A Lean 4 model of the BQ batching queue with a machine checked proof of safety

James Karlsson

September 2026

Abstract

This is a Lean 4 model of the BQ batching queue of Milman, Kogan, Lev, Luchangco and Petrank [5, 6], together with a machine checked proof that the model is linearisable [1] with respect to a sequential FIFO queue. BQ departs from an ordinary concurrent queue by letting a thread defer its enqueues and dequeues as futures, holding them back from shared memory until that thread evaluates one of them or makes a direct call. The pending operations then reach the queue together, which is what a verification has to contend with, because an entire batch must be justified at the single instant the algorithm assigns to it. Batches containing enqueues advertise themselves through an announcement installed in the head, and a thread that encounters one completes the batch on behalf of the owner, so the state any thread depends on can change beneath it between its own steps. The model spells out Listings 1 to 13 of the algorithm as a step relation with one atomic transition for every access to shared memory, which turns the interleavings the source paper reasons about in prose into ordinary sequences of transitions. Stability under helping rests on a rely condition over one global predicate and one predicate for each program counter of each thread, recording that the ghost list only grows, that a next field and an announcement link change at most once, and that the abstract head never retreats. A ghost log collects one entry at each linearisation point, and every entry holds one complete pending batch in call order. The main theorem `bq_linearisable` establishes, for arbitrary thread scripts under an arbitrary schedule, that replaying the log against the sequential specification yields exactly the queue each reachable state represents, and that every completed future and every directly returned value appears in the log with the same result. The development covers the safety argument of the correctness section of the journal version [6] and leaves the lock freedom that helping is designed to deliver outside its scope. Every result is checked in Lean 4, resting on `propext`, `Classical.choice` and `Quot.sound` as its only axioms and containing no `sorry`.

1 Introduction

A concurrent FIFO queue is one of the most heavily studied lock free data structures, and the queue of Michael and Scott [4] is the reference point against which later designs are measured. BQ [5, 6] keeps the Michael and Scott queue as its shared core and adds a batching interface on top. A thread issues its enqueues and dequeues as futures that touch no shared memory, and the whole batch of pending operations is applied to the shared queue in a single burst when the thread evaluates one of its futures or makes an ordinary blocking call. Amortising the synchronisation cost of a batch across all of its operations is what makes the design fast, and it is also what makes the design hard to reason about, because the operations in a batch do not take effect one at a time in the places a reader would expect.

The correctness condition at stake is linearisability [1]. A concurrent history is linearisable when each operation appears to take effect atomically at some instant between its invocation and its response, and the resulting order of those instants is a legal history of the sequential object. For a queue that means agreement with a sequential FIFO queue. The difficulty BQ raises is that a batch that contains an enqueue takes effect for all of its operations at once, at the single compare and swap that links the batch into the shared list, and that compare and swap may be performed by a thread other than the one that owns the batch. A helping thread that meets an installed announcement finishes the batch

before returning to its own work, so the shared state a thread has read can be changed underneath it by another thread between two of its own steps.

This article describes an executable operational model of BQ written in Lean 4 [3] and a machine checked proof that the model is linearisable with respect to a sequential FIFO queue. The contributions are the following.

- An operational model of BQ that follows Listings 1 to 13 of the algorithm, in which every access to shared memory is a single atomic step of a transition relation and every interleaving the source paper argues about informally is an ordinary sequence of those steps (Section 4).
- A machine checked linearisability theorem, `bq_linearisable`, quantified over all thread scripts and all schedules, established through a state invariant that combines one global predicate with one thread local predicate per program counter (Sections 5 and 6).
- A rely condition in the style of Jones [2] that captures what any single step is permitted to do to the state another thread depends on, together with the stability lemma `LI_stable` that shows every thread local invariant survives a step of another thread (Section 5).
- A formalisation of the combinatorial counting of failing dequeues that lets *UpdateHead* place the new head without replaying a batch (Section 7).
- A randomised differential simulator that runs the same model definitions under an arbitrary scheduler and checks the proved invariants after every step (Section 8).

The scope of the proof is safety. It shows that whatever the schedule allows, no reachable state ever disagrees with the sequential specification. The lock freedom that the announcement and helping mechanism is designed to provide, which the journal version proves in its own subsection, is not formalised here, and the model does not treat memory reclamation or the elimination variant of the algorithm. Section 10 returns to these boundaries.

2 Background

2.1 Linearisability

Linearisability [1] is the standard correctness condition for a concurrent object. Every operation is required to appear to take effect atomically at one instant, its linearisation point, lying between the operation invocation and its response, such that the sequence of operations ordered by their linearisation points is a legal history of the sequential specification. Herlihy and Wing phrase this as three requirements on the constructed sequential witness. Its operations must be a legal history of the object, the responses it gives must match the responses the concurrent execution actually returned, and its order must respect the real time order of non overlapping operations. For a FIFO queue the sequential specification is the obvious one, where an enqueue appends and a dequeue removes the oldest element or reports the queue empty.

2.2 The Michael and Scott queue

The Michael and Scott queue [4] represents the queue as a singly linked list of nodes with a shared head and tail, each carried together with a modification counter to defeat the ABA problem. An enqueue links a new node at the end of the list with a compare and swap on the next pointer of the last node and then swings the tail forward, while a dequeue advances the head with a compare and swap. A thread that finds the tail lagging behind the last node helps advance it before proceeding. BQ keeps this structure as its shared queue and layers batching on top, so the linearisation points of its non batched operations are inherited directly from the underlying design.

2.3 Rely guarantee reasoning

Reasoning about a program whose shared state is modified concurrently by other threads is the setting of rely guarantee reasoning, introduced by Jones [2]. A rely condition is a relation between the state before and after any step that another thread might take, and a proof of a thread local property is discharged by showing the property is stable, meaning it is preserved by every state change the rely condition admits. The model here uses a rely condition to isolate exactly what one step of some thread can do to the parts of the state another thread has read, and the stability lemma establishes that the thread local part of the invariant survives any such step.

3 The algorithm

BQ starts from the queue of Michael and Scott [4] and adds future operations. A thread can call *FutureEnqueue* and *FutureDequeue* as often as it likes, and nothing touches shared memory until it evaluates one of its futures or calls a plain *Enqueue* or *Dequeue*. At that point the whole pending batch goes out together. If the batch contains any enqueues it goes through an announcement installed in the head of the queue, and any thread that runs into the announcement helps finish the batch before getting on with its own work, which is what keeps the queue lock free. Figure 1 walks through the six steps of one such batch on a small queue.

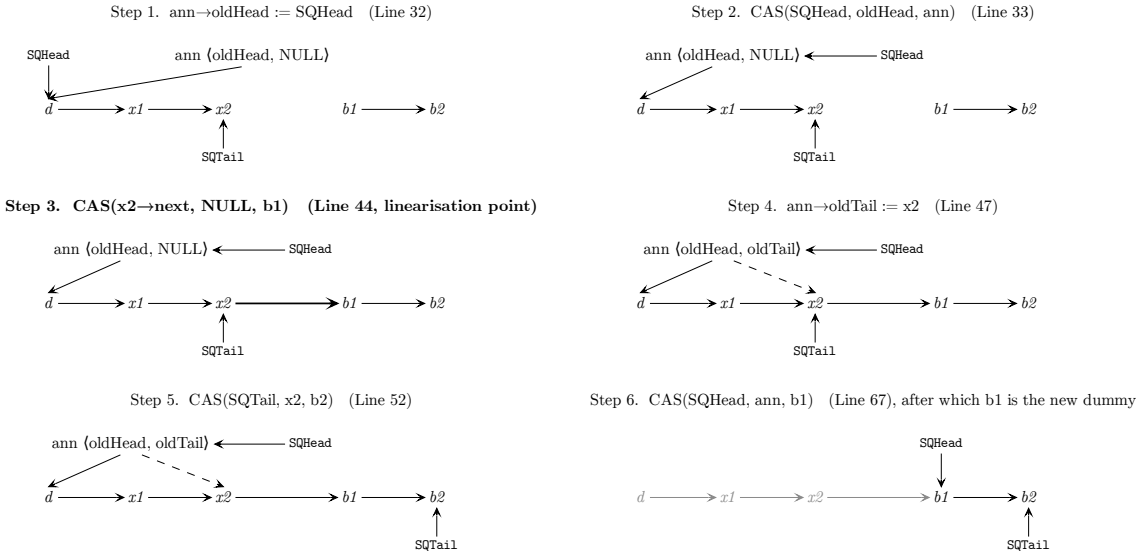

Figure 1: Running the batch D E E D D against the queue $x_1\, x_2$. The link at Line 44 is the moment the whole batch takes effect, and the new head comes straight from Corollary 3.

A batch made only of dequeues doesnt bother with an announcement. The thread just walks forward from the head over at most *deqsNum* nodes and moves the head with a single compare and swap. Thats cheaper, but it shifts the linearisation point somewhere awkward, and Section 6.2 explains why.

4 The model

Every thread carries a program counter, and `BQ.step s t` performs exactly one atomic step of thread t. A compare and swap is one step and so is a single read of a shared word, while purely local work like building the list of pending enqueue nodes is folded into whichever step comes next. Shared memory

is a map from node addresses to next pointers and items, together with the head, the tail and a table of announcements. The head holds either a pointer with its dequeue counter or an announcement, just like the *PtrCntOrAnn* union in the algorithm. A run of the queue is a script of *Enqueue*, *Dequeue*, *FutureEnqueue*, *FutureDequeue* and *Evaluate* calls for each thread, and the reachable states are those obtained from the initial state by any finite sequence of steps of any threads. Figure 2 shows every program counter and every step between them.

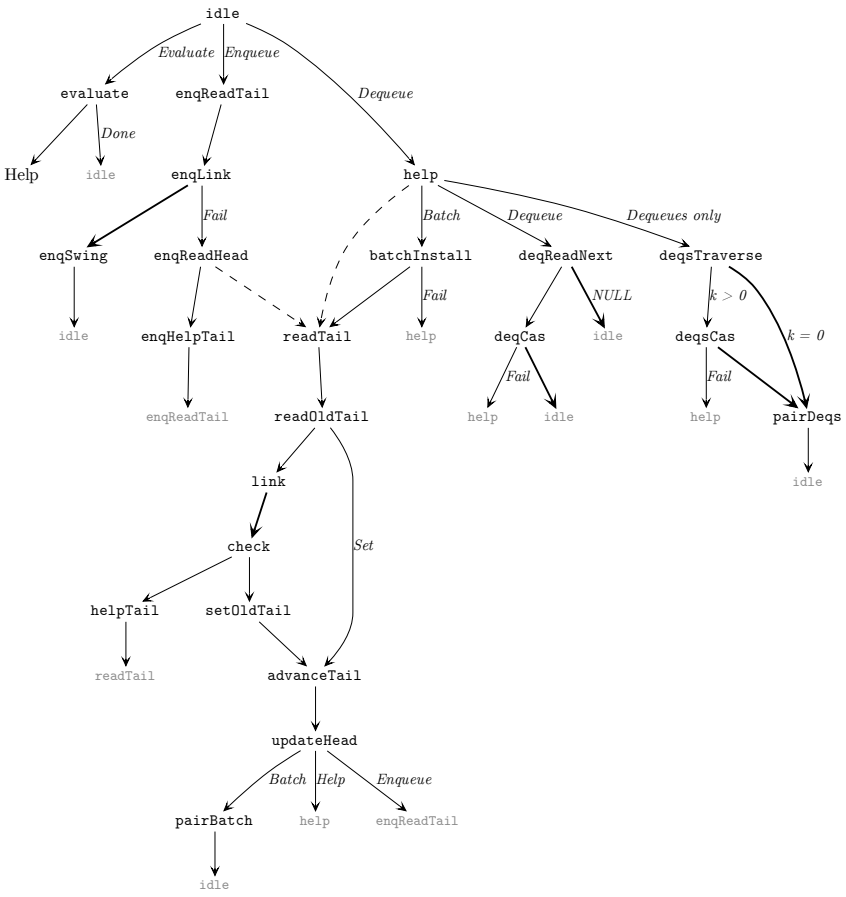

Figure 2: The transition graph of `BQ.step`. Bold edges are linearisation points and dashed edges enter *ExecuteAnn* to help another batch.

The proof needs more than the algorithm keeps track of, so the same definitions also carry some ghost state. A ghost list records every node thats ever been linked, in order, which turns the rather slippery abstract head of the algorithm into a plain index into that list and turns the queue counters into indices too. A ghost log gets one entry at every linearisation point, holding the batch along with the results a sequential queue would have given it at that moment. The abstract queue of a state is read off the ghost list by dropping its head index and mapping each remaining node to its item. Figures 3 to 9 give the control flow of each listing, with the program counter that implements every statement written underneath it.

5 The invariant and stability

The invariant is one global predicate plus one predicate for each program counter of each thread. The global predicate collects the facts that hold of the shared queue in every reachable state, among them that the ghost list is duplicate free and links up through the next pointers, that a published

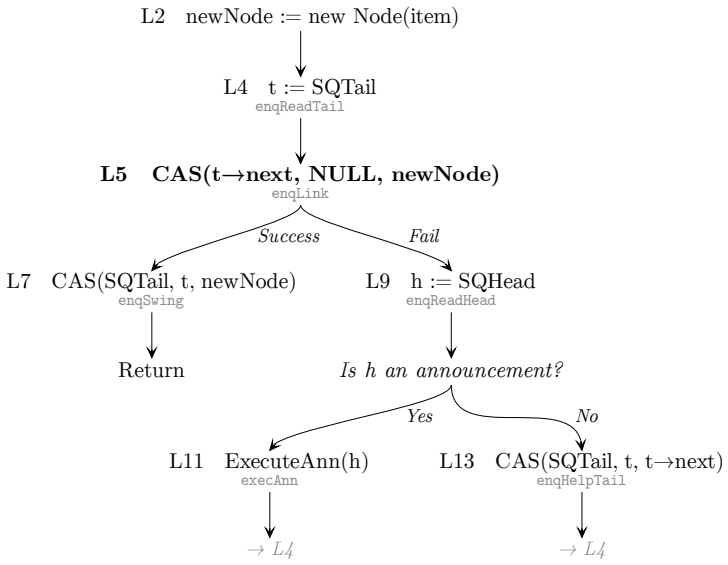

Figure 3: Listing 1, EnqueueToShared.

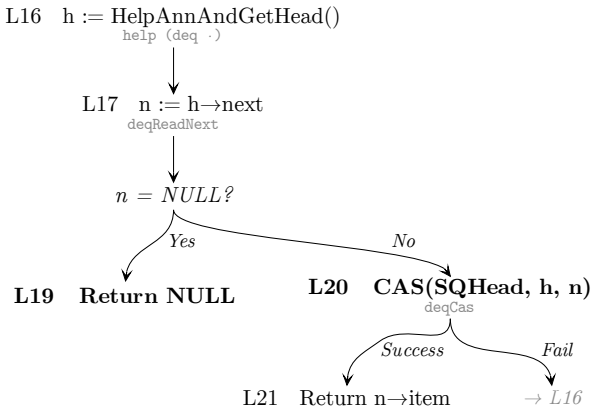

Figure 4: Listing 2, DequeueFromShared.

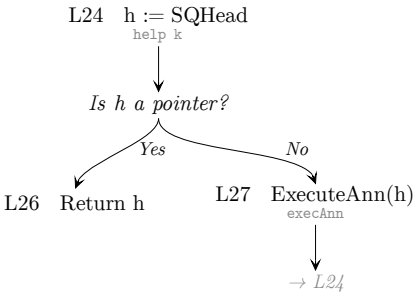

Figure 5: Listing 3, HelpAnnAndGetHead.

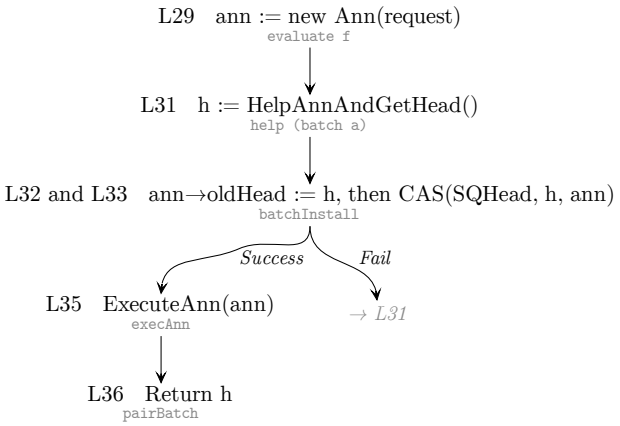

Figure 6: Listing 4, ExecuteBatch.

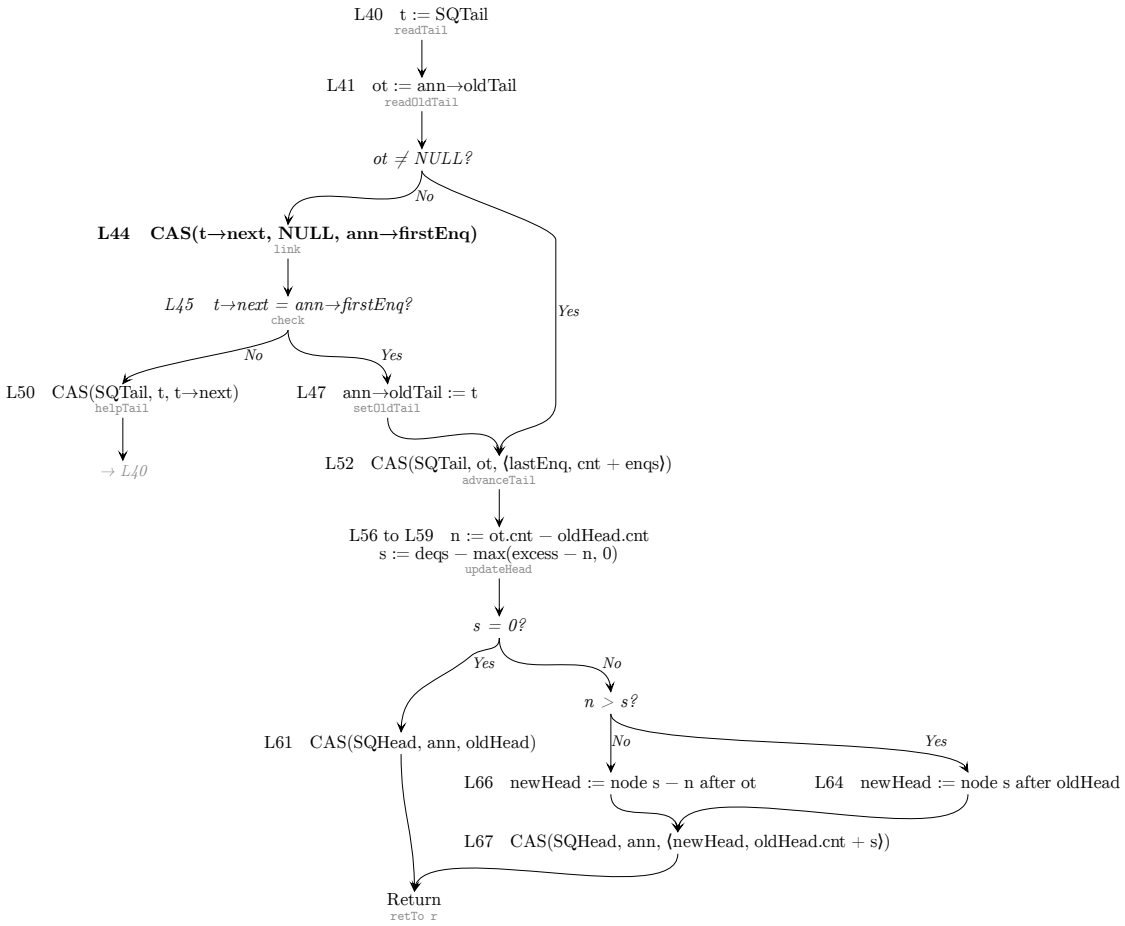

Figure 7: Listing 5, ExecuteAnn and UpdateHead, which any thread can run on behalf of whoever owns the batch.

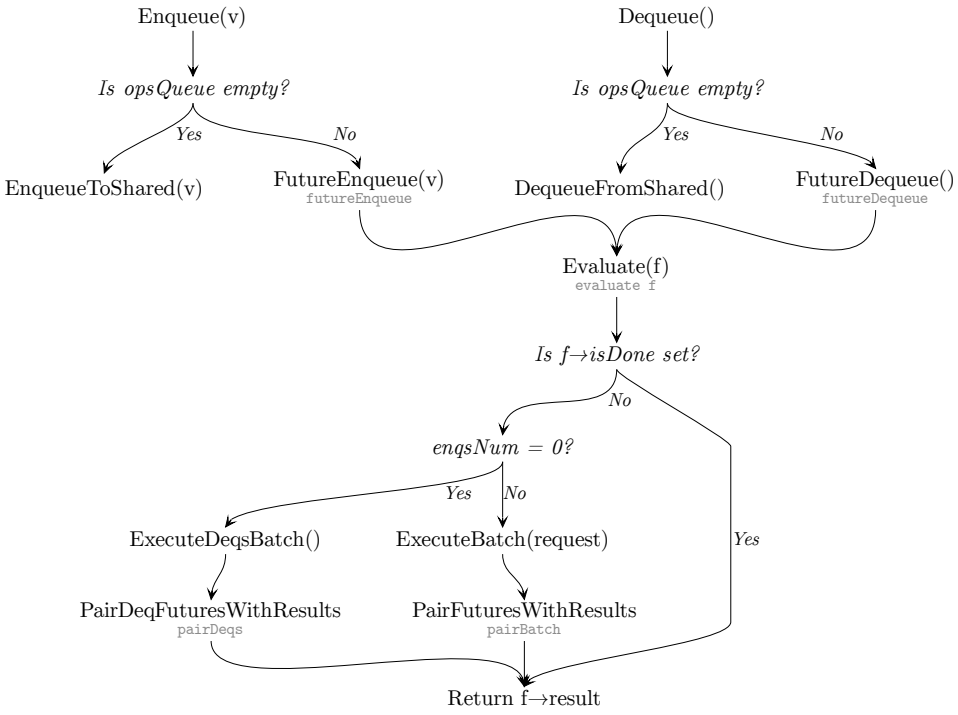

Figure 8: Listings 6 to 11, the interface methods.

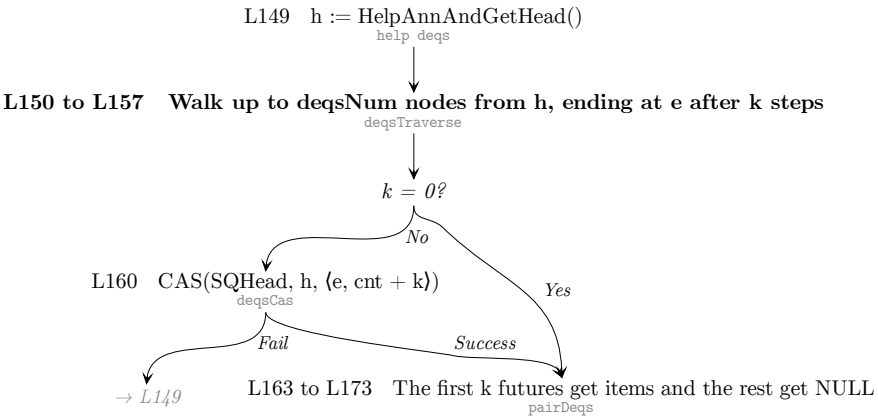

Figure 9: Listings 12 and 13, the batch made only of dequeues.

announcement carries a sound old head and a bounded excess count, and that a linked announcement fixes the position of its first and last enqueued nodes in the ghost list. It also carries the three facts that the theorem reads off directly, that the log replays to the abstract queue, that every completed future is recorded, and that every returned value is recorded. The thread local predicate pins down, for the particular program counter a thread sits at, what it must have established so far and what it is still entitled to assume about the shared state.

The trickiest part is that threads help each other, so a thread cant assume nothing changed between its own steps. A rely condition [2] pins down exactly what any step is allowed to do to the state another thread depends on. The ghost list only grows, a next field changes at most once and only from null, an announcement link is set at most once and only to the current last index, an announcement that is already published keeps its data, and the abstract head never moves backwards. `LI_stable` proves that every thread local invariant survives any step that respects those rules, which is what lets the per step proof treat the acting thread and every bystander thread separately. On top of that, the global facts `tailBound` and `order` carry Claim 8.5 and the ordering behind Claim 8.15 of the journal version, and together they make sure the link on Line 44 succeeds at most once for each batch. Figures 10 to 13 redraw the timelines the algorithm uses for these arguments.

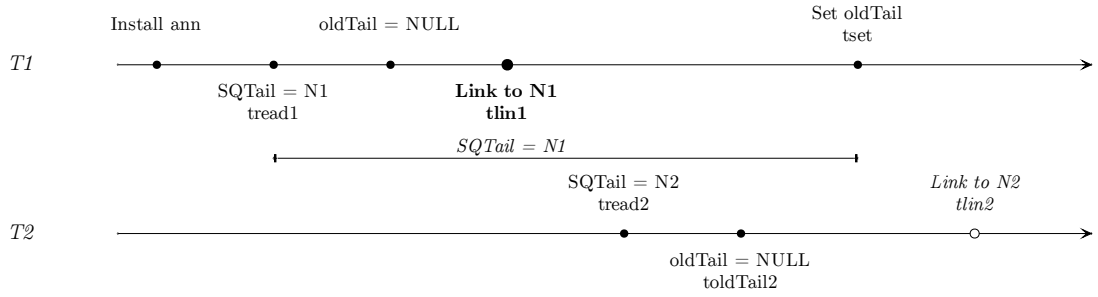

Figure 10: No batch gets linked twice (Section 8.3.3 of the journal version).

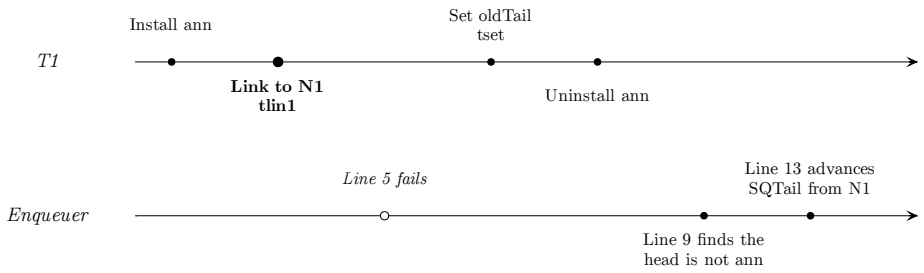

Figure 11: An enqueuer only advances the tail after tset (Lemma 8.4).

The whole argument is a single induction. The invariant holds of the initial state, and `glob_step` shows that one step of any thread from any state satisfying the invariant lands in a state that satisfies it again, so `reachable_glob` carries it to every reachable state. Each program counter contributes one case of `glob_step`, and within a case the acting thread advances by the concrete effect of its step while every other thread is carried across by `LI_stable` applied to the rely condition that the step

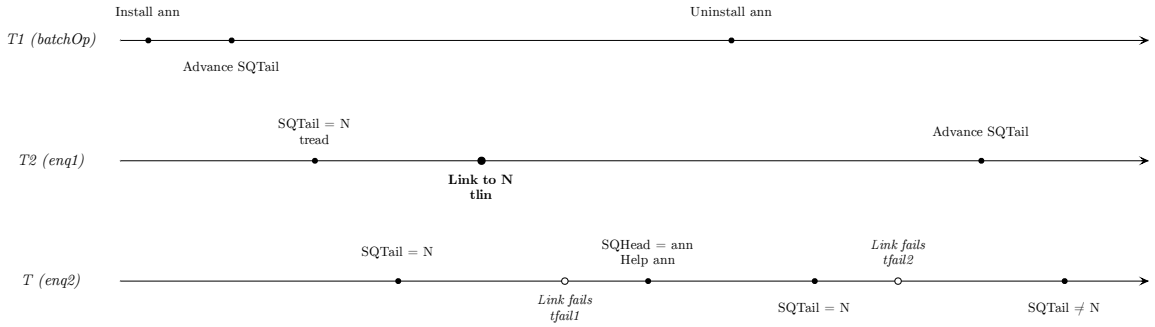

Figure 12: One enqueue failing another while an announcement is installed (Claim 8.14).

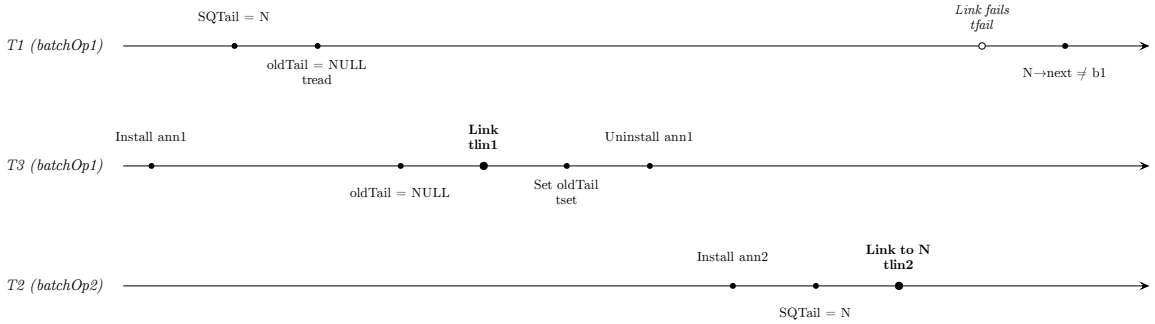

Figure 13: No batch can fail another (Claim 8.15).

satisfies.

6 Linearisability

Theorem 1 (`bq_linearisable`)**.** *Whatever the thread scripts and whatever the schedule, every reachable state s satisfies three things. The log of s replays under the sequential FIFO specification with every recorded result matching, and it finishes in exactly the queue that the shared list of s represents. Every future marked done holds the result its operation got in the log. Every value handed back by an unbatched enqueue or dequeue shows up in the log with the same result.*

The theorem falls out of the global invariant, since its three conjuncts are three of the invariant fields, and the invariant is proved of every reachable state by the induction of Section 5. Each log entry is one whole pending batch from a single thread in call order, so a batch really does take effect all at once.

Of the three things linearisability asks for [1], two are propositions above and the third comes out of the construction. Legality is the replay conjunct and agreement with what callers saw is the other two conjuncts, both of them proved. Real time order is not a separate proposition, because an entry joins the log at the step that linearises it, so the order of the log is already the order of those instants. The one case where an entry is written at a later step than the instant it belongs to is a batch made only of dequeues, and `insertAt` puts it back at the position the log had at that instant, which is what Section 6.2 is about.

6.1 Linearisation points

The linearisation points are the ones the algorithm picks. A single enqueue takes effect at the compare and swap that links its node, a dequeue at its head compare and swap or at the null read on Line 17 that reports the queue empty, and a batch containing enqueues at the compare and swap on Line 44 that links its list, no matter which thread happens to perform it. That last case is the only linearisation point that a helping thread can carry out on behalf of another, and it is the reason the invariant has to survive interference by an arbitrary helper, not just by the batch owner.

6.2 Batches made only of dequeues

These are the awkward case. A batch like this takes effect at its last next read, but that read only counts if the compare and swap that moves the head goes through later, and by then other threads may already have logged operations of their own. The model deals with this by remembering where the log was at the read and slotting the entry in at that position once the compare and swap succeeds. `replay_pure_drop` shows thats sound, because the only entries that can have crept in between are made purely of enqueues, and moving the batch in front of them doesnt change a single result. If you linearise at the compare and swap instead, the simulator starts reporting violations almost straight away, which is exactly the scenario Section 8.2 of the journal version warns about.

6.3 Pairing futures with results

Once a batch has taken effect its owner still has to hand each future its result, and it does that after the fact by walking the list. `pairLoop_spec` and `pairDeqLoop_spec` prove that the pairing loops of Listings 11 and 13 hand out exactly what the specification says, including the rather unstable read of the next field of the last enqueued node that Listing 11 makes.

7 Counting failing dequeues

Section 5.2 of the algorithm gets its own proofs. For a batch B, write $d(B)$ for its number of dequeues and $x(B)$ for its number of excess dequeues, meaning the ones that would fail on an empty queue.

Lemma 2 (`lemma_5_3`)**.** *$x(B)$ equals the largest value, over all prefixes P of B, of the number of dequeues in P minus the number of enqueues in P.*

Corollary 3 (`claim_5_4`, `corollary_5_5`)**.** *Run against a queue of length n, the batch B has* $\max(x(B) - n, 0)$ *failing dequeues and so* $d(B) - \max(x(B) - n, 0)$ *successful ones.*

UpdateHead uses exactly this count to find the new head without replaying the batch. `listing9_excess` shows that the counter FutureDequeue keeps as it goes computes $x(B)$, and `example_EDDEEDDDEDDEE` checks the worked example from the algorithm, which has three excess dequeues.

8 The simulator

The same definitions that the proof reasons about are also executable, and a small driver runs them under a random scheduler. It builds a script for each thread, then repeatedly picks a thread and applies one step, and after every step it checks the proved invariants against the running state, among them that the log still replays to the abstract queue and that every completed future and every returned value is recorded. A run of two hundred schedules of four threads making thirty calls each covers on the order of a hundred and ninety thousand steps and ten thousand log entries, exercising the helping path tens of thousands of times and the retroactive dequeue batch linearisation a few dozen times, and it reports no violation. The simulator is not part of the proof, but because it shares the model definitions it is a check on the model itself rather than on a separate reimplementation, and it is how the alternative linearisation point of a dequeues only batch was found to fail.

9 Trust base

Every result in the development is proved with no appeal to any axiom beyond the three that Lean 4 [3] treats as standard, namely propositional extensionality, the axiom of choice and the soundness of quotients. The main theorem `bq_linearisable` depends on exactly `propext`, `Classical.choice` and `Quot.sound`, and no file contains a `sorry` or introduces an axiom of its own. The specification the model is proved against is a few lines of ordinary functional code for a sequential FIFO queue, small enough to read and agree with by inspection, and the theorem relates the model to that specification rather than to a second machine assisted artefact.

10 Conclusion

The model turns the informal interleaving arguments of the BQ correctness proof into an induction over a transition relation, and the resulting linearisability theorem holds for every schedule with the helping mechanism treated in full. The boundaries are deliberate. The proof is a safety proof, so the lock freedom that the announcement and helping mechanism exists to provide, which the journal version establishes in its own subsection through the backward branch bound of Claim 8.14, is stated but not formalised. The model works over unbounded node addresses and does not treat memory reclamation, and it covers the base algorithm of Listings 1 to 13 rather than the elimination variant. Each of these is a natural direction for extending the development, and the rely condition and per program counter invariant are the parts most likely to carry over to them.

References

[1] M. P. Herlihy and J. M. Wing. *Linearizability, A Correctness Condition for Concurrent Objects.* ACM Transactions on Programming Languages and Systems 12(3), pages 463 to 492, 1990. `https://doi.org/10.1145/78969.78972`

[2] C. B. Jones. *Tentative Steps Toward a Development Method for Interfering Programs.* ACM Transactions on Programming Languages and Systems 5(4), pages 596 to 619, 1983. `https://doi.org/10.1145/69575.69577`

[3] L. de Moura and S. Ullrich. *The Lean 4 Theorem Prover and Programming Language.* In Automated Deduction, CADE 28, pages 625 to 635, 2021. `https://doi.org/10.1007/978-3-030-79876-5_37`

[4] M. M. Michael and M. L. Scott. *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.* In Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC 1996), pages 267 to 275. `https://doi.org/10.1145/248052.248106`

[5] G. Milman, A. Kogan, Y. Lev, V. Luchangco and E. Petrank. *BQ, A Lock Free Queue with Batching.* In Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2018), pages 99 to 109. `https://doi.org/10.1145/3210377.3210388`

[6] G. Milman-Sela, A. Kogan, Y. Lev, V. Luchangco and E. Petrank. *BQ, A Lock Free Queue with Batching.* ACM Transactions on Parallel Computing 9(1), pages 1 to 49, 2022. `https://doi.org/10.1145/3512757`