An example: specifications of queues

Uri Abraham *

March 26, 2014

Abstract

The main line of this course is to present two approaches to modeling concurrency: the one that is based on the notion of states (which is more common) and the other that is based on the notion of structures that interpret a predicate language signature. In this lecture we describe how queues can be specified in these two approaches.

1 Discussion

We will study and compare two types of specifications in this course, those based on the notion of state and those based on the notion of event. In today's lecture we will begin to understand the nature of these two approaches by considering a very simple example: the specification of a queue. We all know that a queue is a FIFO storage that enables enqueue and dequeue operations. It is possible that the enqueue and dequeue events are complex operations that extend in time, and a strict locking-out policy in which only one process at a time is allowed access to the queue may slow down the activity of the system. Therefore a concurrent queue in which concurrent dequeue and enqueue operations are possible is advantageous. In a concurrent queue a process may dequeue (remove) an item from the queue while another process is enqueuing (adding an item to the queue) and there is no need for a synchronized coupling between the two processes. When we think about such enqueue and dequeue operations that can happen concurrently, the notion of a state of a queue is unclear: what is the state of a queue while the enqueue (or dequeue) operation is still taking place? Later in our course, we will study different answers to this question, but in this lecture

*Departments of Mathematics and Computer Science, Ben-Gurion University. Mathematical Models for Concurrency course, 2014.

we do assume that each of the events is instantaneous and has such a short duration that we may assume that there is no overlapping between the different events. Even under this "atomicity" assumption, the specification of queues is not completely evident and there are marked differences between the two modes of specifications that we shall describe here. So at this stage we deal with serial queues, and only later in the course we shall return to the question of specifying concurrently operating queues.

Another important problem discussed in this lecture is the problem of types or sorts. We find that multi-sorted languages and structures are more natural and convenient for specifications, and the queue specification is an example of this convenience. Yet a strict typed approach is not without its problems. We will describe some of these problems and discuss ways to address them.

1.1 Specification of queues

When we specify queues we must have first a clear idea about their behavior, which may be expressed and explained in an intuitive and informal manner. This idea is intuitive and informal, but nonetheless it must be there: formalization cannot come out of nowhere and it must correspond to previous expectations. Sometimes formalization sharpens our previous informal understanding, or even alter it somewhat, but formalization without prior expectations is meaningless. To specify queues we must first decide which of the queue operations we want to include in our specification. Do we want to have the peek operation for example or the isEmpty function? For simplicity, we have here just the two basic enqueue and dequeue operations, and we rely on our intuitive understanding of the FIFO discipline. Usually, and surely in every real life application, queues have a limited capacity. Should we have this capacity as a parameter in our specification, or perhaps assume that the queue that we specify has an unbounded capacity? Again, for simplicity we opt for unbounded capacity, and hence all enqueue operations should be terminating: it is never the case that an enqueue has to fail because the queue is full. But what about dequeue operations executed on an empty queue? There are two options here which correspond to two possible types of queues. The first option is to require that all operation executions, enqueue and dequeue, are terminating, but that a dequeue operation may also return the "empty" token which signals to the user of the queue that it is empty. The second option is to have the possibility of non-terminating dequeue events. A dequeue on an empty queue just wait till some enqueue fills the queue, and if the queue remains empty forever then

the dequeue never terminates. (Or in case other processes always empty the queue before our dequeue operation executes, then that dequeue may remain non-terminating.) We prefer at this stage to assume that all operation executions are terminating, and hence we shall have an "empty" value in our specification.

Another very basic question is this: do we allow concurrent operation executions or not? For example, can a dequeue event be concurrent with an enqueue event? Surely we want to allow concurrency (after all the course is about concurrency), but in this first stage we want to simplify as much as possible our discussion, and hence we shall assume that the events are linearly ordered in the temporal precedence relation $<$. (Later on we shall study the Herlihy–Wing linearization theory which will allow us to relax this linearity assumption.)

The queue specification question. Suppose that we are given a (possibly infinite) sequence of events $e_0 < e_1 < \cdots$ where some of the events are *enq* events and some are *deq* events, and we are also given a function *Val* which assigns to each event a value in $Data \cup \{\emptyset\}$ such that the following holds.

1. To any *enq* event e, $Val(e) \in Data$ (this value is said to be the value *deposited on the queue*),
2. and to every dequeue events e $Val(e) \in Data \cup \{\emptyset\}$ (if $Val(e) \neq \emptyset$ then we say that $Val(e)$ is the value dequeued by e, and if $Val(e) = \emptyset$ we say that e *returned empty*).

So given such a sequence of dequeue and enqueue events with values, is it a legitimate queue sequence or not? Under what conditions can we say that such a sequence display a correct behavior of a queue?

We shall give two answers to this questions which are different but equivalent. The first answer relies on a "return" function γ defined over the events, and the second answer on functions that describe the state of the queue at each moment. These two answers correspond to the event-based and (respectively) to the state-based approach to concurrency, and a main aim of this course is to gain a better understanding of concurrency by comparing these approaches.

The completion problem. The queue specification question can be rephrased in a somewhat different manner. Suppose a sequence $\langle e_i \mid i \in \omega \rangle$ of events which we take to be infinite for simplicity (so ω is the set of natural numbers). Assume that these events are linearly ordered in time. First come e_0 then e_1 and so on. That is $e_i < e_{i+1}$ where $<$ is the temporal precedence relation: $x < y$ for events x and y means that event x terminates before

event y begins. Assume that each event is said to be either an enqueue or else a dequeue event. Assume that a *Val* function is defined on the enqueue events but not on the dequeue events. For every enqueue event e, *Val*(e) is some *Data* value that e adds to the queue. The completion problem is to extend the function *Val* on the dequeue events (so as to assign to every dequeue event d in the sequence a value *Val*$(d) \in$ *Data* $\cup \{\emptyset\}$) in such a way that the resulting sequence of values represent a (uniquely defined) correct queue behavior.

It is evident that an answer to the completion problem provides an answer to the specification problem as well. For if we have to determine whether a sequence of enqueue and dequeue events with values is correct, we can remove for a moment the values of the dequeue events and obtain a sequence with values attached only to the enqueue events, and then we use the unique answer to the completion problem and check if the completion of values on the dequeue events is identical to the given values on these events.

So we shall deal here with the completion problem, and as we have said above, we are going to give two answers and then prove their equivalence. The first answer is called "event based" or "functional based", and the second answer is "state based". Later (in Section 2) we shall cast the function based specifications in the language of Tarskian model theory, and this will show the usefulness of notion of multisorted Tarskian structures.

1.1.1 The functional approach to the completion problem

Let $\langle e_i \mid i \in \omega \rangle$ be a sequence of enqueue/dequeue events, and suppose that *Val* is defined over the enqueue events in the sequence. The functional approach to the completion problem is based on the possibility of defining a function γ which relates any dequeue event of value v to that enqueue event that deposited v. We shall define for every dequeue event e_i its value *Val*(e_i) which is either a *Data* value or the "empty" token $\emptyset$, and when *Val*$(e_i) \neq \emptyset$ we shall also define $\gamma(e_i) = e_j$ where $j < i$ is such that e_j is an enqueue event that deposited the value removed by e_i. The definition is by induction on i. So suppose that these functions (*Val* and γ) are already defined on $\{e_{i'} \mid i' < i\}$ and we face now e_i which is a dequeue event. Let $F(i)$ be the set of all enqueue events e_j for $j < i$ that are not of the form $\gamma(e_k)$ for e_k a dequeue event with $k < i$. That is, $F(i)$ is the set of enqueue events that precede e_i that were not already been dequeued when e_i was about to begin.

1. If $F(i) = \emptyset$ then we define *Val*$(e_i) = \emptyset$. The intuition is that when $F(i)$ is empty then the queue is empty because all items deposited by

enqueue events that precede e_i were already been removed before e_i began. We say in this case that e_i is an "empty dequeue event". If e_i is an empty dequeue event then e_i is not in domain of γ.

2. If $F(i) \neq \emptyset$, then let $e_{j_0} \in F(i)$ be with minimal index j_0. Define $\gamma(e_i) = e_{j_0}$ and define $\mathit{Val}(e_i) = \mathit{Val}(e_{j_0})$. We say in this case that e_i is a "nonempty dequeue event".

What are the main properties of these functions γ and Val that we have just defined?

1. The domain of γ is a set of nonempty dequeue events, and if e_i is in the domain of γ then $\gamma(e_i)$ is an enqueue event e_j such that $j < i$ and $\mathit{Val}(\gamma(e_i)) = \mathit{Val}(e_i)$.

2. γ is one-to-one on its domain, and in fact γ is order preserving, that is if e_m and e_n are in the domain of γ and $m < n$, then $\gamma(e_m) < \gamma(e_n)$. To prove this, suppose that $m < n$ and e_n, e_m are nonepmty dequque events. We first note that the intersection of $F(n)$ with the set $\{e_j \mid j < m\}$ is a subset of $F(m)$ and that $\gamma(e_m) \notin F(n)$. This implies that $\gamma(e_m) < e_k$ for every $e_k \in F(n)$, and from this, $\gamma(e_m) < \gamma(e_n)$ follows as desired.

3. The range of γ is an initial segment of the enqueue events. That is if $e_m \in \text{range}(\gamma)$ and $n < m$ is such that e_n is an enqueue event, then e_n is also in the range of γ. Indeed, if $e_m = \gamma(e_i)$ then the choice of e_m as the minimum of $F(i)$ implies that e_n is in the range of γ (and $e_n = \gamma(e_j)$ for some $j < i$).

We have restated these three properties in Figure 1. It seems to me that these properties are the mathematical expression of the intuitive understanding of the First Come First Served principle. So the answer to the queue specification question is the following. A sequence of events with values so that some of the events are considered to be enqueue events and some are considered to be dequeue events is a FIFO queue sequence if there exists a function γ so that the properties listed in Figure 1 hold.

1.1.2 The State Based approach to the queue specification question

The state of a queue is represented as a finite sequence of values. If s is such a sequence, then we write $s = \langle s_0, \ldots, s_{n-1} \rangle$ where n is the length of

1. The domain of γ is a set of dequeue events, and for each e_i in its domain $\gamma(e_i) = e_j$ is an enqueue event such that $j < i$ and $\mathit{Val}(e_j) = \mathit{Val}(e_i)$.

2. γ is order preserving and its range is an initial segment of the enqueue events.

3. For any dequeue event e_i the following are all equivalent.

   (a) e_i is in the domain of γ.

   (b) $\mathit{Val}(e_i) \neq \emptyset$.

   (c) There is an enqueue event e_n for $n < i$ which is not in the range of $\gamma \restriction \{e_j \mid j < i\}$.

Figure 1: Properties of the γ function that formalize the fifo relation between the dequeue and enqueue events.

s, and, when s is nonempty, we say that s_0 is the first member of s (it is at the "head" of s) and s_{n-1} is the last member, said to be at its "tail".

As in the previous section, we assume a sequence $e_0 < e_1 < \cdots$ of events, some of which are said to be enqueue events and some dequeue events, and a function *Val* is defined on the enqueue events with values in *Data* (values that are never the "empty" token $\emptyset$). And the completion problem asks to complete the *Val* function on the dequeue events in accordance with our understanding of a queue data structure. With this aim in mind, we shall define two state functions *Qbefore* and *Qafter* defined on the events (on both the enqueue events and the dequeue events) with the intention that for every event e $\mathit{Qbefore}(e)$ is the state of the queue when e is about to begin, and $\mathit{Qafter}(e)$ is the queue when e terminates. So the values of these functions are finite sequences of data values. The definition of these functions on e_i is by induction on i.

1. For $i = 0$, $\mathit{Qbefore}(e_0) = \emptyset$. That is the queue is initially empty.

2. Suppose that $\mathit{Qbefore}(e_i)$ is defined. We shall define $\mathit{Qafter}(e_i)$, and then automatically set $\mathit{Qbefore}(e_{i+1}) = \mathit{Qafter}(e_i)$ and continue. The definition of $\mathit{Qafter}(e_i)$ depends on e_i.

   (a) If e_i is an enqueue event and $a = \mathit{Val}(e_i)$ its value, then we set $\mathit{Qafter}(e_i) = \mathit{Qbefore}(e_i)^\frown\langle a\rangle$. That is, $\mathit{Qafter}(e_i)$ is obtained by concatenating the value a onto the sequence $\mathit{Qbefore}(e_i)$. So if

Qbefore$(e_i) = (a_0, \ldots, a_{n-1})$ then *Qafter*$(e_i) = (a_0, \ldots, a_{n-1}, a_n)$ where $a_n = a$.

(b) If e_i is a dequeue event, there are two cases. In case *Qbefore*$(e_i) = \emptyset$, we define *Val*$(e_i) = \emptyset$, and *Qafter*$(e_i) = \emptyset$. In case *Qbefore*$(e_i) = (a_0, \ldots, a_{n-1})$ is non-empty, we define *Val*$(e_i) = a_0$ and *Qafter*$(e_i) = (a_1, \ldots, a_{n-1})$.

So, if we follow the state based approach, then a sequence $e_0 < e_1 < \cdots$ of dequeue/enqueue events with values is a queue sequence if there exist functions *Qbefore* and *Qafter* so that the following hold:

1. For every i, *Qbefore*(e_i) and *Qafter*(e_i) are finite sequences of elements of *Data* type.
2. *Qbefore*$(e_0) = \emptyset$, and *Qafter*$(e_i) =$ *Qbefore*(e_{i+1}) for every i.
3. If e_i is an enqueue event then *Qafter*$(e_i) =$ *Qbefore*$(e_i)^\frown\langle$ *Val*$(e_i)\rangle$.
4. If e_i is a dequeue event then there are two cases.
   (a) If *Qbefore*$(e_i) = \emptyset$ then *Val*$(e_i) = \emptyset$ and *Qafter*$(e_i) = \emptyset$.
   (b) If *Qbefore*$(e_i) \neq \emptyset$ and a is the first member of the sequence *Qbefore*(e_i), then *Val*$(e_i) = a$ and $\langle a\rangle^\frown$ *Qafter*$(e_i) =$ *Qbefore*(e_i).

To prove that the event-functional based and the state based approaches are equivalent we need the following theorem.

Theorem 1.1 *Let $e_0 < e_1 < \cdots$ be a sequence of enqueue/dequeue events, and suppose that Val is a function defined over the enqueue events and into the Data set. Then the completion of Val over the dequeue events is the same if done with the event based definition or the state based definition.*

Exercise 1.2 *Prove this theorem.*

Hints: To prove Theorem 1.1 we assume an infinite sequence $e_0 < e_1 < \cdots$ partitioned into enqueue and dequeue events, and suppose that *Val* is a function defined over the enqueue events in this sequence with values in the *Data* set. We denote with Val_1 and Val_2 the functions that the γ and state based completions respectively yields. So Val_1 and Val_2 are defined over the dequeue events and have values in $Data \cup \{\emptyset\}$. We have to prove that for every dequeue event e_i $Val_1(e_i) = Val_2(e_i)$. Recall that the γ completion consists in defining by induction on i (whenever e_i is a dequeue event) its

value $Val_1(e_i)$ as well as $\gamma_1(e_i)$ when e_i is in the domain of γ. For this we defined

$$F(i) = \{e_j \mid j < i \wedge enq(e_j) \wedge \forall k < i\ (deq(e_k) \wedge e_k \in \mathrm{dom}(\gamma) \rightarrow\ \gamma(e_k) \neq e_j)\}.$$

Let $i_0 < \cdots, i_{n-1}$ be an enumeration in increasing order of the indices of members of $F(i)$. (So $F(i) = \{e_{i_0}, \ldots, e_{i_{n-1}}\}$.) and let $values(F(i))$ be the sequence of *Data* values $(Val(e_{i_0}), \ldots, Val(e_{i_{n-1}}))$. (In case $F(i) = \emptyset$, $values(F(i)) = \emptyset$ as well.) The main ingredient in the proof of Theorem 1.1 is the proof (by induction on i) that $values(F(i)) = Qbefore(e_i)$.

1.2 Discussion

We have seen two ways to specify in a mathematical manner the notion of serial queue: the functional-event and the state based formalizations. Any formalization starts with some informal notion and moves to a more rigorous definition ("explication" to use Carnap's terminology) which is amenable to mathematical discussion. The notion of queue involves at the intuitive level two related ideas which are nonetheless quite distinctive. The first idea is that a queue is a fifo storage which means that the objects are removed in an orderly manner: an object is removed from the queue only after all previously enqueued objects have been removed. This idea leads to the definition of the removal function γ and to its properties. The second intuitive idea is that the elements of the queue are kept in a sequence where objects are added at the tail and removed from the head of the queue. This second idea leads to the definition of the queue states and the functions *Qbefore* and *Qafter* which give the state of the queue before and after an event e_i. It may seem that the definition that employs the γ function is more abstract than the one that employs an explicit queue state, and this explains perhaps why it is easier to learn about queues by the state approach. Yet we have seen that these two formalizations are equivalent (Theorem 1.1), and hence we are free to use whichever definition seems to be more suitable for a particular application we have in mind. In my experience, it is often the case that in correctness proofs when we want to prove that a certain algorithm implements a queue, then the γ approach is handier and easier to use.

In the following section we want to show how the notion of " multi-sorted structure" that was defined in the logic part of our lectures can help. We will take again the γ based definition of queues, but now cast it in the terminology of some formally defined language and structures.

2 Formal specifications of queues

In this part of our lecture we use a first-order predicate language to express properties of queues. Then we will use this language to give a formal specification of fifo queue. In order to define the language we first define its (multi-sorted) signature: the "L_q signature".

Sorts: There are two sorts: *Event* and *Data*.

Constants: There is a constant $\perp$ (called "bottom" or "undefined" –it will be used to "fill up" partially defined functions). Another constant is "empty".

Variables: We reserve variables for the different sorts. For example a, b, c, d, x, y (possibly with indices) are variables over sort *Event*. (Feel free to change or add variables in your exercises.)

Predicates: A binary predicate $<$ is defined on sort *Event*. (When $a < b$ we say that a precedes b.) A unary predicate *terminating* is defined on *Event*. Additional unary predicates in the basic signature are: *enq* and *deq* that are defined on sort *Event*.

Functions: We have two functions in the signature. $Val : Event \rightarrow Data \cup \{$ "empty"$\}$, and $\gamma : Event \rightarrow Event \cup \{\perp\}$.

We will discuss next the following issues:

1. The intended intuitive meaning of the symbols in the signature.
2. The language formation rules by which the language itself is defined.
3. The possible interpretations of the signature.
4. The properties that can be expressed in the L_q language and its extensions.

Based on the results of the previous section, we intend to model the situation in which enqueue and dequeue operations are sequentially executed on a queue. Every event is terminating. When the queue is empty, a *deq* operation will return the "empty" token value rather than a proper *Data* value. And we assume, for simplicity, queues with unbounded capacity and so there is nothing to prevent a termination of an enqueue operation. If $enq(e)$ then $Val(e)$ is the *Data* value enqueued by e onto the tail of the

queue. For a *deq* event e, $Val(e) \in Data$ is the value returned by e; it is "empty" when the queue is empty and it is the *Data* value at the head of the queue otherwise. We have seen in the previous section that this state based intuition that we have on the queue is equivalent to the functional fifo intuition which is expressed by means of a function γ, and the L_q language is designed to formally express this intuition.

2.0.1 The L_q language and its interpretations

In a single-sorted signature there are no sorts, and all objects in the intended universe are of the same type. The advantage of this type of logic is its simplicity. If, for example, F is a function symbol in a single-sorted signature and M is an interpretation of that signature, then F^M (the interpretation of F) is a complete function. That is, to say, a function defined over all the universe of M. In a multiple-sorted signature, however, functions and predicates are related to specific sorts. For example, the unary predicate *terminating* is defined over the *Event* sort (*terminating*(e) says that event e is of bounded duration), but to say that a data value is terminating is just meaningless. Thus one may wish to distinguish between meaningful and meaningless terms and formulas. Let's try to do it in order to discovers the problems that this distinction brings about. So one would define the terms of the language by induction, and associate with each term its sort (*Event* or *Data*). The idea is that if $\tau_1, \ldots, \tau_n$ are terms with associated sorts $S_1, \ldots, S_n$ and if F is some function symbol of sort $(S_1, \ldots, S_n; T)$ (namely F has arity n and it accepts parameters of sorts $S_1, \ldots, S_n$ and it returns objects of sort T) then $F(\tau_1, \ldots, \tau_n)$ is a term of sort T. When the definition of terms is done, then atomic formulas can be defined. If P is any predicate of sort $(S_1, \ldots, S_n)$ and τ_i are terms of sort S_i (for $1 \leq i \leq n$) then $P(\tau_1, \ldots, \tau_n)$ is a meaningful atomic formula. But if some τ_i is not of sort S_i then this is not an atomic formula. Finally formulas can be defined as explained in previous lectures using logical connectives and quantifiers.

A marked advantage of multi-sorted languages is that they allow in a very natural way partially defined functions. That is, if a function F is supposedly defined only on objects of sort A for example, then the signature states this assumption and then if a term $F(\tau)$ is legitimately formed then we know for sure that τ is of type A.

When we try to apply this approach to the L_q language, we immediately reach a problem. Although the function symbol γ is to be defined on the *Event* sort, its intended domain is a subset of the collection of *deq* events rather than all events. So if e is a variable of sort *Event*, for example, then

$\gamma(e)$ has a meaningful denotation in a structure M and assignment σ if and only if $\sigma(e)$ is a *deq* event in M so that the queue is not empty at e. There is no way to know this in advance and hence we cannot say in advance of any assignment whether $\gamma(e)$ is a meaningful term.

I do not see any "easy" solution to this problem, and the following is perhaps the best compromise. A special member is assumed in the universe of any structure, the "undefined" element $\perp$ whose interpretation $\perp^M$ is in the universe $|M|$ of any interpreting structure. The function γ^M is totally defined over all of $Event^M$, and $\gamma^M(a) \in Event^M \cup \{\perp^M\}$ for every $a \in Event^M$, but we have that $\gamma^M(a) \neq \perp^M$ if and only if $deq^M(a)$ *and* $Val(a) \neq$ "*empty*" holds in M.

Thus, if e is an *Event* variable then $\gamma(e)$ is a legitimate (well-formed) term, but we cannot know if it is meaningful (i.e. different from $\perp$) or not. Hence, for example, $Val(\gamma(e))$ is either a *Data* value (in case $\gamma(e) \neq \perp$) or else the undefined $\perp$ value.

Thus we realize that even multi-sorted logic may require the undefined $\perp$ value, and we are forced to admit partially defined functions such as γ into multi-sorted logic. Hence, one may argue, let's return to the simpler single-sorted paradigm, since there is not much of a difference between having the S_i's as sorts and having them as predicates over a uniform universe. Still we may want to salvage some of the obvious advantages of many sorted languages, their flexibility and naturalness of expression. So we suggest the following. Let's keep the sorts, the sort specific variables, and the possibility of specifying functions and predicates on specific sorts, but we will also define the set $\{a \in |M| \mid F^M(a) \neq \perp^M\}$ as the "virtual domain of F" (as opposed to its formal domain which is all of $|M|$ or all members of some specific sort). If we denote the virtual domain of F by $\mathrm{dom}(F)$, then $x \in \mathrm{dom}(F)$ has the same meaning as $F(x) \neq \perp$. For example, the formal domain of the function γ is the set of all events (namely the sort *Event*), but $\mathrm{dom}(\gamma)$ is the set of all dequeue events e so that $Val(e) \neq$ "*void*". Now we can write for example $\forall e \in \mathrm{dom}(\gamma)(Val(e) = Val(\gamma(e))$. The meaning of such a sentence is that if the terms in the equation (here $\gamma(e)$) are defined then the two sides of the equation are equal.

To sum-up our discussion, we can define now an interpretation of the L_q signature to be a structure $\mathcal{M}$ consisting of the following features.

1. The universe of $\mathcal{M}$ is a non-empty set M. For each sort S of the signature, $S^{\mathcal{M}} \subseteq M$ is the interpretation of sort S by $\mathcal{M}$. So $Event^{\mathcal{M}}$ is the set of events of $\mathcal{M}$, and $Data^M$ is the interpretation of sort *Data*. Each constant is interpreted as a member of M, and in our case we

have $\perp^{\mathcal{M}} \in |M|$ (a value not in any sort).

2. For every n-ary predicate P in L_q, $P^{\mathcal{M}}$ is a collection of n-tuples from $|M|$. If the signature specifies the sorts of P then P^M respects this requirement. For example, $<^{\mathcal{M}} \subset \mathit{Event}^M \times \mathit{Event}^M$ is a binary relation on $\mathit{Event}^{\mathcal{M}}$, and $\mathit{terminating}^{\mathcal{M}} \subseteq \mathit{Event}^{\mathcal{M}}$.

3. Every n-ary function symbol F has an interpretation $F^{\mathcal{M}}$ which is an n-place function from $|M|$ to $|M|$. The signature gives some information on the virtual domain of the function. For example, the *Val* function has signature $(\mathit{Event}, \mathit{Data})$. This means that if $a \in \mathit{Event}$ then $\mathit{Val}^{\mathcal{M}}(a) \in \mathit{Data}$. But the γ function has signature $(\mathit{Event}, \mathit{Event} \cup \{\perp\})$ which means that on some events e $\gamma(e) = \perp$ is a possibility, and so the signature alone does not completely specify the domain of this function and additional information is needed.

Recall that we defined for any structure M that interprets a signature the notion of assignment. An assignment is a function σ from the set of variables into $|M|$ so that for every variable v of sort S $\sigma(v) \in S^M$. That is, each variable is given a value in its sort, and if the variable is general (not attached to any specific sort) then its value can be any member of the universe of M. Then we defined $\mathit{Val}_M(\tau, \sigma) \in |M|$ for every term τ. $\mathit{Val}_M(\tau, \sigma) = \perp^M$ is possible. Then for every formula φ we define $\mathit{Val}_M(\varphi, \sigma)$ to be either true or false.

So unlike terms which may acquire an undefined value $\perp$, a formula is either true or false (given any assignment). For example, if P is a predicate of sort S (which means intuitively that it would be meaningless to apply this predicate on objects not in sort S) and if τ is a term so that $\mathit{Val}_M(\tau, \sigma)$ is $\perp$ (is not in sort S^M) then $\mathit{Val}_M(P(\tau), \sigma) = \mathsf{false}$ (rather than "undefined").

Clearly not every interpretation of the L_q signature displays a behavior that can be expected from a queue. For example, there is nothing in the signature that ensures that the value of a dequeue event e is the same as the value of its corresponding enqueue $\gamma(e)$. In the following subsection we will deal with the question of expressing properties of the queue in the L_q language.

3 Specifying queues in the γ language

It is important to learn how properties expressed in natural language (such as English) can be formally expressed in the γ queue language that we specified

above. We will give below examples of properties that are expressed first in English and then in the formal γ language.

1. All events are terminating:

$$\forall x \in Event(terminating(x)).$$

2. Relation $<$ is a linear ordering on the events. This sentence has two parts. We assume that x, y, z are variables of sort *Event*.

   (a) $<$ is transitive and irreflexive:

$$\forall x, y, z((x < y \wedge y < z \rightarrow x < z) \wedge \neg\, x < x) \tag{1}$$

   and

   (b) linearity:

$$\forall x, y(x < y \vee x = y \vee y < x)$$

   .

3. No event is both a dequeue and an enqueue event:

$$\forall x(enq(x) \rightarrow \neg(deq(x)).$$

4. Every *enq* event has a proper *Data* value.

$$\forall x(enq(x) \rightarrow Val(x) \in Data \setminus \{\text{“}void''\}).$$

The following formula $isEmptyAt(e)$ says that the queue is empty at the beginning of event e. The idea is that to express the emptiness of the queue at e we say that for every enqueue event a that precedes e there is a dequeue events b that precedes e as well and so that b obtained the item deposited by a on the queue. So no events are left in that queue when e begins; all were already removed.

$$\forall a(enq(a) \wedge a < e \rightarrow \exists b\ (deq(b) \wedge b < e \wedge \gamma(b) = a)).$$

We also define $isNonEmptyAt(e) \equiv \neg\ isEmptyAt(e)$.

The specification of queues in the γ L_q language is given in Figure 2.

In our specification (Figure (2)) we require that the events are linearly ordered by the $<$ relation, but we actually want the sort Event is interpreted in our structures as either a finite set of a set that has the order-type of natural numbers. There is no way, however, to write down a formula in our language (or in any first-order language) that ensures this.

1. All events are terminating.

$$\forall e(e \in \mathit{Event} \rightarrow \mathit{terminating}(e)).$$

2. Relation $<$ is a linear ordering on the events. This sentence has two parts.

   (a) $<$ is transitive and irreflexive. This is equation (1).

   (b) linearity:

$$\forall x, y(x < y \vee x = y \vee y < x)$$

   .

3. No event is both a dequeue and an enqueue event.

4. Every *enq* event has a proper *Data* value.

$$\forall x(\mathit{enq}(x) \rightarrow \mathit{Val}(x) \in \mathit{Data} \ \wedge \mathit{Val}(x) \neq \text{“empty”}).$$

5. The function γ is defined on the *deq* events d such that $\mathit{isNonEmptyAt}(d)$, and for every such *deq* event $\gamma(d)$ is an enqueue event that precedes d and has the same value as d:

$$\forall a(\gamma(a) \neq \perp \leftrightarrow (\mathit{deq}(a) \wedge \mathit{isNonEmptyAt}(a))$$
$$\forall a(\gamma(a) \neq \perp \rightarrow \mathit{enq}(\gamma(a)) \wedge \gamma(a) < a \wedge \mathit{Val}(\gamma(a)) = \mathit{Val}(a)).$$

6. For every dequeue event d, if $\mathit{isEmptyAt}(d)$ then $\mathit{Val}(d) =$ “empty”, but if $\mathit{isNonEmptyAt}(d)$ then $\gamma(d)$ is the first enqueue event that has not already been dequeued.

Figure 2: Linear queue specification with the γ return function.

4 A functional stack specification

In this exercise we want to specify linear stacks in a way that is similar to the queue specification with the γ function that we have seen. A stack is a data structure that operates under the "last-in-first-out" regime. Our stack supports two actions: push and pop. Visually, we think that a stack has just one opening (its top) so that it is always the topmost object that a pop action removes from the stack, and any object that a push action adds to the stack is added on its top. We assume that the stack has an unlimited capacity, so that all push actions are terminating. A pop action executed on an empty stack returns the token "empty". In the following specification the function γ is partially defined over the *Event* sort. For every pop event a executed on a non-empty stack, $\gamma(a)$ is that push event whose value a returns. The language in which we specify our stack is defined first.

Sorts: There are two sorts: *Event* and *Data*.

Constants: There is a constant $\perp$ (called "bottom" or "undefined" –it will be used to "fill up" partially defined functions). Another constant is "empty".

Variables: We reserve variables for the different sorts. For example a, b, c, d, x, y (possibly with indices) are variables over sort *Event*. (Feel free to change or add variables in your answer.)

Predicates: A binary predicate $<$ is defined on sort *Event*. (When $a < b$ we say that a precedes b.) A unary predicate *terminating* is defined on *Event*. Additional unary predicates in the basic signature are: *Pop* and *Push* that are defined on sort *Event*.

Functions: We have two functions in the stack signature.

1. $Val : Event \rightarrow Data \cup \{$ "empty"$\}$,
2. $\gamma : Event \rightarrow Event \cup \{\perp\}$.

Exercise 4.1 *Write a formal specification of the stack in this language.*