N87-28302

p.10

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

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

Leonora Benzinger

Computer Science Dept., University of Illinois, Urbana, Illinois 61801

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') $\equiv$ q' = <>.

post–ENQUEUE(q,e,q') $\equiv$ q' = q $\|$ <e>.

pre–DEQUEUE(q) $\equiv$ q $\neq$ <>.
post–DEQUEUE(q,e,q') $\equiv$ q' = tl(q) and e = hd(q).

post–EMPTY(q,q',b) $\equiv$ 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

```
Queue1 = [node];
node   = record
           E : Element;
           PTR : Queue1
          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.

retr : Queue1 —> Queue

retr(q1) ≡ if q1 = NIL then <>
else (<q1.E> ¦¦ 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 80, 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 80, p.187].

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

aa. (∀ q1 ∈ Queue1)(∃ q ∈ Queue such that q = retr(q1)).

Proof. We use structural induction on Queue1. Suppose q1 = NIL. Then retr(q1) = <> and <> ∈ Queue.

Suppose q1 ∈ Queue1 and q1 ≠ NIL. Then retr(q1) = <q1.E> ¦¦ retr(q1.PTR). By the induction hypothesis there exists q' ∈ Queue such that q' = retr(q1.PTR). Let q = <q1.E> ¦¦ q'. Clearly, q ∈ Queue and q = retr(q1).

ab. (∀ q ∈ Queue)(∃ q1 ∈ Queue1 such that q = retr(q1)).

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

Let q ∈ Queue and suppose that q ≠ NIL. It follows that q = hd(q) ¦¦ tl(q) where tl(q) ∈ Queue. By the induction hypothesis, there exists q1' ∈ Queue1 such that retr(q1') = tl(q). Define q1 ∈ Queue1 as follows:

$$q1.E = hd(q) \text{ and } q1.PTR = q1'.$$

Then $retr(q1) = q$.

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:

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

and properly extended notions of hd, 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 llist, llist1, llist2 be linked lists. Denote by hd the head of a linked list. It is defined as follows:

$$hd(llist) \equiv llist.E.$$

The tail of a linked list is denoted by tl. The definition is:

$$tl(llist) \equiv llist.PTR.$$

The length of a linked list is denoted by len. The definition is:

$$\begin{aligned} len(llist) \equiv\ & \text{if } llist = NIL \text{ then } 0 \\ & \text{else } 1 + len(tl(llist)). \end{aligned}$$

The index operator extended to linked lists is given by:

$$llist(i) \equiv \text{if } i = 1 \text{ then } hd(llist)$$

$$\text{else } \mathrm{tl}(\mathrm{llist})(i - 1).$$

The concatenation operator extended to linked lists is given by:

$$\begin{array}{l} \mathrm{llist1} \,\|\, \mathrm{llist2} \equiv \text{the unique linked list such that:} \\ (\forall\, i \in \{1,\ldots,\mathrm{len}(\mathrm{llist1})\}\ (\mathrm{llist}(i) = \mathrm{llist1}(i))) \text{ and} \\ (\forall\, i \in \{1,\ldots,\mathrm{len}(\mathrm{llist2})\}\ (\mathrm{llist}(i + \mathrm{len}(\mathrm{llist1})) = \mathrm{llist2}(i)). \end{array}$$

We observe that $\mathrm{llist} \,\|\, \mathrm{NIL} = \mathrm{NIL} \,\|\, \mathrm{llist} = \mathrm{llist}$.

3.3.4. The retrieve function has an inverse

To define $\langle \mathrm{hd}(\mathrm{llist}) \rangle$ 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 Queuel to List1, the collection of all linked lists, by defining retrieve as follows :

$$\mathrm{retr} : \mathrm{List1} \longrightarrow \mathrm{List}$$

$$\begin{array}{l} \mathrm{retr}(l1) \equiv \text{if } l1 = \mathrm{NIL} \text{ then } \langle\,\rangle \\ \qquad\qquad \text{else } (\langle l1.E \rangle \,\|\, \mathrm{retr}(l1.PTR)). \end{array}$$

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

Lemma. Let l1, l2 in List1 and assume that $\mathrm{retr}(l1) = \mathrm{retr}(l2)$. Then $l1 = l2$.

Proof. The proof is by structural induction. Suppose $l1 = \mathrm{NIL}$ and $l2 \neq \mathrm{NIL}$. Then $\mathrm{retr}(l1) = \langle\,\rangle$ but $\mathrm{retr}(l2) = \langle l2.E \rangle \,\|\, \mathrm{retr}(l2.PTR)$. This contradicts the assumption that $\mathrm{retr}(l1) = \mathrm{retr}(l2)$.

Next, let $l1 \neq \mathrm{NIL}$ and $\mathrm{retr}(l1) = \mathrm{retr}(l2)$ for some l2 in List1. Furthermore, suppose that for each linked sublist l1' of l1, if $\mathrm{retr}(l1') = \mathrm{retr}(l2')$, where l2' is a linked sublist of l2, then $l1' = l2'$. We note that $l2 \neq \mathrm{NIL}$ since $l2 = \mathrm{NIL}$ implies that $\mathrm{retr}(l2) = \langle\,\rangle$, in which case $\mathrm{retr}(l2) \neq \mathrm{retr}(l1)$. Therefore $\mathrm{retr}(l2) = \langle l2.E \rangle \,\|\, \mathrm{retr}(l2.PTR)$. We also have $\mathrm{retr}(l1) = \langle l1.E \rangle \,\|\, \mathrm{retr}(l1.PTR)$. Since $\mathrm{ret}(l1) = \mathrm{ret}(l2)$, $\langle l1.E \rangle = \langle l2.E \rangle$ and $\mathrm{retr}(l1.PTR) = \mathrm{retr}(l2.PTR)$. 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 $\mathrm{retr}(l1) = l$. Define invretr as:

$$\mathrm{invretr}(l) \equiv l1.$$

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 $\mathrm{retr}(l1) = l$. Assume furthermore that $l1 \neq \mathrm{NIL}$ and that $l1.E = e$. We define the linked list formed from the element l1.E as follows:

$$\langle l1.E \rangle \equiv \mathrm{invretr}(\langle \mathrm{hd}(l) \rangle).$$

In particular, $\langle \mathrm{hd}(l1) \rangle = \mathrm{invretr}(\langle \mathrm{hd}(l) \rangle)$. 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

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') $\equiv$ q1' = NIL.

post–ENQUEUE1(q1,q1',e) $\equiv$ q1' = q1 $\|$ $<e>$.

pre–DEQUEUE1(q1) $\equiv$ q1 $\neq$ NIL.
post–DEQUEUE1(q1,q1',res) $\equiv$ q1' = q1.PTR and res = q1.E.

post–EMPTY1(q1,q1',b) $\equiv$ 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 = $<e>$ = 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

$$\begin{aligned}(l1 \| <e>).PTR &= (<e1, e2, ..., en> \| <e>).PTR \\ &= <e1, e2, ..., en, e>.PTR \\ &= <e2, ..., en, e> \\ &= <e2, ..., en> \| <e>.\end{aligned}$$

Let l1' = $<e2, ..., en>$. We observe that l1' $\in$ List1 and len(l1') = n – 1.

Lemma. Let $<e>$, $l1 \in List1$. Then $retr(l1 \mathbin{\|} <e>) = retr(l1) \mathbin{\|} <e>$.

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

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

$$retr(l1 \mathbin{\|} <e>) = retr(<(l1 \mathbin{\|} <e>).E> \mathbin{\|} retr((l1 \mathbin{\|} <e>).PTR).$$

We note that $l1.E = (l1 \mathbin{\|} <e>).E$ so that

$$retr(l1 \mathbin{\|} <e>) = <e'> \mathbin{\|} retr((l1 \mathbin{\|} <e>).PTR).$$

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

$$retr((l1 \mathbin{\|} <e>).PTR) = retr(l1' \mathbin{\|} <e>) = retr(l1') \mathbin{\|} <e>.$$

It follows that

$$retr(l1 \mathbin{\|} <e>) = <e'> \mathbin{\|} (retr(l1') \mathbin{\|} <e>).$$

But from the definition of the retrieve function

$$retr(l1) = <hd(l1)> \mathbin{\|} retr(l1.PTR).$$

Therefore, $retr(l1 \mathbin{\|} <e>) = retr(l1) \mathbin{\|} <e>$.

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

Proof. We use induction on len(l2). When len(l2) = 0 we have

$$retr(l1 \mathbin{\|} l2) = retr(l1 \mathbin{\|} <>) = retr(l1).$$

In List we have

$$retr(l1) \mathbin{\|} retr(l2) = retr(l1) \mathbin{\|} <> = retr(l1).$$

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

$$retr(l1) \mathbin{\|} retr(l2) = retr(l1) \mathbin{\|} (<hd(l2)> \mathbin{\|} retr(tl(l2)))$$

$$= (retr(l1) \mathbin{\|} <hd(l2)>) \mathbin{\|} retr(tl(l2)).$$

By the induction hypothesis and the previous lemma,

$$(retr(l1) \mathbin{\|} <hd(l2)>) \mathbin{\|} retr(tl(l2) = retr(l1 \mathbin{\|} hd(l2)) \mathbin{\|} retr(tl(l2)).$$

Since len(l2) = n, len(tl(l2)) = n − 1 so that we can use the induction hypothesis with l2' = tl(l2). It follows that

$$\text{retr}(l1 \mathbin{\Vert} \langle\text{hd}(l2)\rangle) \mathbin{\Vert} \text{retr}(\text{tl}(l2)) = \text{retr}((l1 \mathbin{\Vert} \langle\text{hd}(l2)\rangle) \mathbin{\Vert} \text{tl}(l2))$$

$$= \text{retr}(l1 \mathbin{\Vert} (\langle\text{hd}(l2)\rangle \mathbin{\Vert} \text{tl}(l2)))$$

$$= \text{retr}(l1 \mathbin{\Vert} l2).$$

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 :

da. (∀ q1 ∈ Queue1)(pre–OP(retr(q1),args) => pre–OP1(q1,args)).

ra. (∀ q1 ∈ Queue1)(pre–OP1(q1,args) and post–OP1(q1,args,q1',res) => post–OP(retr(q1),args,retr(q1'),res)).

da. (∀ q1 ∈ Queue1)(pre–INIT(retr(q1),args) => pre–INIT1(q1,args)).

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

ra. (∀ q1 ∈ Queue1)(pre–INIT1(q1,args) and post–INIT1(q1,args,q1',res) => post–INIT(retr(q1),args,retr(q1'),res)).

Proof. Since q1' = NIL we know that retr(q1') = <>.

da. (∀ q1 ∈ Queue1)(pre–ENQUEUE(retr(q1),args) => pre–ENQUEUE1(q1,args)).

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

ra. (∀ q1 ∈ Queue1)(pre–ENQUEUE1(q1,args) and post–ENQUEUE1(q1,args,q1',res) => post–ENQUEUE(retr(q1),args,retr(q1'),res)).

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

da. (∀ q1 ∈ Queue1)(pre–DEQUEUE1(retr(q1),args) => pre–DEQUEUE(q1,args)).

Proof. Since retr(q1) ≠ <>, q1 ≠ NIL.

ra. (∀ q1 ∈ Queue1)(pre–DEQUEUE1(q1,args) and post–DEQUEUE1(q1,args,q1',res) => post–DEQUEUE(retr(q1),args,retr(q1'),res).

Proof. We have q1 ≠ NIL and q1' = q1.PTR and res = q1.E. From the definition of the retrieve function, retr(q1) = <q1.E> ‖ retr(q1.PTR). Then retr(q1') = retr(q1.PTR) = tl(retr(q1)). Finally, res = q1.E = hd(retr(q1)).

da. (∀ q1 ∈ Queue1)(pre–EMPTY(retr(q1),args) => pre–EMPTY1(q1,args)).

Proof. This is immediate since the pre-conditions are both TRUE.

ra. $(\forall$ q1 $\in$ Queue1)(pre-EMPTY1(q1,args) and post-EMPTY1(q1,args,q1',res) => post-EMPTY(retr(q1),args,retr(q1'),res)).

Proof. We have q1 = q1' and (b <=> q1 = NIL). Since q1 = q1', retr(q1) = retr(q1'). But q1 = NIL implies that retr(q1) = <>. Therefore, b => q1 = NIL => retr(q1) = <>. Next, suppose that retr(q1) = <>. Since retr is 1 to 1, q1 = NIL => b. Therefore, b <=> (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".

```
Queuerep :: Heap: Ptr —> Noderep
        where Noderep :: ELT : Element
                   PTER : ^[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^.nextq := arrive;
    tail := arrive
  end; (* enqueue *)

function dequeue(var head, tail : qtr) : char;
  begin
    if head <> nil then
```

```
        begin
          dequeue := head^.data;
          head := head^.nextq;
          if head = nil then
            tail := nil
        end
  end; (* dequeue *)
```

References.

Jones, Cliff B., *Software Development : A Rigorous Approach*, Prentice-Hall International, Inc., London, 1980.