NIL \in List1 and len(NIL) = 0.

Let len(l1) = n. Then l1 = <e1, e2, ..., en> where ei \in Element for i = 1, 2, ..., n and the ei's are not necessarily distinct. We have

(l1 || <e>).PTR = (<e1, e2, ..., en> || <e>).PTR
= <e1, e2, ..., en, e>.PTR
= <e2, ..., en, e>
= <e2, ..., en> || <e>.

Let l1' = <e2, ..., en>. We observe that l1' \in List1 and len(l1') = n - 1.
Lemma. Let \( <e> \), \( l1 \in \text{List1} \). Then \( \text{retr}(l1 \parallel <e>) = \text{retr}(l1) \parallel <e> \).

Proof. We use induction on \( \text{len}(l1) \). Suppose that \( \text{len}(l1) = 0 \). Then \( l1 = \text{NIL} \). It follows that \( \text{retr}(l1 \parallel <e>) = \text{retr}(<e>) = <e> = \text{retr}(l1) \parallel <e> \).

Assume that the lemma holds \( \forall l1' \in \text{List1} \) for which \( \text{len}(l1') < n \) for some integer \( n > 0 \). Let \( l1 \in \text{List1} \) and suppose that \( \text{len}(l1) = n \) and let \( l1.E = e' \). We have

\[
\text{retr}(l1 \parallel <e>) = \text{retr}(<l1 \parallel <e>).E \parallel \text{retr}(l1 \parallel <e>).\text{PTR}.
\]

We note that \( l1.E = (l1 \parallel <e>).E \) so that

\[
\text{retr}(l1 \parallel <e>) = <e'> \parallel \text{retr}(l1 \parallel <e>).\text{PTR}.
\]

We can rewrite \( (l1 \parallel <e>).\text{PTR} \) as \( l1' \parallel <e> \) where \( \text{len}(l1') < n \) from the previous lemma. By the induction hypothesis,

\[
\text{retr}(l1 \parallel <e>).\text{PTR} = \text{retr}(l1' \parallel <e>) = \text{retr}(l1') \parallel <e>.
\]

It follows that

\[
\text{retr}(l1 \parallel <e>) = <e'> \parallel (\text{retr}(l1') \parallel <e>).
\]

But from the definition of the retrieve function\n
\[
\text{retr}(l1) = <\text{hd}(l1).> \parallel \text{retr}(l1.\text{PTR}).
\]

Therefore,

\[
\text{retr}(l1 \parallel <e>) = \text{retr}(l1) \parallel <e>.
\]

Theorem. \( \forall l1, l2 \in \text{List1}, \text{retr}(l1 \parallel l2) = \text{retr}(l1) \parallel \text{retr}(l2) \), that is, the retrieve function is an isomorphism from the set of linked lists to the set of lists.

Proof. We use induction on \( \text{len}(l2) \). When \( \text{len}(l2) = 0 \) we have

\[
\text{retr}(l1 \parallel l2) = \text{retr}(l1 \parallel <> ) = \text{retr}(l1).
\]

In List we have

\[
\text{retr}(l1) \parallel \text{retr}(l2) = \text{retr}(l1) \parallel <> = \text{retr}(l1).
\]

Assume that \( \text{retr}(l1 \parallel l2') = \text{retr}(l1) \parallel \text{retr}(l2') \) for \( l2' \in \text{List1} \) for which \( \text{len}(l2') < n \) for some positive integer \( n \). Suppose that \( \text{len}(l2) = n \). Then

\[
\text{retr}(l1) \parallel \text{retr}(l2) = \text{retr}(l1) \parallel (<\text{hd}(l2).> \parallel \text{retr}(\text{tl}(l2)))
\]

\[
= (\text{retr}(l1) \parallel <\text{hd}(l2).>) \parallel \text{retr}(\text{tl}(l2)).
\]

By the induction hypothesis and the previous lemma,

\[
(\text{retr}(l1) \parallel <\text{hd}(l2).>) \parallel \text{retr}(\text{tl}(l2)) = \text{retr}(l1 \parallel \text{hd}(l2)) \parallel \text{retr}(\text{tl}(l2)).
\]

Since \( \text{len}(l2) = n \), \( \text{len}(\text{tl}(l2)) = n - 1 \) so that we can use the induction hypothesis with \( l2' = \text{tl}(l2) \). It
follows that
\[
\text{retr}(l_1 || <\text{hd}(l_2)> || \text{retr}(\text{tl}(l_2))) = \text{retr}((l_1 || <\text{hd}(l_2)> || \text{tl}(l_2))
\]
\[
= \text{retr}(l_1 || (<\text{hd}(l_2)> || \text{tl}(l_2)))
\]
\[
= \text{retr}(l_1 || l_2).
\]

3.3.8. The operations on Queue1 model the operations on Queue

The next step is to show that each of the new operations on Queue1: INIT1, ENQUEUE1, DEQUEUE1, and EMPTY1 correspond to the operations INIT, ENQUEUE, DEQUEUE, and EMPTY on Queue. For each of the operations on Queue1 we must show that both da and ra [Jones 80] hold, where da and ra are:

\[\text{da. } (\forall q_1 \in \text{Queue1})(\text{pre-OP}(\text{retr}(q_1), \text{args}) \Rightarrow \text{pre-OP}(q_1, \text{args})).\]

\[\text{ra. } (\forall q_1 \in \text{Queue1})(\text{pre-INIT}(\text{retr}(q_1), \text{args}) \Rightarrow \text{pre-INIT}(q_1, \text{args})).\]

**Proof.** The proof is immediate since pre-INIT and pre-INIT1 are both TRUE.

\[\text{ra. } (\forall q_1 \in \text{Queue1})(\text{pre-ENQUEUE}(\text{retr}(q_1), \text{args}) \Rightarrow \text{pre-ENQUEUE}(q_1, \text{args})).\]

**Proof.** This follows immediately since the pre-conditions for ENQUEUE and ENQUEUE1 are both TRUE.

\[\text{ra. } (\forall q_1 \in \text{Queue1})(\text{pre-DEQUEUE}(\text{retr}(q_1), \text{args}) \Rightarrow \text{pre-DEQUEUE}(q_1, \text{args})).\]

**Proof.** We have q1' = q1 || <e> and retr(q1') = retr(q1 || <e>). By the lemma of 2.3.7, retr(q1') = retr(q1) || <e>.

\[\text{da. } (\forall q_1 \in \text{Queue1})(\text{pre-EMPTY}(\text{retr}(q_1), \text{args}) \Rightarrow \text{pre-EMPTY}(q_1, \text{args})).\]

**Proof.** Since retr(q1) ≠ <>, q1 ≠ NIL.
Proof. This is immediate since the pre-conditions are both TRUE.

ra. \( \forall q_1 \in \text{Queue1} \) [pre-EMPTY1(q1, args) and post-EMPTY1(q1, args, q1', res) \implies \text{post-EMPTY}(\text{retr}(q1), \text{args}, \text{retr}(q1'), \text{res})].

Proof. We have \( q_1 = q_1' \) and \( b \iff q_1 = \text{NIL} \). Since \( q_1 = q_1' \), \( \text{retr}(q1) = \text{retr}(q1') \). But \( q_1 = \text{NIL} \) implies that \( \text{retr}(q1) = <> \). Therefore, \( b \iff q_1 = \text{NIL} \implies \text{retr}(q1) = <> \). Next, suppose that \( \text{retr}(q1) = <> \). Since retr is 1 to 1, \( q_1 = \text{NIL} \iff b \). Therefore, \( b \iff (\text{retr}(q1) = <> \).

4. The Realization of the Queue Object in Pascal

To realize the queue object in Pascal we need a refinement which maps the queue-like structure into a representation of the queue in terms of pointers and variables on the Pascal "heap".

\[
\text{Queuerep} :: \text{Heap} : \text{Ptr} \rightarrow \text{Noderep}
\]

where \( \text{Noderep} :: \text{ELT} : \text{Element} \)
\[
\text{PTER} : "[\text{Ptr}]".
\]

A further refinement is necessary to go from the queue representation to an implementation of a queue in Pascal.

program queue;

type
  qptr = ^qrec;
  qrec = record
    qdata : char;
    qnext : qptr
  end; (* qrec *)
var
  head : qptr;
  tail : qptr;

function empty : boolean;
begin
  empty := (head = nil)
end; (* empty *)

procedure init;
begin
  head := nil;
  tail := nil
end; (* init *)

procedure enqueue(arrive : qptr);
begin
  if arrive <> nil then
    arrive^.qnext := nil;
  if empty then
    head := arrive
  else tail^.qnext := arrive;
  tail := arrive
end; (* enqueue *)

function dequeue(var head, tail : qptr) : char;
begin
  if head <> nil then
begin
  dequeue := head^.data;
  head := head^.nextq;
  if head == nil then
    tail := nil
  end
end; (* dequeue *)

References.