Verifying Linearisability with Potential Linearisation Points

John Derrick[1], Gerhard Schellhorn[2], and Heike Wehrheim[3]

[1] Department of Computing, University of Sheffield, Sheffield, UK
J.Derrick@dcs.shef.ac.uk
[2] Universität Augsburg, Institut für Informatik, 86135 Augsburg, Germany
schellhorn@informatik.uni-augsburg.de
[3] Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany
wehrheim@uni-paderborn.de

Abstract. Linearisability is the key correctness criterion for concurrent implementations of data structures shared by multiple processes. In this paper we present a proof of linearisability of the *lazy* implementation of a set due to Heller et al. The lazy set presents one of the most challenging issues in verifying linearisability: a linearisation point of an operation set by a process other than the one executing it. For this we develop a proof strategy based on refinement which uses *thread local* simulation conditions and the technique of *potential* linearisation points. The former allows us to prove linearisability for arbitrary numbers of processes by looking at only two processes at a time, the latter permits disposing with reasoning about the past. All proofs have been mechanically carried out using the interactive prover KIV.

1 Introduction

The setting of this work are data structures such as sets, stacks and queues that are shared by parallel processes. To increase the opportunities for concurrency (particularly relevant in a multicore context), implementations of these, so called, *concurrent objects* usually apply fine-grained synchronisation schemes for access. Fine-grained synchronisation disposes with locking the whole data structure during access, and locks only single elements (e.g., nodes in a linked list representation). The extreme to this are implementations of operations taking no locks at all.

Such highly concurrent algorithms are intrinsically difficult to prove correct, the down-side of the performance gain from permitting concurrency is the much harder verification problem: how can one verify that the implementation of a concurrent object is correct? Here, the key correctness property to be shown is *linearisability* [11]. It permits one to view operations on concurrent objects as though they occur atomically, in some sequential order [11]:

> Linearisability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response.

This "point" in between invocation and response of an operation is referred to as the *linearisation point* (LP).

A number of different techniques have been employed to verify linearisability, ranging from shape analysis [1], separation logic and rely-guarantee reasoning [21] to simulation-based methods. The concurrent algorithm considered in this paper (the lazy set of Heller et al. [9]) poses a particular challenge for verification: the linearisation point for one of the operations does not coincide with the execution of an instruction of the source code, but rather can be set by a process other than the one executing the operation. As a consequence, the real LP of this operation is only known when it finishes. This has lead to the development of a number of proof techniques for the lazy set looking into the *past*: the first approach in [20] argues that knowing the outcome of this operation its linearisation point can be found, later approaches use backward simulation proofs [3] or "hindsight" techniques [16].

In this paper we propose a new technique for verifying linearisability of the lazy set which avoids having to look into the past. The technique extends our previous approach [4,5] to cope with the class of algorithms like the lazy set. In general, we carry out a proof of *refinement*: the concurrent implementation is shown to be a valid refinement of the abstract data structure. Our proof principle consists of two levels: we have *local* (i.e., thread modular) *simulation conditions* which need to be verified for the concurrent implementation at hand, and a *general theory* which links the local conditions with linearisability and thus shows their soundness.

Both levels have been formally verified with KIV using standard higher-order logic. A web presentation with all details can be found at [12]. Unfortunately we are not able to describe the global part of the theory in this paper, where we focus on the local simulation conditions and their application to the lazy set.

The key idea of the local conditions is to define *potential* linearisation points, which solve the issue of LPs set by other processes. The next section gives our running example of the lazy set. In section 3 we introduce our refinement technique, and in section 4 we show how we can derive local proof obligations that can cope with the type of algorithm exemplified by our running example. In Section 5 we discuss how these proof obligations can be discharged for this implementation. Finally, Section 6 gives related work and concludes.

2 The Lazy Concurrent Set

Our running example is a concurrent implementation of a set data structure and its access operations. The abstract data type $A = (AS, ASInit, (AOp_p^i)_{i \in I, p \in P})$ uses a finite set of integers as abstract state: $AS \mathrel{\widehat=} [set : \mathbb{F}\,\mathbb{Z}]$. The set is initially empty (*ASInit*), and then allows for three operations, $I = \{1, 2, 3\}$ executed by processes $p \in P$: integers can be *added*, *removed*, and we have a test of containment: *contains*. Abstractly, all these operations are atomic. They all return a boolean result: *add* and *remove* return *true* if the set has been changed.

Here, we study the highly concurrent implementation proposed in [9]. The set is implemented by a sorted linked list. Its elements appear in the nodes of

```
add(e):
  A1 : n1, n3 := locate(e);
  A2 : if n3.val != e then
  A3 :    n2 := new Node(e);
  A4 :    n2.next := n3;
  A5 :    n1.next := n2;
  A6 :    res := true;
  A7 : else res := false
       endif ;
  A8 : n1.unlock();
  A9 : n3.unlock();
  A10: return res
```

```
remove(e):
  R1 : n1, n2 := locate(e);
  R2 :  if n2.val = e then
  R2b:    n2.mrk := true;
  R3 :    n3 := n2.next;
  R4 :    n1.next := n3;
  R5 :    res := true;
  R6 : else res := false
       endif;
  R7 : n1.unlock();
  R8 : n2.unlock();
  R9 : return res
```

Fig. 1. Operations *add* and *remove*

the list in a strictly increasing order. The list has two sentinel nodes: *head* with value $-\infty$ and *tail* with value ∞. Every node has a `val` field with an integer, a `nxt` field for the pointer to the next node and a `mrk` bit (used to mark nodes that as logically deleted).

In the algorithm, atomicity is given up as to allow for concurrency. Concurrency here means that several processes should be able to execute operations on the set at the same time, thus the steps of operations of the algorithm in Fig. 1 can be interleaved. To cope with this interleaving, each node in the list is associated with a lock. Operations `n.lock()` and `n.unlock()` lock and unlock a node `n`.

Operations *add* and *remove* rely on an additional operation *locate* (see Fig. 2) which finds the appropriate position of the element (to be added or removed) and then locks the two adjacent nodes. Operation *add* then checks whether the second locked node already contains the new element to be added, and if not, creates a new node and inserts it (by redirecting the pointer of the previous node) into the list. Operation *remove* proceeds in two steps (when *locate* has found the element to be removed): first, it will *mark* the node as deleted using the `mrk` bit in line R2b[1] (lazy). Then it will physically remove the node by redirecting pointers. Both *add* and *remove* unlock the nodes returned by *locate* at the end.

The locking scheme of operation *locate* (see Figure 2) is an *optimistic* one: while traversing the list in search of the element, it does not lock nodes. Only when the correct position has been found, the previous and current node is locked. Since these nodes might have been removed by other processes while the search loop was running, the *locate* always *validates* the found candidates. Validation has to check that neither of the locked nodes have already been logically deleted (`mrk` bit set), and that the nodes are still adjacent. If this fails, locate has to be restarted. Note that the marking bit is used to ensure that removal can be done as one atomic step.

Finally, the most interesting operation is *contains*. The implementation of contains is *wait-free* and uses no locks at all. It searches for the element itself

[1] Line R2b is the only modification of *remove* compared to the pessimistic version studied in [5].

```
locate(e):
  while (true) {
L1:  pred := Head;
L2:  curr := pred.next;
L3:  while (curr.val < e) {
L4:      pred := curr;
L5:      curr := curr.next; }
L6:  pred.lock();
L7:  curr.lock();
/* validate */
L8 : if ! pred.mrk
L9:     and ! curr.mrk
L10:    and pred.next = curr
L11: then return pred, curr
L12: else { pred.unlock()
L13:        curr.unlock(); }
} /* end of while(true) */
```

```
contains(e):
I1: curr := Head;
I2: while (curr.val < e)
I3:      curr := curr.next;
I4: if curr.mrk then res:= false;
I5: else res := (curr.val = e)
I6: return res
```

Fig. 2. Operations *locate* and *contains*

(without use of *locate*) and also checks for the `mrk` bit. It is this omission of locking combined with the lazyness of *remove* which makes verification of linearisability hard.

Our proof technique given in the next section relies on a proof of linearisability via *refinement* of the abstract type defined above to a concrete data type $C = (CS, CSInit, (COp^j_p)_{j \in J, p \in P})$ that we define now as a Z specification.

We start with modelling the global heap *mem*, which is a partial function from a basic type *Ref* of references (with $null \in Ref$) to memory cells: cells consist of a value of type $\mathbb{Z}$ (plus $-\infty, \infty$), can be locked by a process from a set P, can be marked and have a (potentially null) reference to the next node. To access these components of a cell with address r we write $r.val$, $r.lck$, $r.mrk$ and $r.nxt$, respectively. The heap *mem* together with the *head* reference forms the global state GS. Initially, the global state ($GSInit$) just consists of a list with head and tail node.

The local state of one process LS consists of the tuple of the local variables of the algorithms, together with a type $pc : PC$ for the program counter. Its initial state, given by $LSInit$ (not shown) has $pc = 1$ to indicate that no operation is running. All other values of the initial state are unused, so they can be arbitrary.

The complete concrete state space CS is defined by combining GS with a local state function assigning a local state $lsf(p)$ to every process $p \in P$.

$$\begin{array}{|l}
\multicolumn{1}{l}{GS} \\ \hline
head : Ref \\
mem : Ref \nrightarrow \\
\qquad (\mathbb{Z} \cup \{-\infty, \infty\}) \times \\
\qquad (P \cup \{none\}) \times \mathbb{B} \times Ref \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{GSInit} \\ \hline
GS' \\ \hline
n_h, n_t \in Ref \\
mem' = \{n_h \mapsto (-\infty, none, false, n_t), \\
\qquad n_t \mapsto (\infty, none, false, null)\} \\
head' = n_h \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{LS} \\ \hline
n1, n2, n3 : Ref \\
curr, pred : Ref \\
res : \mathbb{B}, e : \mathbb{Z}, pc : PC \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{CS} \\ \hline
GS \\
lsf : P \rightarrow LS \\ \hline
\end{array}$$

To define the concrete operations, we first define operations COP_j on one local state. For this, each line of the algorithms is turned into one Z operation. The following gives the Z specification of lines I1 and I4 of *contains*[2].

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{containsI}1} \\
\hline
\Xi GS \\
\Delta LS \\
\hline
pc = I1 \wedge pc' = I2 \\
curr' = head \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{containsI}4} \\
\hline
\Xi GS \\
\Delta LS \\
\hline
pc = I4 \wedge curr.mrk \wedge pc' = I6 \\
\neg res' \\
\hline
\end{array}$$

These operations are then promoted (using the same standard schema as in [5]) to operations COp_p^j on CS for each process $p \in P$, which work on the local state $lsf(p)$. Initialisation *CSInit* of the concrete state space is defined similarly.

3 Linearisability and Refinement

Linearisability requires that operations should appear as taking place atomically, i.e., take effect instantaneously at some point in time, even though the atomicity of operations has been given up in the implementation and an actual concrete execution might be an arbitrary interleaving of steps from the above algorithm. This "point in time" is the *linearisation point* (LP). Linearisability permits one to view operations on concurrent data structures as though they occur in some sequential order, namely the order of their linearisation points.

Our proof technique introduced in [4] and further elaborated in [5] relies on a proof of linearisability via *refinement*. Basically, we show that the concurrent implementation $C = (CS, CSInit, (COp_p^j)_{j \in J, p \in P})$ is a *non-atomic refinement* [6] of the abstract data type $A = (AS, ASInit, (AOp_p^i)_{i \in I, p \in P})$.

Here, non-atomic means that a step of the concrete data type COp_p^j that is part of the implementation of AOp_p^i can either match an empty step *skip* or an execution of AOp_p^i. Basically, the steps representing linearisation points have to match with the abstract operations, and all other steps correspond to *skip* steps. To do so, we first of all have to determine the linearisation points. For some simpler classes of algorithms (e.g., stack and non-lazy set considered in [5]), LPs can be determined from the current state of a process (basically, its program counter), and in our methodology are fixed by defining a so-called *status* function assigning values from a type *STATUS*:

$$STATUS ::= IDLE \mid IN \mid OUT$$

Therein, *IDLE* represents an idle process, and *IN* and *OUT* describe the status of processes being before and after their linearisation points, respectively. With the help of the *status* function we define specific status-dependent proof obligations in [4,5]. The proof obligations are *local*, i.e., they do not consider all the

[2] We use the Object-Z approach and mention only those variables which are changed.

processes, but only two specific processes p and q. p is executing a step, and q represents an arbitrary other process, which might be affected. Such local proof obligations are possible for many linearizable algorithms, where typically it does not matter for one process *which* other process affects the global state, but only *how* the state is affected. This is true also for the lazy set, where the only relevant information a process sees from others is new cells being introduced or old cells being marked.

The proof obligations in [5] are particular instances of forward simulations. The status tells us whether an individual concrete step has to be matched with a *skip* or an operation of the abstract data type in the simulation. They prove that the concurrent implementation is a non-atomic refinement of an abstract data type (given that the LPs can be defined this way). In a second step, it has to be proven that this kind of non-atomic refinement actually shows linearisability (the *general theory*). Both the linearisability proofs for concrete data structures and the general proof of soundness of our refinement theory have been mechanically conducted using the interactive prover KIV [17]. None of the other approaches for verifying linearisability has a mechanised proof that their proof obligations imply Herlihy and Wing's original definition of linearisability [11].

For our case study, the proof obligations of [5] are sufficient to verify the *add* and *remove* operation, where the LP can be identified in the code. E.g., for *add* the LP is either A5 or A7 (for return value *true* and *false*).

However, this technique is not applicable to the *contains* operation (which represents a whole class of similar concurrent operations). The issue is that it is not possible to *statically* determine the linearisation point of *contains* as it depends on future behaviour of processes *other* than the one currently executing *contains*.

An example can make this clearer. Consider the list representation of the set $\{2, 4, 6\}$ in Figure 3 (a). Assume that *contains(4)* has been started and executed its while loop reaching I4. At this point, variable *curr* points to node 4 (see figure). If the next step executed is I4, *contains* would return *true* and the LP could have been the last I3, setting variable *curr*.

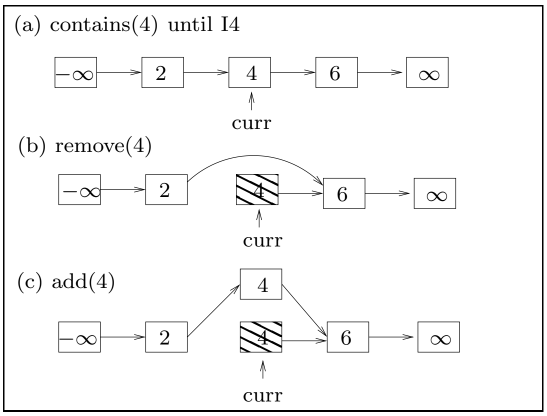

Fig. 3. Sample execution: *contains*(4) until I4 (a), then *remove*(4) (b), then *add*(4)

However, assume that we do not take I4 next, but start another process executing *remove*(4) (completing without any interleaving of operations from other processes). At the end of *remove* we reach situation (b) of Figure 3, leaving the node *curr* pointing at a marked node. If we would now execute I4 next, *contains* would return *false*. Thus taking I3 as LP is wrong (at this point the return value for contains would have been *true*). So let us assume, we choose I4 as LP. This might however still be incorrect: if the next operation is *add*(4) which starts and completes (bringing us into situation (c)), executing I4 would still give us the wrong return value: now 4 is in the set again, so at this point in time it is not correct for *contains*(4) to return *false*.

It turns out that for the sequence *contains*(4) (until I4), *remove*(4) (completely), *add*(4) (completely), *contains*(4) (rest), the only valid linearisation point for *contains* is directly after the LP of *remove*. It gives the following valid sequence of abstract operations: $\langle \mathit{remove}(4, \mathit{true}), \mathit{contains}(4, \mathit{false}), \mathit{add}(4, \mathit{true})\rangle$. However, not every *remove* is a linearisation point for a running *contains*. It crucially depends on **where** the *contains* currently is, and whether some more *add*s will appear **in the future** or not.

Such a case could not be tackled by our current technique, and for the lazy set we need a proof technique which can show linearisability for (a) operations whose linearisation point is set by another process and (b) are determined by future operations. Moreover there is additional complexity in this example, and we also need a technique for situations whereby (c) one step in the implementation can linearise multiple operations (the *remove* can potentially set the LPs of all running *contains*).

4 Local Proof Obligations

The proof obligations have to guarantee that the concrete data type is a refinement of the abstract data type. This is usually shown by defining an abstraction function ($Abs : GS \rightarrow AS$) between concrete and abstract state space, and then showing that initialisation and operation execution of concrete and abstract data type match in a certain way (*simulation*).

Again, we aim at *local* proof obligations, which just consider local states *lsp* and *lsq* of two representative processes *p* and *q*. Process *p* is executing a step of its algorithm, and process *q* might be affected by having to execute its linearisation point (the case in question being process *p* marking a cell, while process *q* searches for its value).

Coping with potential linearisation points: To tackle this issue we need to generalise our status function, with a new status *INOUT* to cover the situation in which an operation has *potentially* linearised (the types in brackets describe types of inputs and outputs). Thus for our example, a process *p* with status *INOUT*(3, *true*) is a process which is potentially after its LP, has 3 as input and will return *true*.

$$\mathit{STATUS} ::= \mathit{IDLE} \mid \mathit{IN}\langle\langle \mathbb{Z} \rangle\rangle \mid \mathit{OUT}\langle\langle \mathbb{B} \rangle\rangle \mid \mathit{INOUT}\langle\langle \mathbb{Z} \times \mathbb{B} \rangle\rangle$$

For every implementation, we need to define a status function

$$status : GS \times LS \rightarrow STATUS$$

assigning a status to a process with local state $ls \in LS$ and current global state $gs \in GS$. The status of a process can change several times during execution of an operation. In particular, several status changes between $INOUT(e, true)$ and $INOUT(e, false)$ are possible if another process executes a step which affects the outcome. Every status change from IN to $INOUT$, $INOUT(e, true)$ to $INOUT(e, false)$ (and vice versa) and $INOUT$ to OUT is a potential linearisation point and has to match with the corresponding abstract operation. It may seem odd that due to the status changes several abstract *contains* appear in a thus constructed run. However, this is sound as *contains* is not modifying the set: the last operation that affects the output value of the status executed in a run is the linearisation point.

Defining the invariants: As in [5], in addition to the abstraction function our theory requires a *local invariant* INV on $GS \times LS$ to capture constraints which are always valid in our linked list implementation (e.g., that tail is always reachable from head). Last, a *disjointness* predicate D over the local states of p and q serves the purpose of keeping disjointness information about local states.

Defining the non-atomic simulation conditions: As in standard simulation conditions, our local proof obligations need to match the behaviour of the concrete and abstract operations. Since we do not have a 1-1 correspondence of abstract and concrete operation anymore, and furthermore, a concrete operation can linearise several processes, and thus match with more than one abstract operation, we have to capture different cases in our simulation conditions. The latter point requires an extension to the theory developed in [5]. Basically, four different types of matchings can occur, each being accompanied by particular status changes.

The most basic type is the classic simulation diagram: process p executes some concrete operation COp_p (bringing us from state cs to cs'), which is the linearisation point, and matches with abstract operation AOp_p (going from abstract as to as') with input in and output out. Concrete and abstract states are related via the abstraction function Abs. The left hand side of Figure 4 describes this case. When process p executes a *potential* linearisation point, both linearisation as well as a *skip* step must be possible. Therefore in this case the abstract state is not allowed to change, as shown on the right hand side. The right hand

$$\begin{tikzcd}
\text{as} \arrow[r, "AOp_p"] & \text{as}' \\
\text{cs} \arrow[u, "\text{Abs}"] \arrow[r, "COp_p"'] & \text{cs}' \arrow[u, "\text{Abs}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{as} \arrow[loop above, "AOp_p \vee skip"] & \\
\text{cs} \arrow[u, "\text{Abs}"] \arrow[r, "COp_p"'] & \text{cs}' \arrow[ul, "\text{Abs}"']
\end{tikzcd}$$

Fig. 4. Simulation types 1 and 2

$$\begin{tikzcd}
\mathrm{as} \arrow[r, "AOp_p"] & \mathrm{as}' \arrow[loop above, "AOp_q"] \\
\mathrm{cs} \arrow[u, "\mathrm{Abs}"] \arrow[r, "COp_p"'] & \mathrm{cs}' \arrow[u, "\mathrm{Abs}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathrm{as} \arrow[loop above, "(AOp_p \vee skip);\ AOp_q"] & \\
\mathrm{cs} \arrow[u, "\mathrm{Abs}"] \arrow[r, "COp_p"'] & \mathrm{cs}' \arrow[ul, "\mathrm{Abs}"']
\end{tikzcd}$$

Fig. 5. Simulation types 3 and 4

diagram (with *skip*) is also used when the concrete step does not execute an LP. No processes other than p are affected in the two cases of Fig. 4.

The next two types (in Fig. 5) consider the case where a step of a process p (possibly) linearises itself as well as linearises a process q. The left diagram of Figure 5 shows the case where the execution of operation *COp* of process p definitely sets its own as well as the linearisation point of process q. Thus the simulation has to guarantee that abstractly the operation of p and q is possible. The right hand side depicts the case where the abstract operation of process p is either no or a potential LP for p, and is therefore not allowed to change the abstract state. Both cases require, that process q does not change the abstract state. This allows to lift the proof to a global scenario, where p linearises a number of operations $q_1, \ldots, q_n$, since their abstract operations can all start in the same state.

The simulation conditions have to formalise all these cases. In these, both the status of p and q are used for deciding which case applies, i.e., which kind of matching to show. Instead of writing several simulation conditions, one for each possible status change, we accumulate all cases in one condition using a so-called *exec* function. This function takes as an input the status of a process before and after executing some operation *COp* ($stat$, $stat'$), the corresponding abstract states as and as', and the index i of the operation currently being run. From this, it determines the verification condition to be shown.

$$\begin{aligned}
& exec(stat, i, stat', as, as') := \exists\, in, in', out, out' \bullet \\
& \quad (stat = IN(in) \wedge stat' = OUT(out') \wedge AOp^i(in, as, as', out')) \\
& \quad \vee (stat = IN(in) \wedge stat' = INOUT(in', out') && (*) \\
& \qquad\qquad \wedge\ AOp^i(in, as, as', out') \wedge in = in' \wedge as = as') \\
& \quad \vee (stat = INOUT(in, out) \wedge stat' = OUT(out') \\
& \qquad\qquad \wedge\ AOp^i(in, as, as', out') \vee (as = as' \wedge out = out')) \\
& \quad \vee (stat = INOUT(in, out) \wedge stat' = INOUT(in', out') \\
& \qquad\qquad \wedge as = as' \wedge (AOp^i(in, as, as', out') \vee out = out') \wedge in = in') \\
& \quad \vee (stat = IN(in) \wedge stat' = IN(in') \wedge as = as' \wedge in = in') \\
& \quad \vee (stat = INOUT(in, out) \wedge stat' = IN(in') \Rightarrow as = as' \wedge in = in') \\
& \quad \vee (stat = OUT(out) \wedge stat' = OUT(out') \wedge as = as' \wedge out = out')
\end{aligned}$$

As an example, consider case (*) in the definition of *exec*. If the status of a process changes from *IN* to *INOUT*, i.e., from before to potentially after the linearisation point, then a corresponding abstract operation must be executed which does not change the abstract state and gets exactly the same input and output as those in the $INOUT$ status. This ensures that e.g., a *contains* with return value *false* cannot match with an abstract *contains* returning *true*.

The case following (*) gives two possibilities for going from $INOUT(in, out)$ to $OUT(out')$. Either the potential linearisation is made permanent ($as = as' \wedge out = out'$), or the potential linearisation is discarded and a new one is established by executing AOp. In general, abstract state changes in AOp are allowed when the operation definitely linearises by setting status to $OUT(out')$.

This lets us finally define the simulation condition. Herein, we use a function *runs* which returns the (index of the) abstract operation a process in local state LS is currently executing (this can be determined from the value of the *pc*).

$$\begin{array}{ll}
\forall\, gs, gs' : GS, lsp, lsq, lsp', lsq' : LS \bullet & \\
\quad INV(gs, lsp) \wedge INV(gs, lsq) \wedge D(lsp, lsq) \wedge COp_p^j(gs, lsp, gs', lsp') & \\
\quad \Rightarrow & (LPO) \\
\quad INV(gs', lsp') \wedge INV(gs', lsq) \wedge D(lsp', lsq) & \\
\quad \wedge\ exec(status(gs, lsp), runs(lsp), status(gs', lsp'), Abs(gs), Abs(gs')) & \\
\quad \wedge\ exec(status(gs, lsq), runs(lsq), status(gs', lsq), Abs(gs'), Abs(gs')) &
\end{array}$$

Basically, (LPO) requires to show that (a) the invariant and the disjointness properties are kept when a concrete operation is executed, and (b) the appropriate matching as defined by *exec* can be carried out for both p and q. Please note that lsq is left unchanged by COp_p. Since (LPO) just refers to two local states lsp and lsq , but never to the complete concrete state CS, we have obtained a *local proof obligation.*

In addition to this simulation rule, we have two simpler proof obligations considering the special cases of invocation and return steps. These disallow abstract state changes and status changes of q (no linearisation). The status of p is required to change from $IDLE$ to $IN(in)$ and from $OUT(out)$ to $IDLE$ with the correct input resp. output value of COp_p. Due to lack of space we will not give them here. We also omit the simple initialisation conditions.

5 Verification of the Case Study

Verification of the case study requires to instantiate the predicates and functions used in the proof obligation (LPO) . We start with the status function:

$$\begin{array}{l}
ls.pc = 1 \Rightarrow status(gs, ls) = IDLE \\
ls.pc \in \{A1, \ldots, A5, A7, R1, R2, R2b, R6\} \Rightarrow status(gs, ls) = IN(e) \\
ls.pc \in \{A6, R3, R4, R5\} \Rightarrow status(gs, ls) = OUT(true) \\
ls.pc \in \{A8, A9, A10, R7, R8, R9\} \Rightarrow status(gs, ls) = OUT(res) \\
ls.pc = I1 \Rightarrow status(gs, ls) = IN(e) \\
ls.pc \in \{I2, I3, I4\} \Rightarrow status(gs, ls) = INOUT(e, \\
\qquad\qquad\qquad \exists\, r.reachable(curr, r, mem) \wedge r.val = e \wedge \neg\, r.mrk) \\
ls.pc = I5 \Rightarrow status(gs, ls) = OUT(curr.val = e) \\
ls.pc = I6 \Rightarrow status(gs, ls) = OUT(res)
\end{array}$$

The definition gives the LPs of the *add* algorithm as A5 (for $res = true$) and A7. Before and at this point in the algorithm the status is $IN(e)$, after it $OUT(res)$. Similarly, the LPs for *remove* are the marking operation at R2b when *true* is returned, and the negative case of R2 for *false*.

The interesting clauses are the last four for the *contains* algorithm. Initially the status is $IN(e)$ for $pc = I1$, and at the end of the algorithm it has definitely linearised: at I5 the cell *curr* has been fixed, so the test $curr.val = e$ determines the output, at I6 the output is already stored in *res*.

While the algorithm executes its main loop (I2,I3,I4) we exploit that *contains* can potentially linearise at any time by using a status of the form $INOUT(e, bv)$. The correct output value *bv* is simple to determine: it is just the value that *contains* would return if it would now run to completion without interruption (i.e., no other process executing steps). Note that this uniform characterisation should be applicable to every algorithm with potential LPs. For the *contains* algorithm this specialises to the value *bv* being true iff an unmarked cell is reachable from *curr* that contains *e*.

By using this status definition the algorithm "changes its mind" about the linearisation point and its outcome as often as necessary. Our proof obligation just requires that every change is justified by the current set representation. In particular, a process *p* marking the element that is searched by process *q* (the step from (a) to (b) in Fig. 3) will change *bv* in the status of process *q* executing *contains* to *false*. This is justified, since it is removed from the set representation too: executing an abstract *contains* with result *false* is possible after removal, we have an instance of simulation type 3 in Fig. 5. A process *q* adding a cell with *e* behind *curr* will change *bv* to *true*. Again this is justified, since the element is also added to the set. Adding an element that does not become reachable (e.g., stepping from (b) to (c) in Fig. 3) will keep $bv = false$.

By using an *INOUT* status the problem of finding the right LP is no longer a difficulty for the verification of the case study. The KIV proof of (LPO) just unfolds the definition of *exec* and checks whether the abstraction function changes correctly. All global reasoning and reasoning about the past has been moved into the generic theory.

It remains to be shown how the rest of the predicates and functions used in (LPO) are instantiated. Many of these instances are similar to the ones for verifying the pessimistic algorithm in [5]. In particular, the abstraction function just specifies that the abstract set consists of those values $r.val \neq \pm\infty$, for which a reference *r* is reachable from *head*. Also, the disjointness predicate *D* is solely used to ensure that *p* and *q* never share their newly allocated cell before adding it to the set representation (i.e., when both are at A4 or A5). The invariant consists of three parts.

$$\begin{aligned} INV(gs, ls) := (&\exists\, tail \in Ref \bullet HEADTAILINV(gs, tail) \wedge \\ &\forall\, r \in \mathrm{dom}(mem) \bullet NODEINV(gs, tail, r)) \wedge INVL(gs, ls) \end{aligned}$$

The first part, *HEADTAILINV* specifies the global invariant for the current data structure: a unique cell *tail* is always reachable from *head* such that $head.val = -\infty$, $tail.val = \infty$. Both *head* and *tail* are never marked.

The interesting part is the second. It gives a condition *NODEINV* for all allocated references r. This condition is necessary, since in contrast to the pessimistic version, the lazy algorithms for *contains* and *locate* may visit arbitrary old cells that have been marked and may also have been removed from the current set representation (as shown in Fig. 3 (b)).

```
$NODEINV((head, mem), tail, r) :=$
$(r.nxt \neq null \Rightarrow r.nxt \in \mathrm{dom}(mem) \wedge r.val < r.nxt.val) \wedge$
if $r.mrk$ then $r.val \neq -\infty \wedge reachable(r, tail, mem)$  /* class 1 */
else if $\exists\, r_0 \in \mathrm{dom}(mem) \bullet r_0.nxt = r$
then $reachable(head, r, mem) \wedge reachable(r, tail, mem)$  /* class 2 */
else if $r.val = -\infty$ then $head = r$  /* class 3 */
else $r.val \neq \pm\infty \wedge (r.nxt = null \vee reachable(r, tail, mem))$  /* class 4 */
```

NODEINV requires that even old cells are in strictly ascending order. It also divides the allocated cells into four classes. The first class contains all marked cells. These never contain $-\infty$, and allow to reach *tail* in a finite number of steps: the cells form a tree shape with pointers going upwards towards *tail* as the root. This ensures that *contains* never accesses dangling references. The second class are pointers that have a predecessor r_0. All these cells are part of the current set representation. Whenever *contains* or *locate* reach an *unmarked* cell by computing a successor, the cell is definitely a member of the set representation. Finally, there are cells which have no predecessor. One cell is *head* (third case). All remaining cells (fourth class) have just been allocated in *add* by some process, but have not yet been inserted into the set representation. These cells do not have a value $-\infty$ and either their *nxt* pointer is still *null* (A4) or has been set at A5, making *tail* reachable. Note that although we give the intuition, that *some* process has allocated such a cell, our *local* predicate avoids this *global* characterisation, which would have to quantify over existing processes. Figuring out a *simple*[3] classification of the allocated cells that works *locally* was the main difficulty specific to this case study.

The full invariant finally contains a local invariant *INVL* with assertions for intermediate states of the algorithms, typically by characterising the program counter values, where they hold. The main assertion for *contains*

$$\begin{aligned}&(ls.pc \in \{I2, I3, I5, I6\} \Rightarrow\\ &ls.curr \in \mathrm{dom}(mem) \wedge (reachable(head, ls.curr, mem) \vee ls.curr.mrk)\end{aligned}$$

ensures, that *curr* is always an allocated reference, and is either part of the set representation or an old marked cell. A similar property is used for the local variables *pred* and *curr* of *locate*. Note that *NODEINV* implies that this property is preserved when stepping from *curr* to *curr.nxt* in the algorithm.

[3] A generic, but more complex alternative is using an existentially quantified set of local cells, that must be updated where necessary. This is the preferred solution in separation logic, which hides the quantifier (and our D predicate) within the semantics of separating conjunction.

With these instances the verification of the proof obligation (LPO) in KIV is now only slightly more difficult than for the pessimistic case, and the additional complexity is solely due to the more complex invariant $NODEINV$. The technical encoding of Z schemata in KIV is the same as described in Section 7 of [5]. The proof obligation is given in KIV as three goals, one that proves invariance of $INV(gs, lsp)$, a second that proves $INV(gs, lsq)$ and $D(lsp, lsq)$, and finally one that checks the clauses about *exec*. Although an abstraction *function* is sufficient for the case study, the three goals in KIV generalise (LPO) using an abstraction *relation*, which shows that they are an instance of backward simulation. The proofs for the case study split immediately into 67 cases (one for each Z operation). Altogether the main proofs needed 276 interactions. Getting the details of the case study right took the second author about a week of work. All proofs and specifications (including the derivation from a global theory of possibilities that we could not describe here) are available on the Web [12].

6 Conclusion

The only other mechanised proof of the lazy set implementation of [9] we are aware of (except [2], which approximates a full linearizability proof by model checking executions of two fixed operations) is given in [3] using PVS. Like our approach (and [8]) it uses refinement (of IO automata) to prove linearisability.

Although the transition relation of the automaton in [3] corresponds to the disjunction of our operations in Z, the proof strategy is rather different. First, it considers the global automaton (with state CS) instead of a reduction to two processes. Second it defines an intermediate automaton specific to the case study, that splits the refinement into a forward and backward simulation, to cope with the problem that the LP of contains cannot be determined by forward simulation alone. Our approach solves the problem in the generic theory, and thus should be applicable to a wide class of algorithms. Third, the proof strategy uses a predicate *public* to distinguish locally available references from global ones, that are or have been in the set representation: a cell is not public, if it has just been allocated and is stored in the local variable $n2$ of *some* process p at A4 or A5. Such iteration over all processes is incompatible with our reduction to two processes. Finally, the proof idea follows [9]: when *contains* returns false, then there must have been a time in the past when the element was not in the set. Our theory completely avoids such reasoning about the past.

The same argument about the past is also used in [20]. In his PhD [18], Vafeiadis continues this work, giving proof obligations using separation logic and rely-guarantee reasoning. The approach has influenced our work, since Vafeiadis argues (Sect. 5.2.3), that several LPs are acceptable for read-only operations. Our mechanised proofs (that ensure that it is possible to change the *out* value when status is $INOUT$) can be viewed as a formal justification. Vafeiadis' work is not based on refinement, but adds ghost code executing abstract operations to the concrete algorithm at linearisation points. The approach is global, at the LP of *delete* the auxiliary code has to iterate over all threads running *contains*.

It has been implemented and can verify several standard algorithms automatically, though currently not the lazy set example (see [19]).

We did not have space to discuss the global theory underlying our local proof obligations: Any linearizable algorithm can be verified using backward simulation, when the abstract layer is defined using the possibilities from [11]. It is related to Theorems 13.3-5 of N. Lynch's book [14] (see also [13]) as well as to the embedding of linearizability into observational refinement given in [7]. We have mechanized the global theory in KIV and are not aware of any other approaches that have mechanically verified the soundness of their proof technique.

We conjecture that our local proof strategy is applicable to all algorithms which have potential linearisation points outside their thread and where the abstraction function does not change. The optimistic version of the set algorithm is another example of this class, as are algorithms where a potential LP exists that is determined in the future. The latter includes the "dequeue with an empty queue" case in Michael & Scott's queue [15].

Of course there remains future work. For example, two algorithms which would require further extensions include Herlihy & Wing's original queue (which requires a proof with the global theory) and the elimination stack [10], which uses a handshake to linearise a *push* and a *pop*-operation at the same time. The latter would need a reduction of the global theory to *three* processes (the two processes participating, and one representing all others), and we leave this for the future.

References

1. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
2. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 465–479. Springer, Heidelberg (2010)
3. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 475–488. Springer, Heidelberg (2006)
4. Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanizing a correctness proof for a lock-free concurrent stack. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 78–95. Springer, Heidelberg (2008)
5. Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanically verified proof obligations for linearizability. ACM Trans. Program. Lang. Syst. 33(1), 4 (2011)
6. Derrick, J., Wehrheim, H.: Non-atomic refinement in Z and CSP. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 24–44. Springer, Heidelberg (2005)
7. Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theoretical Computer Science 411(51-52), 4379–4398 (2010)
8. Groves, L., Colvin, R.: Trace-based derivation of a scalable lock-free stack algorithm. Formal Aspects of Computing (FAC) 21(1–2), 187–223 (2009)

9. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 305–309. Springer, Heidelberg (2006)
10. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: SPAA 2004, pp. 206–215. ACM Press, New York (2004)
11. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)
12. Web presentation of linearizability theory and the lazy set algorithm (2010), `http://www.informatik.uniaugsburg.de/swt/projects/possibilities.html`
13. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009)
14. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, San Francisco (1996)
15. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. 15th ACM Symp. on Principles of Distributed Computing, pp. 267–275 (1996)
16. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pp. 85–94 (2010)
17. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Automated Deduction—A Basis for Applications, Interactive Theorem Proving, vol. II, ch. 1, pp. 13–39. Kluwer, Dordrecht (1998)
18. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)
19. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
20. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP 2006, pp. 129–136. ACM, New York (2006)
21. Vafeiadis, V., Parkinson, M.: A marriage of rely/Guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)