Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 201 (2008) 69–98
www.elsevier.com/locate/entcs

Trace-based Derivation of a Lock-Free Queue Algorithm

Lindsay Groves[1]

School of Mathematics, Statistics and Computer Science
Victoria University of Wellington, Wellington, New Zealand

Abstract

Lock-free algorithms have been developed to avoid various problems associated with using locks to control access to shared data structures. Instead of preventing interference between processes using mutual exclusion, lock-free algorithms must ensure correct behaviour in the presence of interference. While this avoids the problems with locks, the resulting algorithms are typically more intricate than lock-based algorithms, and allow more complex interactions between processes. The result is that even when the basic idea is easy to understand, the code implementing lock-free algorithms is typically very subtle, hard to understand, and hard to get right.
In this paper, we consider the well-known lock-free queue implementation due to Michael and Scott, and show how a slightly simplified version of this algorithm can be derived from an abstract specification via a series of verifiable refinement steps. Reconstructing a design history in this way allows us to examine the kinds of design decisions that underlie the algorithm as describe by Michael and Scott, and to explore the consequences of some alternative design choices.
Our derivation is based on a refinement calculus with concurrent composition, combined with a reduction approach, based on that proposed by Lipton, Lamport, Cohen, and others, which we have previously used to derive a scalable stack algorithm. The derivation of Michael and Scott's queue algorithm introduces some additional challenges because it uses a "helper" mechanism which means that part of an enqueue operation can be performed by any process, also in a simulation proof the treatment of dequeue on an empty queue requires the use of backward simulation

Key words: Lock-free algorithm, access control, mutual exclusion, refinement, scalable stack algorithm

1 Introduction

Increasing use of concurrent software designs has highlighted many problems associated with the use of locks and mutual exclusion, which have in turn led

[1] Email: lindsay@mcs.vuw.ac.nz

doi:10.1016/j.entcs.2008.02.016

to the development of *lock-free* algorithms to implement concurrent data structures. Rather than avoid interference using mutual exclusion, lock-free algorithms allow interference to occur and, when it does, detect it using strong synchronisation primitives such as Compare and Swap (CAS). The most widely used such algorithm is believed to be Michael and Scott's queue algorithm [18] which has been included in the standard Java concurrency library as of version 1.6. Although the basic ideas underlying this algorithm are quite simple, the detail of the implementation is surprising subtle. Several authors have presented verifications of this algorithm, or minor variants thereof, but these tend to be either too informal to be convincing or too detailed to be enlightening.

In this paper, we attempt to give a more intelligible presentation of Michael and Scott's queue algorithm by showing how it (or rather the version included in the Java concurrency library, which assumes the existence of automatic garbage collection) can be derived from an abstract specification via a sequence of verifiable refinement steps. Our aim in doing this is to try to understand the kinds of design decisions that would lead to the construction of such an algorithm. We do not claim that the algorithm was originally constructed in anything like this way, or that this approach would allow us to discover new algorithms — rather we aim to identify the important design choices that must have been made at some point (even if not consciously). This also allows us to identify alternatives that might have been considered, and explore the variations in the algorithm that these would lead to. We do not attempt to derive the algorithm entirely from "first principles" — in order to keep the discussion to a reasonable length, we make use of prior knowledge of data structures, common patterns in the design of lock-free algorithms, and knowledge of the choices that Michael and Scott made.

We are primarily concerned with showing that the algorithm is linearisable with respect to an abstract specification of a concurrent queue; we will also argue that the algorithm is lock-free. *Linearisability* [12] is the standard safety property for concurrent data structures, and requires that each operation appears to occur atomically at some point between its invocation and its response, and correctly implement its abstract specification. *Lock-freedom* is a liveness property which ensures that the system as a whole makes progress, even though individual operations may never terminate. More precisely, a system is lock-free if some operation will always complete within a finite number of steps of the system. [2] This property precludes the use of locks and guarantees freedom from deadlock and livelock, but not individual starvation.

[2] Some authors call this property *nonblocking*; others use nonblocking as a more general term encompassing other progress conditions such as wait-freedom and obstruction-freedom [11].

We express our algorithms in a language based on the refinement calculus, with procedures and type declarations [20], and parallel composition [3]. Our procedures use **in**, **out** and **in out** parameters, as in Ada; we also use value-returning procedures and name the return value so that it can be constrained in specification statements. In reasoning about linearisability we are not concerned with termination, so a specification statement $w :[\, R \,]$ is required to establish postcondition R, modifying only variables in w, only if the statement terminates. In writing specifications and invariants, we use Z's mathematical notation [26]; in particular, seq T and iseq T are the sets of sequences and injective sequences over T, $\#s$ is the length of the finite sequence s, $A \nrightarrow B$ is the set of partial functions from A to B, $\operatorname{dom} f$ is the domain of function f, and $f \oplus \{x \mapsto y\}$ is the function which is the same as f except at x where its value is y.

We assume a trace semantics similar to that used in [3], except that (following [17]) we define an *execution* to be an alternating sequence of states and actions, starting with a state, and a *trace* to be the sequence of observable actions in an execution. Our notion of refinement is *preservation of linearisability* — at each step in our development, we show that every execution of the lower level model can be transformed into an equivalent execution of the higher level model which preserves the order of non-concurrent operations [7]. We do this in two steps: one showing that for every concurrent execution of the lower level model, there is an equivalent execution in which each queue operation is executed without interruption (this is often called *atomicity*); and one showing that when executed without interruption, the lower level model correctly implements the higher level model. In this way, we separate the sequential aspects of the algorithm (in this case, the basic queue implementation) from the concurrent aspects (dealing with interaction between processes). The sequential aspects are mostly quite straightforward and we do not justify them in detail. We are more concerned with showing how to handle the concurrent aspects, but in the interests of space (and intelligibility), we also justify these aspects in a fairly informal way, reasoning about traces directly, rather than using specific proof rules for refinement. In showing that every concurrent execution can be reduced to an equivalent concurrent one, we mostly use Lipton's theory of left and right movers to rearrange the steps in an execution, however, some aspects of Michael and Scott's algorithm require this to be extended with the ability to deleted or modify some steps.

We begin in Section 2 by presenting an abstract specification for a system involving a shared queue, then in Section 3 we introduce the concrete data structure used to represent the queue and derive concrete specifications for the queue operations in terms of this representation. The main body of the paper

is in Sections 4 and 5, where we derive lock-free implementations of the queue operations, which are essentially the same as those of Michael and Scott. We could begin by considering either operation first — we will start with *enqueue*, since it reveals the major issues that arise in designing the implementation, and these have consequences which affect the design of *dequeue*. Finally, in Section 6 we compare our version of the algorithm with Michael and Scott's, and discuss some other possible variables, and in Section 7, we draw some conclusions and discuss related and future work.

2 Specifying an Abstract Lock-Free Queue

We consider a system consisting of a finite set of concurrent processes which access a shared queue with elements of some type T. Each process occasionally performs an operation on the queue, and otherwise performs actions which do not involve the queue. To model such a system, we abstract away from its other behaviour and just consider its queue operations.

2.1 An abstract concurrent queue

At the most abstract level the behaviour of a system involving a concurrent queue is described by a set of processes each performing a non-deterministically chosen sequence of queue operations (see Figure 1). Since we have abstracted away from the rest of the program, which would otherwise provide values to be enqueued and use values that are dequeued, the values to be enqueued onto the queue are chosen non-deterministically, whereas values returned by *dequeue* are determined by the contents of the queue at the time and are then discarded. We write $||_{p \in \mathcal{P}}\ op_p$ for the parallel composition of processes drawn from a finite set $\mathcal{P}$, each executing an operation op_p, and usually omit the p subscript when the operation does not depend on p.

$$
\begin{array}{l}
\mathit{QUEUE} \mathrel{\widehat{=}} \\
\quad \textbf{var}\ q : \mathit{Queue} := \mathit{EmptyQueue}\ ; \\
\quad \Big|\Big|_{p \in \mathcal{P}} \left(\begin{array}{l} \textbf{var}\ y : T_\perp\ ; \\ \textbf{do}\ \mathit{true} \rightarrow ([]_{x:T}\ \mathit{enqueue}_p(x)) \ []\ \mathit{dequeue}_p(y)\ \textbf{od} \end{array} \right)
\end{array}
$$

Fig. 1. Abstract specification for a system involving a concurrent queue

All of our programs will have this high level structure, but will use different versions of *enqueue* and *dequeue*.

In our initial model, we regard *enqueue* and *dequeue* actions as being atomic, so a trace is a sequence of *enqueue* and *dequeue* actions which is valid according to the semantics of these queue operations, and this queue is clearly

linearisable. We specify the queue operations using a model-based approach, treating an abstract queue as a sequence of values of its component type (i.e. $Queue \mathrel{\widehat=} \text{seq } T$). Thus, *enqueue* and *dequeue* are defined as shown in Figure 2, and an empty queue is represented by the empty sequence, $\langle\rangle$. Note that a *dequeue* on an empty queue returns a distinguished value, $\bot \notin T$, and the result type for *dequeue* is $T_\bot \mathrel{\widehat=} T \cup \{\bot\}$, since in a lock-free implementation *dequeue* cannot wait for the queue to become non-empty.

$$enqueue(\textbf{in } x : T) \mathrel{\widehat=} \quad q : \left[q = q_0 \frown \langle x \rangle \right]$$

$$dequeue(\textbf{out } y : T_\bot) \mathrel{\widehat=} \quad q, y : \left[\begin{array}{l} q = q_0 = \langle\rangle \wedge y = \bot \ \vee \\ q_0 = \langle y \rangle \frown s \end{array} \right]$$

Fig. 2. Abstract specification for queue operations

2.2 Assumptions about the implementation

We wish to construct a lock-free implementation of a concurrent queue, assuming that shared variables can be access using atomic load, store and CAS operations. A CAS instruction tests whether a shared location *loc* has some expected value *old*, and if so atomically updates *loc* to a new value *new* and *succeeds*, returning *true*; otherwise it *fails*, returning *false* and leaving *loc* unchanged. This is specified formally as:

$$CAS(\textbf{in out } loc, \textbf{in } old, \textbf{in } new)\ r : boolean \mathrel{\widehat=} \quad loc, r : \left[\begin{array}{l} loc_0 = old \wedge loc = new \wedge r \ \vee \\ loc_0 \neq old \wedge loc = loc_0 \wedge \neg\, r \end{array} \right]$$

We also assume a dynamic storage environment in which allocation of heap locations is atomic and lock-free. Since [18] does not consider the possibility of heap overflow, we assume that the heap is unbounded.

3 Representing the Queue

Following [18], we will represent the queue using a linked list with a dummy node at front of the list, along with *Head* and *Tail* pointers (see Figure 3). When the queue is empty, *Head* and *Tail* both point to the dummy node, and when the queue is non-empty, *Head* points to the dummy node, but its value is not part of the queue. At this stage we will assume that *Tail* points to the last node in the list.[3] Using a dummy node reduces the number

[3] We will see later that this assumption has to be relaxed, but that is an important decision in the design of the queue implementation, so we wish to show carefully why is is required rather than just assume it.

of special cases that need to be considered, since *Head* and *Tail* are always non-null, and reduces contention by ensuring that *Head* and *Tail* only point to the same node if the queue is empty.

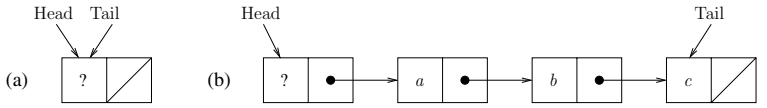

Fig. 3. Basic queue representation

3.1 Modelling the heap

To describe the queue representation formally, we first need to introduce a model for the heap, which — as far as we are concerned — only contains queue nodes. We model the heap as an infinite set of *locations*, of type *Loc*, containing a special value called *null*, and use *NLoc* to denote the set of non-*null* locations. We model fields using partial functions: *value*, from non-*null* locations to T; and *next*, from non-*null* locations to locations.

Thus, we have the following declarations (using Z's mathematical notation for types and predicates):

type Loc
const $null : Loc$
type $NLoc \mathrel{\widehat=} Loc \setminus \{null\}$
var $value : NLoc \pfun T$
var $next : NLoc \pfun Loc$
var $Head, Tail : NLoc$

Heap locations are allocated using a *newNode*() operation, specified as follows:

$$newNode()\ loc : NLoc \mathrel{\widehat=}$$
$$loc, value, next : \left[\begin{array}{l} \mathrm{dom}\ value \setminus \mathrm{dom}\ value_0 = \{loc\} \\ \mathrm{dom}\ next \setminus \mathrm{dom}\ next_0 = \{loc\} \end{array} \right]$$

Head, *Tail*, *value* and *next* are assumed to be encapsulated within a module, so they can be seen by all processes performing queue operations, but from nowhere else. Queue nodes (i.e. elements of *value* and *next*) can only be accessed via location values returned by *newNode*().

In writing programming constructs (tests and assignment statements), we use traditional field selection notation, and write, for example, *head.next* in expressions rather than *next*(*head*). We also use the following abbreviations for assignments to fields:

$$node.value := x \quad \equiv \quad value := value \oplus \{node \mapsto x\}$$
$$node.next := y \quad \equiv \quad next := next \oplus \{node \mapsto y\}$$

A CAS updating the *next* field of a node n must be defined specially, since it modifies the heap:

$$CAS(n.next, old, new) \quad \equiv \quad next, r : \left[\begin{array}{l} next(n) = old \land \\ \quad next = next_0 \oplus \{n \mapsto new\} \land r \ \lor \\ next(n) \neq old \land next = next_0 \land \neg\, r \end{array} \right]$$

3.2 State invariant

To be a valid representation of a queue, *value* and *next* must have identical domains, and the linked list must be connected and not contain cycles. We express this requirement as a state invariant, *Inv*, which postulates the existence of a sequence, f, of unique locations (not including *Head* or *null*) corresponding to the nodes in the linked list.[4] We define *Inv* in terms of a predicate *Rep*, which is also used below in defining the abstraction relation *Abs*:

$$Inv(Head, Tail, value, next) \quad \widehat{=}$$
$$\exists f : \mathrm{iseq}_1\, NLoc \bullet Rep(f, Head, Tail, value, next)$$

$$Rep(f, Head, Tail, value, next) \quad \widehat{=}$$
$$\mathrm{dom}\, value = \mathrm{dom}\, next \land next \circ (\langle Head \rangle \frown f) = f \frown \langle null \rangle \land Tail = last(\langle Head \rangle \frown f)$$

$Inv(Head, Tail, value, next)$ ensures that *Head* holds a pointer to the first node in the list, the *next* field of the last node is *null*, the *next* field of every node except the last points to the immediately following node, and that *Tail* holds a pointer to the last node in the list. Requiring f to be injective ensures that the list contains no cycles, and — along with the second conjunct of *Rep* — that *Head* cannot occur in f.

Using Z's data types, the composition $next \circ (\langle Head \rangle \frown f)$ has the effect of mapping the function *next* over the sequence $\langle Head \rangle \frown f$. The last two conjuncts of *Rep* can thus be expressed more directly as:

$$(f = \langle\rangle \Rightarrow next(Head) = null \land Tail = Head\,) \land$$
$$(f \neq \langle\rangle \Rightarrow next(Head) = f(1) \land next(last(f)) = null \land Tail = last(f)\,) \land$$
$$(\forall i : 1 \,..\, \#f - 1 \bullet next(f(i)) = f(i+1)\,)$$

however, the more compact version is more convenient for manipulation.

We define the relationship between the linked list and the abstract queue it represents using an abstraction relation, *Abs*, which is defined in a similar way:

$$Abs(q, Head, Tail, value, next) \quad \widehat{=}$$
$$\exists f : \mathrm{iseq}_1\, NLoc \bullet Rep(f, Head, Tail, value, next) \land value \circ f = q$$

[4] This use of f is analogous to introducing an auxiliary variable, and avoids the need to use recursive functions to reason about reachability or to assemble the abstract queue represented by a list.

Here, the last conjunct is equivalent to $\#f = \#q \wedge \forall\, i : \operatorname{dom} f \bullet \mathit{value}(f(i)) = q(i)$.

3.3 Initialisation

Let us investigate the conditions under which the heap represents the empty queue — this is required for initialisation, and also to detect an empty queue in *dequeue*:

$$
\begin{array}{ll}
 & \mathit{Abs}(\langle\rangle, \mathit{Head}, \mathit{Tail}, \mathit{value}, \mathit{next}) \\
= & \text{(Expand } \mathit{Abs} \text{ and } \mathit{Rep}\text{, substituting } \langle\rangle \text{ for } q) \\
 & \exists f : \operatorname{iseq}_1 \mathit{NLoc} \bullet \\
 & \quad \operatorname{dom} \mathit{value} = \operatorname{dom} \mathit{next} \wedge \mathit{next} \circ (\langle \mathit{Head} \rangle \frown f) = f \frown \langle \mathit{null} \rangle \wedge \\
 & \quad \mathit{Tail} = \mathit{last}(\langle \mathit{Head} \rangle \frown f) \wedge \mathit{value} \circ f = \langle\rangle \\
= & \text{(} \mathit{value} \circ f = \langle\rangle \text{ implies } f = \langle\rangle\text{; apply one point rule and simplify)} \\
 & \operatorname{dom} \mathit{value} = \operatorname{dom} \mathit{next} \wedge \mathit{next}(\mathit{Head}) = \mathit{null} \wedge \mathit{Tail} = \mathit{Head}
\end{array}
$$

This captures our intuitive description of the initial state where *Head* and *Tail* both point to the dummy node, and can be established using the following initialisation code:

$$
\begin{array}{l}
\mathit{Head} := \mathit{newNode}() \,; \\
\mathit{Head}.\mathit{next} := \mathit{null} \,; \\
\mathit{Tail} := \mathit{head}
\end{array}
$$

Ignoring the first conjunct ($\operatorname{dom} \mathit{value} = \operatorname{dom} \mathit{next}$), which is given by the state invariant, we can detect an empty queue by testing whether $\mathit{next}(\mathit{Head}) = \mathit{null} \wedge \mathit{Head} = \mathit{Tail}$ holds. In fact, it suffices to test either $\mathit{next}(\mathit{Head}) = \mathit{null}$ or $\mathit{Head} = \mathit{Tail}$, since both of these conditions imply $q = \langle\rangle$ when $\mathit{Abs}(q, \mathit{Head}, \mathit{Tail}, \mathit{value}, \mathit{next})$ holds.

3.4 Concrete queue operations

With this abstraction relation, we can now obtain specifications for the queue operations in terms of the linked list representation, using familiar techniques for data refinement. If we calculate concrete specifications using the techniques of Morgan and Gardiner [21] and Morris [22] we obtain very weak specifications which allow the entire queue representation to be reconstructed by each operation. This freedom is utilised by so-called *universal constructions*, which automatically convert a sequential data structure into a lock-free or wait-free one [9]. However, since we wish to obtain a more efficient implementation, we use stronger concrete specifications which retain the old queue representation and make as few changes as possible, as shown in Figure 4. We assume $\mathit{Inv}(\mathit{Head}, \mathit{Tail}, \mathit{value}, \mathit{next})$ as an implicit precondition and postcondition of these operations, and all operations derived from them.

$$enqueue(\mathbf{in}\ x : T) \mathrel{\widehat=} \quad Tail, value, next \mathrel{:} \left[\begin{array}{l} \exists\, n : NLoc \setminus \mathrm{dom}\, value \bullet \\ \quad value = value_0 \oplus \{n \mapsto x\} \wedge \\ \quad next = next_0 \oplus \\ \qquad \{Tail_0 \mapsto n, n \mapsto null\} \wedge \\ \quad Tail = n \end{array}\right]$$

$$dequeue(\mathbf{out}\ y : T_\perp) \mathrel{\widehat=} \quad Head, y \mathrel{:} \left[\begin{array}{l} Head_0 = Tail = Head \wedge y = \perp \ \vee \\ Head_0 \neq Tail \wedge \\ \quad Head = next(Head_0) \wedge \\ \quad y = value(next(Head_0)) \end{array}\right]$$

Fig. 4. Concrete specification for queue operations

It is straightforward to show that these are data refinements of the abstract specifications, either directly or by showing that they are operational refinements of the calculated specifications [21,22]. Notice that *enqueue* never changes *Head*, while *dequeue* never changes *value*, *next* or (at this stage) *Tail*.

This transformation does not introduce any concurrency problems, since it simply replaces abstract atomic actions by concrete ones. It therefore follows that it is a valid refinement (i.e. that linearisability is preserved), since for any trace that can be produced using the specifications in Figure 4, we can obtain an equivalent trace of *QUEUE* by replacing all occurrences of these specifications by the corresponding specifications given in Figure 1.

4 Refining enqueue

We now consider how to implement the *enqueue* operation as specified in Figure 4. Throughout this discussion we will assume that *dequeue* operations are performed atomically, as specified in Figure 4 — indeed, we can ignore *dequeue* operations at this stage, since allowing *dequeue* operations does not allow any state to be reached that cannot be reached using only *enqueue* operations.

4.1 Allocating a new node

Examining the specification of *enqueue*, the first thing we need to do is provide a witness for n. So, operationally, our first step is to create a new node. We declare a local variable, *node*, and split the specification for *enqueue* into a sequence: the first component (*NewNode*) allocates and initialises a new node; the second component (*AddNode*) performs the rest of *enqueue*, adding the new node to the list and updating *Tail*. Thus, the concrete specification for *enqueue* is refined to:

$$\begin{array}{l}\textbf{var } node : NLoc \textbf{ ;} \\ node, \\ value, \\ next \end{array} :\left[\begin{array}{l} node \in NLoc \setminus \mathrm{dom}\ value \wedge \\ value = value_0 \oplus \{node \mapsto x\} \wedge \\ next = next_0 \oplus \{node \mapsto null\} \end{array}\right] \textbf{ ;} \qquad (NewNode)$$

$$\begin{array}{l} Tail, \\ next \end{array} :\left[\begin{array}{l} next = next_0 \oplus \{Tail_0 \mapsto node\} \wedge \\ Tail = node \end{array}\right] \qquad (AddNode)$$

It is straightforward to show that this is a sequential refinement of the concrete specification for *enqueue*. To demonstrate atomicity, we show that for every concurrent execution containing a completed execution of this implementation of *enqueue*, there is an equivalent execution in which these steps are executed without interruption. We first observe that any execution containing a completed execution of *enqueue* by process p has the form $\alpha\ NewNode_p\ \beta\ AddNode_p\ \gamma$, where α, β and γ are any sequences of actions where β contains no p-actions.[5] Now, *NewNode* only updates local variables, and heap locations that no other process can see, and so commutes with all actions of other processes; i.e. $NewNode_p\ a_q = a_q\ NewNode_p$, for actions a and distinct processes p and q. Thus, we have $\alpha\ NewNode_p\ \beta\ AddNode_p\ \gamma = \alpha\ \beta\ NewNode_p\ AddNode_p\ \gamma$. By repeatedly applying this transformation, any execution of *QUEUE* can be transformed into one in which all completed executions of *enqueue* are performed without interruption, and is thus equivalent to an execution of *QUEUE*.

NewNode can now be refined to a sequence of assignments:

$node := newNode()$;
$node.value := x$;
$node.next := null$

Again, it is easy to show that this is a correct sequential refinement of *NewNode* (and that the last two assignments could be swapped). It is also atomic, since all three statements are both-movers, as their effects are not visible to other processes.

4.2 Weakening the invariant

Next, we consider how to implement *AddNode*, which involves updating both *Tail* and *next*. We can't update both of these in a single atomic action, and updating only one of them will break the state invariant.[6] Updating *Tail*

[5] In describing traces, we omit concatenation operators and identify an action with the singleton sequence containing it, so we write $\alpha\ NewNode_p\ \beta\ AddNode_p\ \gamma$ instead of $\alpha \frown \langle NewNode_p \rangle \frown \beta \frown \langle AddNode_p \rangle \frown \gamma$.

[6] Things would be simple if we had a double CAS instruction that could update $next(Tail)$ and *Tail* in one atomic operations. However, we are only assuming that the target hardware has a CAS that can update one location, so these steps have to be done sequentially.

first would leave *Tail* pointing to a node that was not connected to the list, and it seems unlikely that we would be able to make such an algorithm work (if another process started an *enqueue* in the ensuing state that node would probably get lost).

Michael and Scott's solution to this problem is to first update *next* so as to append the new node, and then advance *Tail* (or, as it turns out, attempt to). This means that other processes may observe a state in which *Tail* does not point to the last node in the list, so we have to weaken the state invariant to relax this condition. The important question is how much do we weaken the invariant? One option would be to discard *Tail* entirely, and traverse the list from *Head* every time we attempt to append a new node, but that would be unnecessarily expensive. It turns out that we can obtain an efficient lock-free implementation by allowing *Tail* to "lag" behind the actual end of the list by at most one node, as illustrated in Figure 5, which shows alternative representations of a queue containing a, b and c (cf. Figure 3(b)) and one containing just c.

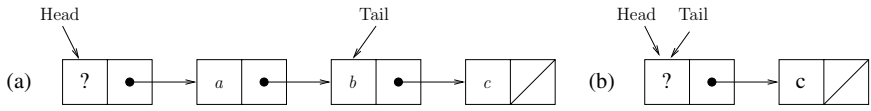

Fig. 5. Queue representation with *Tail* lagging

Thus, the revised invariant (which for simplicity, and — we hope — minimal risk of confusion, we will continue to call *Inv*) is obtained by weakening the last conjunct of *Rep*, by adding the underlined disjunct:

$$\begin{aligned} &Rep(f, Head, Tail, value, next) \;\; \widehat{=} \\ &\quad \mathrm{dom}\, value = \mathrm{dom}\, next \wedge next \circ (\langle Head \rangle \frown f) = f \frown \langle null \rangle \wedge \\ &\quad (\; Tail = last(\langle Head \rangle \frown f) \vee \underline{next(Tail) = last(\langle Head \rangle \frown f)}\;) \end{aligned}$$

The definition of *Abs* remains unchanged, since the abstract queue represented by the linked list does not depend on *Tail*.

The initial state constructed in Section 3.3 still correctly represents the empty queue. With this invariant, $next(Head) = null$ still implies that the queue is empty, but $Head = Tail$ doesn't since we may have a queue containing one value where *Tail* is lagging.

In any state, $next(Tail) = null$ holds if *Tail* is not lagging, and $next(Tail) \neq null \wedge next(next(Tail)) = null$ holds if *Tail* is lagging.

The derivation so far is still valid with the new invariant. The only difference is that queue operations may now be applied in a state where *Tail* is lagging, since with the anticipated sequential decomposition, an operation may be invoked in a state where an enqueueing process has appended a new node but not yet advanced *Tail*. This is reflected in the implicit precondition.

We could, at this stage, also relax the specification of *enqueue* by allowing it to establish the new invariant, rather than the old one, which means that it could leave *Tail* lagging — we will discuss this option briefly in Section 6, but will not pursue it here. We will consider the effect of the new invariant of the specification of *dequeue* in Section 5.

4.3 Splitting AddNode

Pursuing our decision to append the new node and then advance *Tail*, we now consider how to split *AddNode* into a suitable sequential composition. Our first, tentative, attempt is to consider the obvious sequential refinement of *AddNode*:

$$next : [\, next = next_0 \oplus \{Tail_0 \mapsto node\} \,] \;; \qquad (Append?)$$
$$Tail : [\, Tail = next(Tail_0) \,] \qquad (Advance?)$$

This will not work, however, since *Append*? only behaves correctly if *Tail* is not lagging. If *Tail* is lagging, it is because another process has begun an *enqueue*, and performed its *Append*? but not its *Advance*?. In a lock-based implementation, we would just wait for that process to complete its operation, but that is not acceptable in a lock-free algorithm. Instead, we modify *Append*? so that if it sees *Tail* lagging, it advances *Tail* — thus completing the other process's *enqueue* operation — before attempting to append its node. Now, however, after an enqueueing process appends its node, another process may advance *Tail* before this process gets to execute its *Advance*? action, so we must allow *Advance*? to advance *Tail* only if it is actually lagging and otherwise leave *Tail* unchanged.

Our second attempt, taking these interactions into account, is to refine *AddNode* to:

$$next, Tail : \begin{bmatrix} next_0(Tail) = null \wedge \\ next = next_0 \oplus \{Tail \mapsto node\} \end{bmatrix} \;; \qquad (Append)$$
$$Tail : \begin{bmatrix} next(Tail_0) \neq null \wedge Tail = next(Tail_0) \;\vee \\ next(Tail_0) = null \wedge Tail = Tail_0 \end{bmatrix} \qquad (Advance)$$

It is easy to see that this is also a correct sequential refinement of *AddNode*, since when this code is executed sequentially, *Append* will always start in a state where $next(Tail) = null$ and end in a state where $next(Tail) \neq null$, in which case this sequence is equivalent to *Append*? ; *Advance*?. Note that $next_0(Tail) = null$ is unambiguous, and is equivalent to $Tail = Tail_0 = null \vee Tail_0 \neq null \wedge Tail = next_0(Tail_0)$, since we can show that there is only ever one location, l, in the domain of *next* such that $next(l) = null$.

Showing that the implementation is atomic is not so straightforward. *Append* and *Advance* both update shared variables, so in general they do

not commute with other actions that access these variables. We would not expect *Append* to move, since it determines the new value of the abstract queue, and is thus the linearisation point for *enqueue*. And moving an *Advance* left over another *Advance* would affect whether *Tail* is updated, and thus alter the behaviour of the execution as seen by other processes. So let us consider more carefully how *Append* and *Advance* actions interact.

First, notice that an execution containing a completed execution of *AddNode* will contain an execution of *Append* which may or may not update *Tail*, and an execution of *Advance*, which also may or may not update *Tail*. Let us write $Append^+$ (resp., $Advance^+$) to denote an execution of *Append* (resp., *Advance*) which advances *Tail* and $Append^-$ (resp., $Advance^-$) to denote one that doesn't. The above discussion shows that in a sequential execution, every execution of *AddNode* consists of an $Append^-$ followed by an $Advance^+$.

Next, observe that $Append^+$ is equivalent to $Advance^+$; $Append^-$. Thus, we can assume that *Tail* is only updated by $Advance^+$ actions, and don't need to consider $Append^+$ actions any further.

Now, if process p performs an *Append* and later its *Advance* fails, this can only be because another process, say q, has advanced *Tail* with an $Advance^+$. But in this case, the effect is the same as if p had performed the $Advance^+$ and q then performed an $Advance^-$, since we can show that $Advance_q^+\ Advance_p^- = Advance_p^+\ Advance_q^-$. Thus, in an execution of the form $\alpha\ Append_p^-\ \beta\ Advance_p^-\ \gamma$, β must be of the form $\beta_1\ Advance_q^+\ \beta_2$, and the execution is equivalent to $\alpha\ Append_p^-\ \beta_1\ \ Advance_p^+\ Advance_q^-\ \beta_2\ \gamma$. That is, there is an execution in which p gets to perform its *Advance* successfully before q decides whether to advance *Tail*.

In this way, we can move $Advance^+$ actions left, if necessary reassigning them to different processes, until every $Advance^+$ is immediately preceded by an $Append^-$ action by the same process; $Advance^-$ actions can then be discarded. Thus, we can turn any execution containing a completed execution of this implementation if *AddNode* into one in which the corresponding occurrences of *Append* and *Advance* are executed without interruption, which is then equivalent to an execution of *AddNode*.

The ability to reassign an *Advance* action to a different process, and omit an *Advance* action which is no longer needed, is crucial to being able to verify Michael and Scott's algorithm — just permuting the order of actions as in Lipton's version of reduction, is not sufficient. The key observation here is that any execution consisting of only *enqueue* operations will contain an alternating sequence of *Append* and *Advance* actions. It doesn't matter what process performs the $Advance^+$ actions, so we can obtain a sequential execu-

tion by assuming that they are performed by the same process as the preceding *Append*, and discarding all unsuccessful *Advance* actions.

4.4 Anticipating interference and introducing a loop

We now consider how to implement *Append*. We will have to test whether $next(Tail)$ is *null*, in order to determine whether to advance *Tail*, but *Tail* or $next(Tail)$ may change between performing the test and taking the selected action. In this case we want that action to fail and try again. So before introducing the test, we refine *Append* to a loop whose body either "succeeds", completing the *Append* action and exiting the loop, or "fails", leaving *next* and *Tail* unchanged and continuing:

$$\begin{array}{l}\textbf{var } r : boolean \textbf{ ;}\\ \textbf{repeat}\\ \quad next,\\ \quad Tail, \textbf{:} \left[\begin{array}{l} next_0(Tail) = null \wedge \\ \quad next = next_0 \oplus \{Tail \mapsto node\} \wedge r \ \vee \\ next = next_0 \wedge Tail = Tail_0 \wedge \neg\, r \end{array}\right] \qquad (tryAppend)\\ \quad r\\ \textbf{until } r\end{array}$$

Now, any execution of this implementation of *Append* consists of zero or more failed executions of *tryAppend*, followed by one successful execution of *tryAppend*.[7] The failed executions of *tryAppend* can be discarded — the effect is just the same as if the process waited a bit longer rather than attempting those executions. This leaves a single successful execution of *tryAppend* node, which is then equivalent to *Append*, since the assignment to r is not observable by other processes.

At this stage, *tryAppend* chooses nondeterministically whether to succeed or fail, so it is possible that the loop will not terminate. This is sufficient for proving linearisability, since it is a safety property. In subsequent refinements, we will tighten the conditions governing these options as it becomes clear under what circumstances a *tryAppend* action is able to succeed.

4.5 Splitting tryAppend

We now consider how to implement *tryAppend*. As indicated above, we need to test whether $next(Tail)$ is *null*, and if not (attempt to) update *Tail* before we (attempt to) append the new node. But if we do advance *Tail*, another process may perform an *Append* before this process gets to append its node, so we need to do the test again. This can happen any number of times, so we

[7] We will ignore the tests on r, since they aren't really part of the logic of the algorithm. The same effect could be obtained by using an **exit** statement to terminate the loop, but that form is not as convenient to derive from a specification.

need to use a loop. While it might be tempting to consider a nested loop, we can use the loop we have already introduced. To facilitate this, we rearrange *tryAppend* to separate updating *Tail* and appending the new node:

$$\begin{array}{l} next, \\ Tail, \\ r \end{array} : \left[\begin{array}{l} next_0(Tail_0) = null \wedge Tail = Tail_0 \\ \quad next = next_0 \oplus \{Tail_0 \mapsto node\} \wedge r \ \vee \\ next_0(Tail_0) \neq null \wedge Tail = next_0(Tail_0) \wedge \\ \quad next = next_0 \wedge \neg\, r \ \vee \\ next = next_0 \wedge Tail = Tail_0 \wedge \neg\, r \end{array} \right] \qquad (tryApp)$$

It is not straightforward to show that this is a correct sequential refinement, since the effect of a single *tryAppend* action may now be spread over two executions of the loop body, one updating *Tail* and one appending the new node. Writing *tryApp*.1, *tryApp*.2 and *tryApp*.3 for the three branches of *tryApp*, we observe that in every state where *tryApp* may be executed, either *tryApp*.1 or *tryApp*.2 is enabled, but not both. Furthermore, *tryApp*.1 causes the loop to exit, while *tryApp*.2 enables *tryApp*.1 and disables itself. Thus, in a sequential execution, any execution of the loop with *tryApp* as its body will execute *tryApp*.2 at most once and then *tryApp*.1 exactly once, and is thus equivalent to the first branch of *tryAppend*.

In a concurrent execution, things are more complicated, since actions of other processes may cause the loop to perform any number of *tryApp*.2 actions. If the loop executes no *tryApp*.2 actions, its *tryApp*.1 action maps to a *tryAppend* action. If the loop executes at least one *tryApp*.2 action, the last such action can be matched with the *tryApp*.1, in much the same way that $Advance^+$ were matched with $Append^-$ actions in Section 4.3, so the two together map to a *tryAppend* action, and any previous *tryApp*.2 actions map to $Advance^+$ actions.

4.6 Taking a snapshot

We now consider how to implement *tryApp*. Before testing whether $next(Tail)$ is *null*, we introduce local variables, *tail* and *nextl*, to hold the values of *Tail* and $next(Tail)$.[8] Using local variables reduces the number of more expensive accesses on shared memory, but more importantly allows us to fix on their values in a particular state and reason about it, even though the contents of those locations may have changed subsequently. We will eventually use these local variables in place of the corresponding shared variables, provided that

[8] Michael and Scott use *next* for the local variable as well as the field name. It is harder to distinguish the two uses in specification statements, where *next* can appear in the frame and we need to write conditions like $next = next_0$, than it is in code, where a field name is always preceded by a dot.

those variables have not been changed since they were read — if they have changed, *tryApp* should fail. We thus refine *tryApp* to:

$$
\begin{array}{l}
\textbf{var } tail, nextl \textbf{ ;} \\
tail := Tail \textbf{ ;} \\
nextl := tail.next \textbf{ ;} \\
next, \\
Tail, \\
r
\end{array}
\Bigg[\!\!\Bigg]
\begin{array}{l}
tail = Tail_0 \wedge nextl = next_0(tail) \wedge \\
\quad next_0(Tail_0) = null \wedge Tail = Tail_0 \wedge \\
\quad next = next_0 \oplus \{Tail_0 \mapsto node\} \wedge r \ \vee \\
tail = Tail_0 \wedge nextl = next_0(tail) \wedge \\
\quad next_0(Tail_0) \neq null \wedge Tail = next_0(Tail_0) \wedge \\
\quad next = next_0 \wedge \neg\, r \ \vee \\
next = next_0 \wedge Tail = Tail_0 \wedge \neg\, r
\end{array}
\qquad (tryApp')
$$

This is a straightforward sequential refinement. To demonstrate atomicity, we first observe that if $tail = Tail_0$, then *Tail* has not changed since it was read in the assignment $tail := Tail$. This follows from the fact that *newNode*() always allocates a new node, so this implementation never reuses heap locations. Without this assumption, the condition $tail = Tail$ would not guarantee that the rest of the queue had not changed in a way that invalidated the *tryApp*′ action — we will discuss this assumption further in Section 6. Next, we observe that $nextl = next_0(tail)$ implies that $next(Tail)$ has not changed since it was read in the assignment to *nextl*. This follows from the fact that, under the assumption that heap locations are not reused, for any location l, $next(l)$ is only assigned twice: once (to *null*) when it is allocated, and once (to a non-*null* value) when it is the last node in the list when a new node is appended. So $next(Tail)$ cannot change from one value to another and back to the previous value again.

We can thus show that an execution containing a completed execution of this implementation of *tryApp* is equivalent to one in which the three statements are executed without interruption at the position of the occurrence of *tryApp*′. If *tryApp*′ takes the first or second branch, the assignments to *tail* and *nextl* can be moved right to that they are adjacent to *tryApp*′, since any intervening actions do not alter *Tail* or $next(Tail)$. If *tryApp*′ takes the third branch, the assignments to *tail* and *nextl* can again be moved right to that they are adjacent to *tryApp*′, since *tryApp*′ can take this branch irrespective of whether *Tail* and $next(Tail)$ have changed.

4.7 Checking whether Tail is lagging

We finally refine *tryApp*′ by introducing the anticipated test to determine whether $next(Tail)$ is *null*, which is now expressed using the local variable *nextl*:

if $nextl = null$ **then**

$$\begin{matrix} next, \\ r \end{matrix} : \left[\begin{array}{l} nextl = null \wedge \\ (\ tail = Tail \wedge nextl = next_0(tail) \wedge \\ \quad next_0(Tail) = null \wedge \\ \quad next = next_0 \oplus \{Tail \mapsto node\} \wedge r \ \vee \\ \ next = next_0 \wedge \neg\ r\ ) \end{array} \right] \qquad (tryAppT)$$

else

$$\begin{matrix} Tail, \\ r \end{matrix} : \left[\begin{array}{l} nextl \neq null \wedge \\ (\ tail = Tail_0 \wedge nextl = next(tail) \wedge \\ \quad next(Tail_0) \neq null \wedge \\ \quad Tail = next(Tail_0) \wedge \neg\ r \ \vee \\ \ Tail = Tail_0 \wedge \neg\ r\ ) \end{array} \right] \qquad (tryAppF)$$

fi

It is easy to see that this is a correct sequential refinement — in a sequential execution, we can assume that $nextl = null$ holds when $tryAppT$ is executed and that $nextl \neq null$ holds when $tryAppF$ is executed. We have therefore deleted the second conjunct of $tryApp'$ from $tryAppT$ and the first conjunct of $tryApp'$ from $tryAppF$, since those branches are now impossible by virtue of the outcome of the **if** test, and made consequent simplifications by contracting the frames of the two specifications. To demonstrate atomicity, we note that an execution of this implementation of $tryApp'$ consists of a test, followed by either $tryAppT$ or $tryAppF$, according to the outcome of the test. Since the test $nextl = null$ only refers to a local variable, it can be moved right over any intervening steps to be adjacent to the resulting occurrence of $tryAppT$ or $tryAppF$.

This reasoning shows that the standard sequential rule for refining a specification statement to an **if** statement [20] is valid in a concurrent setting when the test refers only to local variables.

4.8 Updating next with a CAS

We are now almost ready to implement $tryAppT$ using a CAS. Michael and Scott use a CAS which compares $tail.next$ with $nextl$, so to get it into the required form, we first refine $tryAppT$ to:

$$\begin{matrix} next, \\ r \end{matrix} : \left[\begin{array}{l} next_0(tail) = nextl \wedge \\ \quad next = next_0 \oplus \{tail \mapsto node\} \wedge r \ \vee \\ next_0(tail) \neq nextl \wedge next = next_0 \wedge \neg\ r \end{array} \right] \qquad (tryAppT')$$

To show that this is a correct sequential refinement of $tryAppT$, we first note $nextl = null$ can be assumed to hold from the **if** test. Next, we observe that $Tail = tail$ must also still hold, since if $next(Tail) = null$, $Tail$ cannot be modified before $next(Tail)$ is changed. We can now infer the first conjunct of $tryAppT$ from the first conjunct of $tryAppT'$, while the second conjunct

of $tryAppT$ from the second conjunct of $tryAppT'$. Also notice that have strengthened the second branch of $tryAppT'$ so that $tryAppT$ is deterministic — this ensures that $tryAppT$ only fails when it has to, which (along with similar strengthenings elsewhere) allows us to show that the implementation of *enqueue* is lock-free.

Since the specification of $tryAppT'$ now matches the specification for a CAS (see Section 2.2), we can re-express $tryAppT'$ as:

$$r := CAS(tail.next, nextl, node)$$

Note that the above reasoning would still apply if we used *null* in place of *nextl*, since the test has just shown that *nextl* is equal to *null*, so we could instead write $r := CAS(tail.next, null, node)$.

4.9 Updating Tail with a CAS

Similarly, we are almost ready to implement $tryAppF$ using a CAS, so we refine $tryAddF$ to get it into the required form:

$$\begin{array}{l} Tail, \\ r \end{array} :\left[\begin{array}{l} Tail_0 = tail \wedge Tail = nextl \wedge r \ \vee \\ Tail_0 \neq tail \wedge Tail = Tail_0 \wedge \neg\, r \end{array}\right]; \qquad (tryAppF')$$
$$r := false$$

We have assigned r to *false* separately, so that $tryAppF'$ can assign r in the way required by the CAS, and again we have strengthened the second branch to make $tryAppF'$ deterministic, so $tryAppF'$ will only fail when it actually experiences interference.

We have already shown that if $Tail = tail$ holds, then *Tail* can't have changed since it was read, and that $next(Tail)$ must also be unchanged since it was read into *nextl*, We can now infer the first conjunct of $tryAppF$ from the first conjunct of $tryAppF'$, while the second conjunct of $tryAppT$ follows from the second conjunct of $tryAppF'$.

Since the specification of $tryAppF'$ now matches the specification for a CAS (see Section 2.2), we can re-express $tryAppF'$ as:

$$r := CAS(Tail, tail, next)$$

4.10 Updating Tail with another CAS

Finally, we can also implement *Advance* using a CAS. We can allow *Advance* to assign to r, since it will not be used again within its scope. It can then be re-expressed as:

$$r := CAS(Tail, tail, next)$$

Alternatively, we could declare a new local variable to receive the result of the CAS, or assume (as Michael and Scott do) that we can call CAS as a statement when we don't need its result.

This completes the derivation. The final implementation of *enqueue* is shown in Section 6.

5 Refining dequeue

We now consider how to implement the *dequeue* operation as specified in Figure 4. Following the pattern observed in implementing *enqueue*, we can anticipate that *dequeue* will require a loop, in which we read *Head* into a local variable, attempt to advance *Head* using a CAS, and retry if the CAS fails — the main difference is that with *dequeue*, we need to test whether the queue is empty, and if it is return $\perp$ to indicate this rather than attempting to advance *Head*. With the original state invariant, this would have been quite simple (in fact, it would be very similar to the *pop* operation in Treiber's lock-free stack implementation [24,18]). With the modified invariant, however, things are not so simple. We first need to consider the effect of weakening the state invariant as discussed in Section 4.2.

5.1 Revising the specification of dequeue

Weakening the state invariant has the effect that a *dequeue* operation may now be invoked in a state where *Tail* is lagging, and may finish with *Tail* lagging. We also have to change the concrete specification to reflect that fact that $Head = Tail$ no longer implies that the queue is empty. We observe that $Head \neq Tail$ still implies that the queue is non-empty, and in this case we can complete the *dequeue* by just advancing *Head* and reading the value to be returned — it doesn't matter if *Tail* is lagging and there is no point in insisting that *dequeue* advance *Tail* if is lagging. When $Head = Tail$ holds, $next(head) = null$ implies that the queue is empty, so *dequeue* can just return $\perp$. When $Head = Tail$ and $next(Head) \neq null$ both hold, *Tail* is lagging and there is one element in the queue (see Figure 5(b)). In this case we will follow Michael and Scott and advance *Tail* so that the resulting empty queue has $Head = Tail$.

We thus revise the specification to make these three cases explicit. Since *dequeue* may now modify *Tail*, we also add *Tail* to the frame and change occurrences of *Tail* to $Tail_0$:

$$\begin{array}{l} Head, \\ Tail, \\ y \end{array} :\left[\begin{array}{l} Head_0 = Tail_0 \wedge next(Head_0) = null \wedge \\ \quad Head = Head_0 \wedge Tail = Tail_0 \wedge y = \bot \ \vee \\ Head_0 = Tail_0 \wedge next(Head_0) \neq null \wedge \\ \quad Head = Tail = next(Head_0) \wedge \\ \quad y = value(next(Head_0)) \ \vee \\ Head_0 \neq Tail_0 \wedge Head = next(Head_0) \wedge \\ \quad Tail_0 = Tail_0 \wedge y = value(next(Head_0)) \end{array}\right] \qquad (Dequeue')$$

Observe that in a sequential execution, *Dequeue′* will always be invoked in a state where *Tail* is not lagging (provided that the refinement of *enqueue* enforces this), in which case $Head = Tail$ implies $next(Head_0) = null$ so *Dequeue′* is equivalent to *Dequeue* since the second branch is not possible.

It may seem heavy handed to make this special case explicit at this stage, but that appears to be necessary in order to obtain Michael and Scott's implementation. In fact, a simpler approach is possible: we can detect an empty queue just by testing $next(head) = null$, and we don't need to treat the case of a singleton queue with *Tail* lagging as a special case, since the new invariant allows *Tail* to lag behind *Head* in an empty queue.

5.2 Anticipating interference and introducing a loop

As in the refinement of *Append* in Section 4.4, we anticipate that the operation may experience interference and need to retry. So we introduce a loop whose body either "succeeds", completing the *Append* action and exiting the loop, or "fails", leaving *Head* unchanged and continuing. We also anticipate that checking whether the *Tail* needs to be advanced may need to be done repeatedly, and so modify the second branch to just advance *Tail* and continue in the loop, as we did in Section 4.5.

var r : *boolean* **;**
repeat

$$\begin{array}{l} Head, \\ Tail, \\ y, r \end{array} :\left[\begin{array}{l} Head_0 = Tail_0 \wedge next(Head_0) = null \wedge \\ \quad Head = Head_0 \wedge Tail = Tail_0 \wedge y = \bot \wedge r \ \vee \\ Head_0 = Tail_0 \wedge next(Head_0) \neq null \wedge \\ \quad Head = Head_0 \wedge Tail = next(Head_0) \wedge \neg\, r \ \vee \\ Head_0 \neq Tail_0 \wedge Head = next(Head_0) \wedge \\ \quad Tail = Tail_0 \wedge y = value(next(Head_0)) \wedge r \ \vee \\ Head = Head_0 \wedge Tail = Tail_0 \wedge \neg\, r \end{array}\right] \qquad (tryDeq)$$

until r

To show that this is a correct refinement we observe that any execution of the code will consist of zero or more failed occurrences of the loop body, followed by one successful execution. The failed executions leave the queue unchanged, though they may advance *Tail*. Executions that do not alter the state can be discarded, and executions that advance *Tail* can be reassigned to

the process that performed the preceding *Append*, as we did in Section 4.3. We are then left with a single successful execution, which is equivalent to $dequeue'$.

5.3 Taking a shapshot

Following Michael and Scott, we are going to compare *Head* and *Tail*, to determine whether the queue could be empty. However, we cannot do that and update *Head* atomically, so we introduce local variables *head* and *tail* and read their values into them. At the same time, we notice that we will also need the value of $next(Head)$, so we will store that as well. We thus declare and initialise these local variables, and replace by the corresponding local variable those occurrences of the global variables that are used to test or update values (we leave those that are used to indicate that values are unchanged). We also add conjuncts ensuring that the first three branches will only be enabled if the relevant shared variables are unchanged, forcing $tryDeq'$ to fail via the fourth branch if interference is detected (note that we only constrain *Tail* in this way in the case where they it is going to be updated):

$$
\begin{array}{l}
\textbf{var}\ head, tail, nextl : NLoc\ ; \\
head := Head\ ; \\
tail := Tail\ ; \\
nextl := Head.next\ ; \\
\quad Head, \\
\quad Tail, \\
\quad y, r
\end{array}
\ :\ \left[\begin{array}{l}
head = Head_0 \wedge head = tail \wedge nextl = null \wedge \\
\quad Head = Head_0 \wedge Tail = Tail_0 \wedge y = \bot \wedge r \ \vee \\
head = Head_0 \wedge head = tail \wedge tail = Tail_0 \wedge \\
\quad nextl = next(Head_0) \wedge nextl \neq null \wedge \\
\quad Head = Head_0 \wedge Tail = nextl \wedge \neg\, r \ \vee \\
head = Head_0 \wedge nextl = next(Head_0) \wedge \\
\quad head \neq tail \wedge Head = nextl \wedge \\
\quad Tail = Tail_0 \wedge y = value(nextl) \wedge r \ \vee \\
Head = Head_0 \wedge Tail = Tail_0 \wedge \neg\, r
\end{array}\right] \qquad (tryDeq')
$$

It is easy to see that this is a correct sequential refinement. To demonstrate atomicity, we consider the four branches separately.

In the first branch, we can infer that *Head* is not changed between the first assignment and $tryDeq'$, but we can infer nothing about *Tail*. So we move the first assignment right, and the third assignment and $tryDeq'$ left to the position of the second assignment, so that all four statements are executed without interruption. Note that this shows that the assignment to *tail* is the linearisation point for a *dequeue* returning $\bot$, since it is only at the point where *Tail* was read that we can be sure the queue was empty.

In the other three branches, we will show that we can the three assignments right to the position of $tryDeq'$, so again all four statements are executed without interruption:

- In the second branch, we know that *Head* and *Tail* are unchanged between when they were read and *tryDeq*′, so we can move the three assignments right.
- In the third branch, we know that *Head* has not changed, so its assignment can move right. We also know that *Head* and *Tail* were different at the point where *tail* was read, and we can show that if *Head* and *Tail* are different in some state then they cannot become equal without *Head* changing; thus, we can also move the assignment to *Tail*.
- The fourth branch fails irrespective of the values of *head*, *tail* and *nextl*, so again the assignments can be moved right.

5.4 Checking consistency of head and nextl

Next, we introduce an **if** statement to test whether *Head* still has the value observed when it was read into *head*, and cause *tryDeq*′ to fail if *Head* has changed. If *Head* has not changed, it means that *head* and *nextl* are consistent, i.e. that *nextl* was the value of $next(Head)$ at the point where *Head* was read into *head*. We also delete the constraint $head = Head_0$ from the first branch of *tryDeqT*, since this branch doesn't depend on the current value of *Head*. Thus, *tryDeq*′ becomes:

$$
\begin{array}{l}
\textbf{if}\ \ head = Head\ \textbf{then} \\
\quad Head,\ Tail,\ y, r \ :\ \left[\begin{array}{l}
head = tail \wedge nextl = null \wedge \\
\quad Head = Head_0 \wedge Tail = Tail_0 \wedge y = \bot \wedge r \ \vee \\
head = Head_0 \wedge head = tail \wedge tail = Tail_0 \wedge \\
\quad nextl = next(Head_0) \wedge nextl \neq null \wedge \\
\quad Head = Head_0 \wedge Tail = nextl \wedge \neg\, r \ \vee \\
head = Head_0 \wedge nextl = next(Head_0) \wedge \\
\quad head \neq tail \wedge Head = nextl \wedge \\
\quad Tail = Tail_0 \wedge y = value(nextl) \wedge r \ \vee \\
Head = Head_0 \wedge Tail = Tail_0 \wedge \neg\, r
\end{array}\right] \qquad (tryDeqT) \\
\textbf{else} \\
\quad r := false \qquad (tryDeqF) \\
\textbf{fi}
\end{array}
$$

This is clearly a correct sequential refinement: when $head = Head$ holds, it performs the same specification statement as *tryDeq*′, except for the deletion of $head = Head_0$ and that can be restored in the sequential case; and when $head = Head$ does not hold, it is establishes the fourth branch of *tryDeq*′.

To demonstrate atomicity, we must show that either the test or the selected specification statement can be moved to the position of the other, so the test and the specification are executed without interruption. The first branch *tryDeqT* only depends on local variables, so it can move left to the position of

the test; for the other three branches, the test can move right to the position of *tryDeqT*. Finally, *tryDeqF* can move left to the position of the test.

5.5 Checking whether the queue could have been empty

We now refine *tryDeqT* to determine whether the queue could have been empty by comparing *head* and *tail*:

$$
\begin{array}{l}
\textbf{if } head = tail \textbf{ then} \\
\quad Head, Tail, y, r : \left[\begin{array}{l}
head = tail \wedge nextl = null \wedge \\
\quad Head = Head_0 \wedge Tail = Tail_0 \wedge y = \bot \wedge r \ \vee \\
head = Head_0 \wedge head = tail \wedge tail = Tail_0 \wedge \\
\quad nextl = next(Head_0) \wedge nextl \neq null \wedge \\
\quad Head = Head_0 \wedge Tail = nextl \wedge \neg\, r \ \vee \\
head = Head_0 \wedge nextl = next(Head_0) \wedge \\
\quad head \neq tail \wedge Head = nextl \wedge \\
\quad Tail = Tail_0 \wedge y = value(nextl) \wedge r \ \vee \\
Head = Head_0 \wedge Tail = Tail_0 \wedge \neg\, r
\end{array}\right] \qquad (tryDeqTT) \\
\textbf{else} \\
\quad Head, Tail, y, r : \left[\begin{array}{l}
head = tail \wedge nextl = null \wedge \\
\quad Head = Head_0 \wedge Tail = Tail_0 \wedge y = \bot \wedge r \ \vee \\
head = Head_0 \wedge head = tail \wedge tail = Tail_0 \wedge \\
\quad nextl = next(Head_0) \wedge nextl \neq null \wedge \\
\quad Head = Head_0 \wedge Tail = nextl \wedge \neg\, r \ \vee \\
head = Head_0 \wedge nextl = next(Head_0) \wedge \\
\quad head \neq tail \wedge Head = nextl \wedge \\
\quad Tail = Tail_0 \wedge y = value(nextl) \wedge r \ \vee \\
Head = Head_0 \wedge Tail = Tail_0 \wedge \neg\, r
\end{array}\right] \qquad (tryDeqTF) \\
\textbf{fi}
\end{array}
$$

This is clearly a correct sequential refinement, since it performs the same specification irrespective of the outcome of the test. It is also easy to see that it is atomic since the test refers only to local variables and can thus move right to the position of the subsequent specification statement.

5.6 Checking whether the queue was actually empty

We now refine *tryDeqTT* to test whether the queue was actually empty, by testing whether *nextl* is *null*. We also simplify *tryDeqTT* by noting that the third branch is impossible as we know that $head = tail$ still holds when *tryDeqTT* is executed:

if $nextl = null$ **then**

$$Head, Tail, y, r :\left[\begin{array}{l} head = tail \wedge nextl = null \wedge \\ \quad Head = Head_0 \wedge Tail = Tail_0 \wedge y = \bot \wedge r \ \vee \\ head = Head_0 \wedge head = tail \wedge tail = Tail_0 \wedge \\ \quad nextl = next(Head_0) \wedge nextl \neq null \wedge \\ \quad Head = Head_0 \wedge Tail = nextl \wedge \neg\, r \ \vee \\ Head = Head_0 \wedge Tail = Tail_0 \wedge \neg\, r \end{array}\right] \qquad (tryDeqTTT)$$

else

$$Head, Tail, y, r :\left[\begin{array}{l} head = tail \wedge nextl = null \wedge \\ \quad Head = Head_0 \wedge Tail = Tail_0 \wedge y = \bot \wedge r \ \vee \\ head = Head_0 \wedge head = tail \wedge tail = Tail_0 \wedge \\ \quad nextl = next(Head_0) \wedge nextl \neq null \wedge \\ \quad Head = Head_0 \wedge Tail = nextl \wedge \neg\, r \ \vee \\ Head = Head_0 \wedge Tail = Tail_0 \wedge \neg\, r \end{array}\right] \qquad (tryDeqTTF)$$

fi

Again, this is clearly a correct sequential refinement, since it performs the same specification irrespective of the outcome of the test, and is obviously atomic since the test refers only to local variables and can thus move right to the position of the subsequent specification statement.

5.7 Reporting an empty queue

In *tryDeqTTT* we know the first branch will succeed and we can ignore the other two (the second is impossible, and we don't need the third option). We can thus refine *tryDeqTTT* to two assignments to y and r, signalling that an empty queue has been detected:

$$y := \bot \,;\, r := true$$

5.8 Advancing Tail

In *tryDeqTTF* we know that *nextl* was not *null* when it was read, which means that, at that point, another process had appended a new node to the linked list but not yet advanced *Tail*. In order to ensure that the system as a whole makes progress, the *dequeue* operation should now attempt to complete that *enqueue* by advancing *Tail* — of course, this may not succeed, since some other process may have advanced *Tail* since it was read by this process.

As in Section 4.9, we wish to use a CAS to update *Tail* provided that it is still equal to *tail*. We know the first branch of *tryDeqTTF* is impossible, so we can delete it; we can ensure $Head = Head_0$ by removing *Head* from the frame; and y can also be removed from the frame. We also know that $nextl \neq null$ and $nextl = next(head)$ hold from the **if** tests, so we are left with:

$$Tail, r :\left[\begin{array}{l} tail = Tail_0 \wedge Tail = nextl \wedge \neg\, r \ \vee \\ Tail = Tail_0 \wedge \neg\, r \end{array}\right] \qquad (tryDeqTTF)$$

This is just what we want for the CAS except that it sets r to false in both branches, and may always take the second branch. We can address these differences as we did in Section 4.9 by assigning r to *false* separately and strengthening the second branch by adding $Tail_0 = tail$ as a conjunct. We can then refine *tryDeqTTF* to:

$r := CAS(Tail, tail, next)$;
$r := false$

5.9 Refining tryDeqTF

Since we know that $head \neq tail$ still holds when it is executed, *tryDeqTF* simplifies to:

$$Head, y, r : \left[\begin{array}{l} head = Head_0 \wedge Head = nextl \wedge \\ \quad y = value(nextl) \wedge r \ \vee \\ Head = Head_0 \wedge \neg\, r \end{array} \right] \qquad (tryDeqTF')$$

This looks rather like a CAS, along with the conjunct specifying y. Since it doesn't matter whether we assign y when the CAS fails, we can refine this to:

$y = nextl.value$;
$r := CAS(Head, head, nextl)$

It is easy to see that this is a correct sequential refinement. To show that it is atomic we note that if the CAS succeeds, we can infer that the assignment to y can be moved to the position of the CAS, since *value* is never altered (we also assume that the value of an **out** parameter is not visible until the procedure exits). Indeed, we can place the assignment after the CAS, so we only read *value* if the CAS succeeds, which might be preferable if queue values are large.

This completes the derivation. The final implementation of *enqueue* is shown in Section 6.

6 Discussion

Collecting up the code fragments introduced throughout Sections 4 and 5, and moving local variable declarations to the top, the final implementations of *enqueue* and *dequeue* is as shown in Figure 6.

There are several minor notational differences between these and Michael and Scott's version: they use C-like notation for parameter passing, assignments and tests; they use unconditional loops, and use **return** and **break** statements for loop exit to loop; and they use *next* for the local variable as

```
enqueue(in x : T) ≙
  var node, tail, nextl : NLoc ;
  var r : boolean ;
  node := newNode() ;
  node.value := x ;
  node.next := null ;
  repeat
    tail := Tail ;
    nextl := tail.next ;
    if nextl = null then
      r := CAS(tail.next, nextl, node)
    else
      r := CAS(Tail, tail, nextl) ;
      r := false
    fi
  until r ;
  r := CAS(Tail, tail, nextl)
```

```
dequeue(out y : T⊥) ≙
  var head, tail, nextl : NLoc ;
  var r : boolean ;
  repeat
    head := Head ;
    tail := Tail ;
    nextl := head.next ;
    if head = Head then
      if head = tail then
        if nextl = null then
          y := ⊥ ; r := true
        else
          r := CAS(Tail, tail, nextl) ;
          r := false
        fi
      else
        y = next.value ;
        r := CAS(Head, head, nextl)
      fi
    else
      r := false
    fi
  until r
```

Fig. 6. Final code for *enqueue* and *dequeue*

well as a field name. Also, they return a boolean value to indicate whether *dequeue* detected an empty queue or is returning a proper value, where we return $\perp$ in the latter case.

A more significant difference is that in *enqueue*, they test whether $tail = Tail$ holds before testing $next = null$, to check that *tail* and *nextl* are consistent, and repeat the loop body if this test fails. We have shown that this test is unnecessary. The test $head = Head$ in *dequeue* is unnecessary for similar reasons, but we chose to retain it to see how it affected the derivation. Our discussion has also shown other places where the implemenation could be improved — in particular, we can simplify the test for empty queue in *dequeue* so that the *dequeue* operation never needs to access *Tail*. If we allow *enqueue* to establish the new invariant, and thus leave *Tail* lagging, we see that the final *CAS* in *enqueue* is unnecessary. Changing the order of refinement steps would also lead to some minor variations in the algorithm — for example, introducing the loop in *enqueue* before splitting *AddNode* would place the final CAS inside the loop; introducing the loop before allocating the new node would result in the first three assignments being included in the loop, and a new node being allocated every time the operation was retried.

Finally, we have assumed that heap locations are not reused, which allows us to justify the ABA Freedom assumption. The version of this algorithm

included in the Java concurrency library is identical to ours (apart from the differences already noted), but assumes that the implementation language provides automatic garbage collection. This can be justified as a data refinement step, in which the storage manager is able to reclaim heap locations that are not reachable from any of the pointer variables used in the queue implementation, and locations are made to still appear unique by pairing a hardware pointer address with a unique counter.

Michael and Scott maintain their own free list, to which nodes are returned when they are removed from the queue representation in *dequeue* and from which nodes are allocated (if any are available) in *enqueue* — this avoids the need to count references as required for full garbage collection, but means that the storage used never falls below a factor of the maximum queue size. They also add version numbers to pointer variables, which are incremented every time a pointer is modified, so that a reallocated pointer will always appear different from its previous incarnation. This mechanism can be also be introduced as a further data refinement, but is only strictly correct if version numbers are unbounded. An alternative approach which avoids this problem is described in [10].

7 Conclusions

We have shown that a version of Michael and Scott's lock-free queue can be derived from an abstract specification in a series of verifiable refinement steps in a way that shows the resulting implementation is linearisable. Using this approach allows us to explain why the algorithm is correct in a way that we believe provides more insight than other proofs into why the algorithm is constructed the way it is, and also identifies other choices that might have been taken. Indeed, in some places following Michael and Scott's approach made the derivation more difficult than if simplifications had been made.

Our refinement steps are justified (albeit, rather informally) in a way that separates sequential reasoning from reasoning about possible interference. This kind of separation is quite common, dating back to Owicki and Gries [23], but while they were concerned with avoiding interference, we are concerned with showing that the algorithm works correctly in the presence of interference. We do this using a form of trace reduction, drawing on the work of Lipton, Lamport, Cohen, and others. [16,15,14,4]. We have augmented this basic approach with the ability to discard actions that have no effect — this is needed in other lock-free algorithms [7], where failed attempts are discarded in constructing an equivalent sequential, and is also done in using static analysis to show atomicity [27]. In other work [8] we needed to reduce the steps of two

operations simultaneously because the linearisation point of one operation was a step of a different operation.

In this derivation we also needed to modify actions, because of the "helper" mechanism whereby one process can complete an operation begun by another process. Similar modifications would appear to be needed in other algorithms that use similar mechanisms, such as Shann et al's array-based queue [25,5] and Ladan-Mozes and Shavit's "optimistic" lock-free queues [13]. An alternative approach (taken, for example, in [27]) is to say that since it doesn't matter what process performs an *Advance*, we can introduce a new process (or a set of processes) which runs in parallel with the rest, and just repeatedly performs *Advance* actions, which *enqueue* and *dequeue* no longer need to bother with.

Michael and Scott [18] gave a brief proof of some safety properties, but they were not sufficient to ensure linearisability. Yahav and Sagiv [28] describe an approach to verifying Michael and Scott's safety properties using, but their analysis appears to be very limited as they don't appear to have run the system with both *enqueue*s and *dequeue* being performed.

Doherty et al [6] describes a fully mechanical proof of a variant of Michael and Scott's which is intended to reduce contention in the *dequeue* operation by testing $next = null$, to determine whether the queue is empty, rather than $head = tail$, and only reading *Tail* if this test succeeds. This optimisation was discovered while attempting to prove the original algorithm. Recent work by Amit et al [2] on proving linearisability using model checking with abstraction shows that the algorithm described in [6] generates far fewer states that the original algorithm.

Abrial and Cansell [1] describe a constructive verification of a variant of Michael and Scott's algorithm using Event-B. They prove a variant of linearisability which requires the linearisation point to be the last step taken by an operation, and omit the final CAS from *enqueue* so that *Tail* is always advanced by the next operation that notices *Tail* lagging. They also introduce an additional test in *dequeue*, which requires *Tail* to be read again, before returning *false*. This is precisely the case that required a backward simulation in the verification in [6], and this modification appears to have been required to avoid the need for backward simulation.

Our future work will include mechanising our derivations using a theorem prover such as PVS, which will require more formal statements of the properties that the reduction steps rely on. We also intend to apply the approach to more sophisticated algorithms, such as the scalable queue described in [19], to see what other extensions are required to the basic reduction method.

Acknowledgement

We are grateful to Sun Microsystems Laboratories for financial support, and to Rob Colvin and Mark Moir for helpful discussions relating to this work.

References

[1] Abrial, J.-R. and D. Cansell, *Formal construction of a non-blocking concurrent queue algorithm*, Journal of Universal Computer Science **11** (2005), pp. 744–770.

[2] Amit, D., N. Rinetzky, T. Reps, M. Sagiv and E. Yahav, *Comparison under abstraction for verifying linearizability*, in: *CAV'07: Computer Aided Verification 2007*, 2007, to appear.

[3] Back, R.-J. and J. von Wright, *Trace refinement of action systems*, in: *International Conference on Concurrency Theory*, 1994, pp. 367–384.

[4] Cohen, E. and L. Lamport, *Reduction in TLA*, in: *International Conference on Concurrency Theory (CONCUR)*, 1998, pp. 317–331.

[5] Colvin, R. and L. Groves, *Formal verification of an array-based nonblocking queue*, in: *ICECCS '05: Proceedings of the Internation Conference on Engineering of Complex Computer Systems* (2005), pp. 92–101.

[6] Doherty, S., L. Groves, V. Luchangco and M. Moir, *Formal verification of a practical lock-free queue algorithm.*, in: D. de Frutos-Escrig and M. Núñez, editors, *FORTE2004 : Formal Techniques for Networked and Distributed Systems*, Lecture Notes in Computer Science **3235** (2004), pp. 97–114.

[7] Groves, L., *Reasoning about nonblocking concurrency using reduction*, in: *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)* (2007), pp. 11–14.

[8] Groves, L. and R. Colvin, *Derivation of a scalable lock-free stack algorithm*, in: *Proceedings of the 11th Refinement Workshop (REFINE 2006)*, Electronic Notes in Theoretical Computer Science **187**, 2007, pp. 55–74, (Revised version to appear in Formal Aspects of Computing).

[9] Herlihy, M., *Wait-free synchronization*, ACM Trans. Program. Lang. Syst. **13** (1991), pp. 124–149.

[10] Herlihy, M., V. Luchangco and M. Moir, *The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures*, in: *16th International Conference on Distributed Computing (DISC 2002)*, Lecture Notes in Computer Science **2508**, Toulouse, France, 2002, pp. 339–353.

[11] Herlihy, M., V. Luchangco and M. Moir, *Obstruction-free synchronization: Double-ended queues as an example*, in: *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems* (2003), p. 522.

[12] Herlihy, M. P. and J. M. Wing, *Linearizability: a correctness condition for concurrent objects*, TOPLAS **12** (1990), pp. 463–492.

[13] Ladan-Mozes, E. and N. Shavit, *An optimistic approach to lock-free fifo queues*, in: *Proc. of the 18th International Conference on Distributed Computing*, 2004, pp. 117–131.

[14] Lamport, L., *A theorem on atomicity in distributed algorithms*, Distributed Computing **4** (1990), pp. 59–68.

[15] Lamport, L. and F. B. Schneider, *Pretending atomicity*, Technical Report TR89-1005, DEC, SRC (1989).

[16] Lipton, R. J., *Reduction: a method of proving properties of parallel programs*, Communications of the ACM **18** (1975), pp. 717–721.

[17] Lynch, N. A., "Distributed Algorithms," Morgan Kaufmann, 1996.

[18] Michael, M. M. and M. L. Scott, *Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors*, J. Parallel Distrib. Comput. **51** (1998), pp. 1–26.

[19] Moir, M., D. Nussbaum, O. Shalev and N. Shavit, *Using elimination to implement scalable and lock-free fifo queues*, in: *Proc. 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2005)* (2005), pp. 253–262.

[20] Morgan, C., "Programming from Specifications," Prentice Hall, 1994, second edition.

[21] Morgan, C. and P. H. B. Gardiner, *Data refinement by calculation*, Acta Informatica **27** (1989), pp. 481–503.

[22] Morris, J. M., *Laws of data refinement*, Acta Informatica **26** (1989), pp. 287–308.

[23] Owicki, S. and D. Gries, *An axiomatic proof technique for parallel programs I*, Acta Informatica **6** (1976), pp. 319–340.

[24] R.K.Treiber, *Systems Programming: Coping with Parallelism. RJ5118*, Technical report, IBM Almaden Research Center (1986).

[25] Shann, C.-H., T.-L. Huang and C. Chen, *A practical nonblocking queue algorithm using compare-and-swap*, in: *Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)*, 2000, pp. 470–475.

[26] Spivey, M., "The Z Notation: A Reference Manual," Prentice-Hall International, 1988, second edition, 1992.

[27] Wang, L. and S. D. Stoller, *Static analysis of atomicity for programs with non-blocking synchronization*, in: *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (2005), pp. 61–71.

[28] Yahav, E. and M. Sagiv, *Automatically verifying concurrent queue algorithms*, in: *SoftMC 2003: Workshop on Software Model Checking*, Electronic Notes in Theoretical Computer Science **89** (2003).