Technical Report

UCAM-CL-TR-659
ISSN 1476-2986

Number 659

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A safety proof of a lazy concurrent list-based set implementation

Viktor Vafeiadis, Maurice Herlihy,
Tony Hoare, Marc Shapiro

January 2006

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

A safety proof of a lazy concurrent list-based set implementation

Viktor Vafeiadis
Computer Laboratory, University of Cambridge

Maurice Herlihy
Computer Science Dept., Brown University

Tony Hoare
Microsoft Research Cambridge

Marc Shapiro
INRIA Rocquencourt & LIP6

Abstract

We prove the safety of a practical concurrent list-based implementation due to Heller et al. It exposes an interface of an integer set with methods contains, add, and remove. The implementation uses a combination of fine-grain locking, optimistic and lazy synchronisation. Our proofs are hand-crafted. They use rely-guarantee reasoning and thereby illustrate its power and applicability, as well as some of its limitations. For each method, we identify the linearisation point, and establish its validity. Hence we show that the methods are safe, linearisable and implement a high-level specification. This report is a companion document to our PPoPP 2006 paper entitled "Proving correctness of highly-concurrent linearisable objects".

1 Introduction

This report is a companion to our extended abstract [4] published in PPoPP'06. There, we introduced a particular form of rely-guarantee reasoning and showed how it can be applied to prove the safety of a number of list-based concurrent algorithms. Our examples demonstrated a variety of common design patterns such as lock coupling, optimistic, and lazy synchronisation.

Here, we focus on the safety proof of one particular algorithm described in Section 7 of our extended abstract, which due to Heller et al [2]. The algorithm implements an abstract type of a set with operations *add*, *remove*, and *contains*. Our main proof approach is to identify the linearisation point within the code of each method and to inline the abstract operation at that position. Then, by means of *rely-guarantee reasoning*, we establish a linking invariant (also known as an abstraction map) between the concrete and the abstract state. Thus, we prove that each concrete method, should it return, has the same externally visible effect as its abstract counterpart. The proof (Section 7) is hand-written and demonstrates some of the benefits and the limitations of our approach.

Furthermore, we discuss linearisability in greater depth (Section 4) and we present all the axioms of our R-G formalism, including those omitted for brevity from the extended abstract (Section 5).

2 Predicates and relations

As we will make heavy use of predicates of a single state σ and predicates of two states, we first describe our notation. The former describe a set of system states, whereas the latter describe a set of actions (i.e. transitions) of the system. These two-state predicates relate the state σ immediately after the action to the state immediately before the action, which we shall denote as $\overleftarrow{\sigma}$. Similarly, we shall write $\overleftarrow{x}$ and x for the value of the variable x before and after the action respectively.

Given a single-state predicate p, we can straightforwardly define a corresponding two-state predicate, which requires p to hold in the new state σ, but places no constraint on the old state $\overleftarrow{\sigma}$. We denote this relation by simply overloading p. Similarly, we shall write $\overleftarrow{p}$ for the two-state predicate that is formed by requiring p to hold in the old state $\overleftarrow{\sigma}$ and which places no requirement on the new state σ.

$$\begin{aligned} p(\overleftarrow{\sigma}, \sigma) &\stackrel{\text{def}}{=} p(\sigma) \\ \overleftarrow{p}(\overleftarrow{\sigma}, \sigma) &\stackrel{\text{def}}{=} p(\overleftarrow{\sigma}) \end{aligned}$$

We shall say that a single-state predicate p is preserved by a two-state predicate R, if and only if, $(\overleftarrow{p} \wedge R) \Rightarrow p$, namely iff for all $\overleftarrow{\sigma}$ and σ, $p(\overleftarrow{\sigma}) \wedge R(\overleftarrow{\sigma}, \sigma) \;\Rightarrow\; p(\sigma)$.

We shall use relational notation to abbreviate operations on predicates of two states. Relational composition of predicates describes exactly the intended behaviour of the sequential composition of sequential programs.

$$(P; Q)(\overleftarrow{\sigma}, \sigma) \stackrel{\text{def}}{=} \exists \tau.\ P(\overleftarrow{\sigma}, \tau) \wedge Q(\tau, \sigma)$$

The program that makes no change to the state is described exactly by

$$\mathrm{ID}(\overleftarrow{\sigma}, \sigma) \stackrel{\text{def}}{=} (\overleftarrow{\sigma} = \sigma).$$

The familiar notation R^* (reflexive and transitive closure) describes any finite number of iterations of the program described by R. It is defined

$$R^* \stackrel{\text{def}}{=} \mathrm{ID} \vee R \vee (R; R) \vee (R; R; R) \vee \cdots$$

3 Rely-guarantee specifications

In *rely-guarantee* (R-G) reasoning, each thread is assigned a *rely* condition that characterises the interference that thread can tolerate from the other threads. In return, the thread is assigned a *guarantee* condition that characterises how that thread can interfere with the others. Proving the safety of a program requires proving that (1) if each thread's rely condition is satisfied, then that thread satisfies its guarantee condition, and (2) each thread's guarantee condition implies the others' rely conditions.

The specification of a fine-grain concurrent program requires four predicates: (p, R, G, q).

- The predicates p and q are the *pre-condition* and *post-condition*. They describe the behaviour of the thread as a whole, from the time it starts to the time it terminates (if it does). The pre-condition p, a single-state predicate, describes an assumption about the initial state that must hold for the program to make sense. The post-condition q is a two-state predicate relating the initial state (just before the program starts execution) to the final state (immediately after the program terminates). The post-condition describes the overall *action* of the program, and is meant to be atomic with respect to the rest of the system.

- R and G summarise the properties of the individual atomic actions invoked by the environment (in the case of R) and the thread itself (in the case of G). They are two-state predicates, relating the state $\overleftarrow{\sigma}$ before each individual atomic action to σ, the one immediately after that action. The *rely condition* R bounds the interference the thread can tolerate from the environment, whereas the *guarantee condition* G bounds the interference that it can impose on the other threads.

We require that the pre-condition is always preserved by the rely condition, but we do not impose a similar requirement for post-condition.

In the rely condition, we often want to specify that there are certain variables the environment does not update, or that if some condition is satisfied, the environment actions preserve it [1]. We introduce the following notation for these specifications.

$$\begin{aligned}
\mathrm{ID}(x) &\stackrel{\text{def}}{=} (\overleftarrow{x} = x) \\
\mathrm{ID}(P) &\stackrel{\text{def}}{=} (\overleftarrow{P} = P) \\
\mathrm{Preserve}(P) &\stackrel{\text{def}}{=} (\overleftarrow{P} \Rightarrow P)
\end{aligned}$$

We extend these notations to multiple variables/conditions. For convenience in the post-condition and guarantee condition, we define $\mathsf{Mod}(X)$ to mean that only variables in the set X are modified by the action.

4 Linearisability

A program is said to be linearisable [3], if and only if, all its externally visible effects appear to take place atomically at some instant between the program's invocation and its termination. This instant when all the externally visible effects of the program appear to take place is known as the linearisation point.

This definition only makes sense in a context where the terms "externally visible" and "atomic" are meaningful. We take this to be some simple module, where each module owns some private memory, which is disjoint and unaliased with the rest of the memory. (This disjointness and lack of aliasing could be enforced by an ownership type system or, to some extent, with private objects.) An *externally visible effect* is an update to the global memory and the result of a call to a public method of the module. Similarly, *atomic* will be with respect to all public methods on the module and with any code outside the module. When a module implements a data structure and provides update and access methods, it is often useful to define an abstract version of the data structure and assume

that it is globally visible. Even though it may not directly be visible, all its data would be available through the module's access methods.

At the very least we require a sequentially consistent memory model. Strictly speaking, we also require the order of method calls and returns to be preserved: this amounts to saying that method calls and returns happen through memory. It is, however, safe to have the order of method calls and returns not globally consistent, because the only way to detect a sequential inconsistency would be through shared memory, by this is already required to be sequentially consistent. In our proof below, non-sequential consistency, is not really an issue, because Java guarantees sequential consistency, as long as we declare all fields with race conditions as `volatile`.

For us the linearisation point is a 'mythical point' whose existence we have proved. There need not be a unique linearisation point; if there are several valid linearisation points, we allowed to chose any one of them. In our R-G specifications, the post-condition q will stand for the total (abstract) action of the thread that to an external observer appears to happen atomically at the linearisation point.

If in our R-G specifications $(p,\ R,\ G,\ q)$, we restrict q in a certain way, then we get the more standard presentation of R-G where both the pre-condition and the post-condition are single-state predicates and are preserved by the rely condition R. By treating, however, q as a two-state predicate, which is not necessarily preserved by R, we gain much in expressiveness which is crucial for the specifications we are aiming at. The special case is achieved by taking q to be $\overleftarrow{p} \wedge q'$, where p is the precondition and q' is a predicate restricting the new state which is also preserved by the rely condition R.

It should be noted that whereas the general form is appears important for specifications, the special form described above is more useful for doing proofs, as the proof rules are simplified.

5 Axioms

We shall write $C \models (p,\ R,\ G,\ q)$ for the judgement saying that the program C meets the specification $(p,\ R,\ G,\ q)$. We shall give the associated proof rules below.

As it is customary, we will write $\{p\}\ C\ \{q\}$ for the judgement stating that if the sequential program C is executed in an initial state satisfying the precondition p (and if it terminates), then the final state shall be related to the initial state by q. This partial correctness correctness judgement is well known from Hoare logic; the precise definition of its rules lies outside the scope of this paper.

6 Atomic actions

Atomic actions are denoted by enclosing a program in diamond brackets $\langle C \rangle$. As a sequential program, C can be modelled as a predicate pair $(p,\ q)$. As an atomic action, it becomes:

$$\frac{\{p\}\ C\ \{q\}}{\langle C \rangle \models (p,\ \mathrm{Preserve}(p),\ q \vee \mathrm{ID},\ q)} \qquad \text{(ATOMIC)}$$

The pre-condition and post-condition are unchanged. The guarantee condition is just the post-condition, and the rely condition must at least preserve the precondition. The implementation of atomicity must ensure that this interference cannot take place within the diamond brackets, so the proof of correctness of the atomic region are unaffected by interference. We will show later how the programmer is responsible for ensuring that interference is harmless in-between the atomic actions.

Conditional critical regions can be handled in a similar way:

$$\frac{\{b \wedge p\}\ C\ \{q\}}{\langle b \rightarrow C \rangle\ \models\ (p,\ \text{Preserve}(p),\ q \vee \text{ID},\ q)} \qquad \text{(CRITICAL)}$$

6.1 Sequential composition

For simplicity, we define sequential composition for programs with identical rely and guarantee conditions. (These conditions can always be weakened or strengthened as discussed in Section 6.4.)

$$\frac{\begin{array}{ll} C_1\ \models\ (p_1,\ R,\ G,\ q_1) & \\ C_2\ \models\ (p_2,\ R,\ G,\ q_2) & q_1 \Rightarrow p_2 \end{array}}{C_1; C_2\ \models\ (p_1,\ R,\ G,\ (q_1; R^*; q_2))} \qquad \text{(SEQ)}$$

A sequential composition has the same pre-condition p_1 as its first operand, and the same rely and guarantee conditions as both its operands. For the sequential composition to execute properly, the pre-condition of the second operand must be satisfied when the first operand terminates; hence the proof requirement. Since the entire program will tolerate the same interference R as both its operands except at the very transition between the two operands, the total action of the program will be given by the composition of the actions of its components accounting for environment interference R^* that may occur between them.

6.2 Parallel composition

When threads run concurrently, each thread must ensure that its atomic actions do not interfere with the other threads except as expected. It is therefore essential to prove that the guarantee condition of each thread implies the rely condition of the others. The total action of the program is given by the composition of the actions of the two threads in either order, allowing for environment interference $(R_1 \wedge R_2)^*$ in between. All pre-conditions and rely conditions must hold, but concurrent combination can guarantee only the disjunction of the separate guarantee conditions.

$$\frac{\begin{array}{ll} C_1\ \models\ (p_1,\ R_1,\ G_1,\ q_1) & G_1 \Rightarrow R_2 \\ C_2\ \models\ (p_2,\ R_2,\ G_2,\ q_2) & G_2 \Rightarrow R_1 \end{array}}{C_1 \parallel C_2\ \models\ (p_1 \wedge p_2,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ q)} \qquad \text{(PAR)}$$

where

$$q\ =\ (q_1; (R_1\ \wedge\ R_2)^*; q_2) \vee (q_2; (R_1\ \wedge\ R_2)^*; q_1)$$

In plain English, and generalising to any number of threads, it means that proving the safety of a parallel program reduces to: (*i*) a sequential proof of the post-condition and guarantee condition of each individual thread, assuming its rely condition is true, combined with (*ii*) a pairwise proof that every other thread's guarantee condition implies this thread's rely condition.

6.3 Conditionals and loops

The axiom for conditionals allows for environment interference to occur between the test and the branches of the conditional. This interference need not preserve the whole test b, but perhaps only a part of it b_1. Similarly in the **else** branch for $\neg b$ and b_2.

$$\frac{\begin{array}{ll} C_1 \;\models\; (p \wedge b_1,\, R,\, G,\, q) & \overleftarrow{b} \wedge R^* \;\Rightarrow\; b_1 \\ C_2 \;\models\; (p \wedge b_2,\, R,\, G,\, q) & \overleftarrow{\neg b} \wedge R^* \;\Rightarrow\; b_2 \end{array}}{\textbf{if } \langle b \rangle \textbf{ then } C_1 \textbf{ else } C_2 \;\models\; (p,\, R,\, G,\, q)} \quad \text{(IF)}$$

While loops are treated similarly; interference can occur between the test and the loop body, and between the loop body and the test. Thus the loop invariant J must be preserved by interference, and only the part of the test condition that preserved by interference (i.e., b_1) may be used as a pre-condition for the loop body. There is no need to consider interference after the test failure, as this will be considered in the appropriate composition rule.

$$\frac{C \;\models\; (b_1 \wedge J,\, R,\, G,\, J) \qquad R \Rightarrow \text{Preserve}(J) \qquad (\overleftarrow{b} \wedge R^*) \Rightarrow b_1}{\textbf{while } (\langle b \rangle)\ \{C\} \;\models\; (J,\, R,\, G,\, \neg b \wedge J)} \quad \text{(WHILE)}$$

6.4 Refinement

Programs and specifications can be compared with each other by the standard refinement ordering. A stronger specification is possibly more desirable but more difficult to meet. When developing a program from its specification, it is always valid to replace the specification by a stronger one. A specification is weakened by weakening its post-condition or its guarantee condition. Conversely, it is strengthened by weakening its assumptions.

$$\frac{\begin{array}{ll} p' \Rightarrow p & R' \Rightarrow R \\ G \Rightarrow G' & q \Rightarrow q' \\ C \;\models\; (p,\, R,\, G,\, q) & \end{array}}{C \;\models\; (p',\, R',\, G',\, q')} \quad \text{(REFINE)}$$

6.5 Lifting of linearisation point

Since the post-condition q describes the total action of the linearisation point, we require a rule that, when given a proof that a linearisation point exists, 'lifts' the action of that linearisation point to the linearisable action q of the whole program.

The following rule does this. Assuming that there exists a linearisation point (indicated by the unique moment where the condition b changes value and becomes true), and at

that point the new abstract state A is a (computable) function f of the old abstract state $\overleftarrow{A}$, then the action $A = f(\overleftarrow{A})$ can become the post-condition, namely the total action of the program.

$$\frac{\begin{array}{c} C \;\models\; (p \wedge \neg b,\, R,\, G,\, b) \\ G \Rightarrow ((\neg \overleftarrow{b} \wedge b \wedge A = f(\overleftarrow{A})) \vee (\mathrm{Preserve}(b) \wedge \mathrm{ID}(A))) \\ R \Rightarrow ((\neg \overleftarrow{b} \wedge b \wedge A = f(\overleftarrow{A})) \vee (\mathrm{Preserve}(b) \wedge \mathrm{ID}(A))) \end{array}}{C \;\models\; (p,\, R,\, G,\, A = f(\overleftarrow{A}))} \qquad \text{(LIFT)}$$

Note that the rule allows either the current thread or some other thread from the environment to set b to true and, thus, provide a witness of the linearisation point. For it can be the case (e.g. *contains* method in Section 7) that the linearisation point is between actions of other threads. The premises of this rule are trivially fulfilled when we inline the (atomic) abstract operation at a unique place within the concrete code.[1]

7 Proof of the lazy concurrenct list algorithm

We present a hand-crafted proof of the algorithm, due to Heller et al [2], a sketch of which appeared in our extended abstract [4]. It consists of a linearisable sorted linked list implementation of a set abstract data type. It contains three public methods *contains*, *add*, and *remove* with the (abstract) specifications given below; internally, there exists a private *locate* method, which is used by *add* and *remove*. For simplicity, we shall assume that only one such list exists.

$$\begin{array}{lll} AbsContains(e) : & < & AbsResult := e \in Abs \quad > \\ AbsAdd(e) : & < & AbsResult := e \notin Abs \;; \\ & & Abs := Abs \cup \{e\} \quad > \\ AbsRemove(e) : & < & AbsResult := e \in Abs \;; \\ & & Abs := Abs \setminus \{e\} \quad > \end{array}$$

List nodes have the fields .val of type integer, .next of type pointer, and .marked of type boolean (a single bit used to mark deleted nodes). When a node is first allocated, only the allocating thread can access that its fields; we say that the node is *private*. Once the reference is placed in a shared location, it becomes *public*. We write $\mathsf{Node}(\mathrm{n})$ for the assertion that n is a valid public node, and $\mathsf{PrivateNode}(\mathrm{n})$ for the assertion that n is owned by the current thread. We write $\forall_{\mathsf{Node}}\ \mathrm{n}.\ P$ to quantify over all public nodes. Furthermore, we write $\mathrm{n} \rightarrow \mathrm{m}$ for $\mathsf{Node}(\mathrm{n.next}) \wedge \mathrm{n.next} = \mathrm{m}$, and $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$.

[1] Some care is required is required when applying the lifting rule, as it cannot be combined with other seemingly harmless rules such as:

$$\frac{C \;\models\; (p,\, R,\, G,\, q_1) \qquad C \;\models\; (p,\, R,\, G,\, q_2)}{C \;\models\; (p,\, R,\, G,\, q_1 \wedge q_2)}$$

The problem is that q_1 and q_2 might be linearised with respect to different abstract states (i.e. different 'types').

We shall now give the definitions of the list invariant and the common rely and guarantee conditions; then we will prove the methods obey their specifications.

7.1 Definitions

The list invariant (representation invariant and abstraction map):

$$\begin{aligned} \mathit{ListInv} \stackrel{\text{def}}{=} & \\ & \mathsf{Node}(\text{Head}) \wedge \text{Head.val} = -\infty \wedge \neg\text{Head.marked} \\ \wedge\ & \mathsf{Node}(\text{Tail}) \wedge \text{Tail.val} = +\infty \wedge \neg\text{Tail.marked} \\ \wedge\ & \forall_{\mathsf{Node}}\ \text{n.}\ \text{n.val} < +\infty \Rightarrow \mathsf{Node}(\text{n.next}) \\ \wedge\ & \forall_{\mathsf{Node}}\ \text{n m.}\ \text{n} \rightarrow \text{m} \Rightarrow \text{n.val} < \text{m.val} \\ \wedge\ & \forall_{\mathsf{Node}}\ \text{n.}\ \text{Head} \rightarrow^* \text{n} \vee \text{n.marked} \\ \wedge\ & \mathit{Abs} = \{\text{n.val} \mid \mathsf{Node}(\text{n}) \wedge \neg\text{n.marked} \wedge \text{n.val} \neq \pm\infty\} \end{aligned}$$

The additional predicate required for the linearisability of *contains*:

$$N \stackrel{\text{def}}{=} \forall_{\mathsf{Node}}\ \text{n.}\ (\neg\overleftarrow{\text{n.marked}} \wedge \text{n.marked}) \Rightarrow \text{n.val} \notin \mathit{Abs}$$

$$\begin{aligned} \mathit{locate.Post} \stackrel{\text{def}}{=}\ & \mathit{ListInv} \wedge \text{Head} \rightarrow^* \text{pred} \rightarrow \text{curr} \\ & \wedge\ \text{pred.val} < e \leq \text{curr.val} \\ & \wedge\ \text{pred.owner} = \text{curr.owner} = \mathsf{self} \\ & \wedge\ \neg\text{pred.marked} \wedge \neg\text{curr.marked} \end{aligned}$$

The rely condition is:

$$\begin{aligned} R \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\ \text{n.}\ \text{Preserve}(\mathit{ListInv}) \wedge \text{n.}\mathit{LockRely} \\ & \wedge\ \overleftarrow{\text{n.owner}} = \mathsf{self} \Rightarrow \text{ID}(\text{n.next}, \text{n.marked}) \\ & \wedge\ \overleftarrow{\text{n.owner}} = \mathsf{self} \Rightarrow \text{Preserve}(\text{Head} \rightarrow^* \text{n}) \\ & \wedge\ \text{Preserve}(\text{n.marked}) \wedge \text{ID}(\text{n.val}) \wedge N \end{aligned}$$

The guarantee condition is:

$$\begin{aligned} G \stackrel{\text{def}}{=}\ & \forall_{\mathsf{Node}}\ \text{n.}\ \text{Preserve}(\mathit{ListInv}) \wedge \text{n.}\mathit{LockGuar} \\ & \wedge\ \overleftarrow{\text{n.owner}} \neq \mathsf{self} \Rightarrow \text{ID}(\text{n.next}, \text{n.marked}) \\ & \wedge\ \overleftarrow{\text{n.owner}} \neq \mathsf{self} \Rightarrow \text{Preserve}(\text{Head} \rightarrow^* \text{n}) \\ & \wedge\ \text{Preserve}(\text{n.marked}) \wedge \text{ID}(\text{n.val}) \wedge N \end{aligned}$$

7.2 Proof of $G \Rightarrow R$

First, we shall prove that the guarantee condition of each thread implies the rely conditions of all other threads; namely, for all x and y such that $x \neq y$, we shall show that $G_{\mathsf{self}\ :=\ x} \Rightarrow R_{\mathsf{self}\ :=\ y}$.

Proof. We do a case split on the definition of $R_{\mathsf{self}\ :=\ y}$.

1. $G_{\mathsf{self}\ :=\ x} \Rightarrow \text{Preserve}(\mathit{ListInv})$

2. $G_{\mathsf{self}\ :=\ x} \Rightarrow \text{n}.\mathit{LockRely}$, because $\text{n}.\mathit{LockGuar} \Rightarrow \text{n}.\mathit{LockRely}$
3. $G_{\mathsf{self}\ :=\ x} \Rightarrow (\overleftarrow{\text{n.owner}} = y \Rightarrow \text{ID}(\text{n.next}, \text{n.marked}))$,
   (a) $G_{\mathsf{self}\ :=\ x} \Rightarrow \overleftarrow{\text{n.owner}} \neq x \Rightarrow \text{ID}(\text{n.next}, \text{n.marked})$
   (b) $x \neq y$ (assumption)
4. $G_{\mathsf{self}\ :=\ x} \Rightarrow (\overleftarrow{\text{n.owner}} = \mathsf{self} \Rightarrow \text{Preserve}(\text{Head} \rightarrow^* \text{n}))$
   (a) $G_{\mathsf{self}\ :=\ x} \Rightarrow \overleftarrow{\text{n.owner}} \neq x \Rightarrow \text{Preserve}(\text{Head} \rightarrow^* \text{n})$
   (b) $x \neq y$ (assumption)
5. $G_{\mathsf{self}\ :=\ x} \Rightarrow (\text{Preserve}(\text{n.marked}) \wedge \text{ID}(\text{n.val}) \wedge N)$

□

7.3 Proof of locate(e) returns (pred,curr)

- Precondition: $\mathit{ListInv} \wedge -\infty < e < +\infty$
- Postcondition: $\mathit{ListInv} \wedge \text{Head} \rightarrow^* \text{pred} \rightarrow \text{curr} \wedge \text{pred.val} < e \leq \text{curr.val}$
  $\wedge$ pred.owner = curr.owner = self $\wedge$ $\neg$pred.marked $\wedge$ $\neg$curr.marked
- Rely: R
- Guarantee: G

The algorithm (desugared, and annotated with atomicity assumptions):[2]

```
locate(e) :
 while (true) {
   ⟨pred := Head⟩ ;
   ⟨curr := pred.next⟩ ;
   while (⟨curr.val < e⟩) {
     ⟨pred := curr⟩ ;
     ⟨curr := curr.next⟩
   } ;
   pred.lock() ;
   curr.lock() ;
   if ⟨¬pred.marked⟩
       and ⟨¬curr.marked⟩
       and ⟨pred.next = curr⟩ then
         return pred, curr
   else
     pred.unlock() ;
     curr.unlock()
 }
```

[2]Normally, the short-circuit **and** should have been expanded into nested **if** statements, but for brevity we chose not to do so.

Proof. The outer loop's invariant is just the pre-condition

$$I \quad \equiv \quad (\mathit{ListInv} \wedge -\infty < e < +\infty)$$

which is preserved by interference.
$\{\ I\ \}$
pred := Head
$\{\ I \wedge \text{pred.val} < e \wedge \mathsf{Node}(\text{pred})\ \}$
from the *ListInv*
$\{\ I \wedge \text{pred.val} < e \wedge \mathsf{Node}(\text{pred}, \text{pred.next})\ \}$
curr := pred.next
$\{\ I \wedge \text{pred.val} < e \wedge \mathsf{Node}(\text{pred}, \text{curr})\ \}$

The loop invariant is:
$\{\ I \wedge \text{pred.val} < e \wedge \mathsf{Node}(\text{pred}, \text{curr})\ \}$

while (curr.val $< e$) {
$\{\ I \wedge \text{curr.val} < e \wedge \mathsf{Node}(\text{pred}, \text{curr})\ \}$
pred := curr
$\{\ I \wedge \text{pred.val} < e \wedge \text{curr.val} < e \wedge \mathsf{Node}(\text{pred}, \text{curr})\ \}$
from the *ListInv*
$\{\ I \wedge \text{pred.val} < e \wedge \mathsf{Node}(\text{pred}, \text{curr.next})\ \}$
curr := curr.next
$\{\ I \wedge \text{pred.val} < e \wedge \mathsf{Node}(\text{pred}, \text{curr})\ \}$
}

From here onwards, we will implicitly always assume $\mathsf{Node}(\text{pred}, \text{curr})$.
$\{\ I \wedge \text{pred.val} < e \leq \text{curr.val}\ \}$
pred.lock()
$\{\ I \wedge \text{pred.val} < e \leq \text{curr.val} \wedge \text{pred.owner} = \mathsf{self}\ \}$
curr.lock()
$\{\ I \wedge \text{pred.val} < e \leq \text{curr.val} \wedge \text{pred.owner} = \text{curr.owner} = \mathsf{self}\ \}$

if ¬pred.marked
$\{\ I \wedge \text{pred.val} < e \leq \text{curr.val} \wedge \text{pred.owner} = \text{curr.owner} = \mathsf{self}$
$\wedge\ \neg\text{pred.marked}\ \}$
and ¬curr.marked
$\{\ I \wedge \text{pred.val} < e \leq \text{curr.val} \wedge \text{pred.owner} = \text{curr.owner} = \mathsf{self}$
$\wedge\ \neg\text{pred.marked} \wedge \neg\text{curr.marked}\ \}$
and pred.next = curr **then**
$\{\ I \wedge \text{pred.val} < e \leq \text{curr.val} \wedge \text{pred.owner} = \text{curr.owner} = \mathsf{self}$
$\wedge\ \neg\text{pred.marked} \wedge \neg\text{curr.marked} \wedge \text{pred} \rightarrow \text{curr}\ \}$
$\Rightarrow$
$\{\ \mathit{ListInv} \wedge \text{Head} \rightarrow^* \text{pred} \rightarrow \text{curr} \wedge \text{pred.val} < e \leq \text{curr.val}$
$\wedge\ \text{pred.owner} = \text{curr.owner} = \mathsf{self} \wedge \neg\text{pred.marked} \wedge \neg\text{curr.marked}\ \}$
which is the required post-condition.
else

weakening
$\{\ I \wedge \text{pred.owner} = \text{curr.owner} = \mathsf{self}\ \}$
pred.unlock()
$\{\ I \wedge \text{curr.owner} = \mathsf{self}\ \}$
curr.unlock()
$\{\ I\ \}$
which is the loop invariant.

All conditions are preserved by interference, as
(i) $R \Rightarrow \text{Preserve}(\text{n.marked}, \text{n.owner} = \mathsf{self})$
(ii) $R \Rightarrow \text{ID}(\text{n.val})$
(iii) $R \wedge \overleftarrow{\text{n.owner}} = \mathsf{self} \Rightarrow \text{ID}(\text{n.marked}, \text{n.next})$
(iv) $R \wedge \overleftarrow{\text{n.owner}} = \mathsf{self} \Rightarrow \text{Preserve}(\text{Head} \rightarrow^* \text{n})$
(v) $R \Rightarrow \text{Preserve}(I)$ □

7.4 Add

The method call $Result := add(e)$ has the specification:

- Precondition: $ListInv \wedge -\infty < e < +\infty$
- Postcondition: $ListInv \wedge Result = AbsResult$
- Rely: R
- Guarantee: G

The algorithm with atomicity assumptions, and annotated with the abstract operations at the linearisation point.

$add(e)$:
n1, n3 := $locate(e)$;
if $\langle$n3.val $\neq e\rangle$ **then**
$\langle$n2 := **new** $Node(e)\rangle$;
$\langle$n2.next := n3$\rangle$;
$\langle$makePublic(n2) ;
n1.next := n2 ;
$AbsResult := e \notin Abs$;
$Abs := Abs \cup \{e\}\rangle$;
$Result :=$ true
else
$\langle Result :=$ false ;
$AbsResult := e \notin Abs$;
$Abs := Abs \cup \{e\}\rangle$;
endif ;
n1.unlock() ;
n3.unlock() ;
return $Result$

Proof. Let $I \stackrel{\text{def}}{=} \mathit{ListInv} \wedge -\infty < e < +\infty$.

$\{\ I\ \}$
n1, n3 := $\mathit{locate}(e)$
$\{\ I \wedge \text{Head} \rightarrow^* \text{n1} \rightarrow \text{n3} \wedge \text{n1.val} < e \leq \text{n3.val}$
$\wedge\ \text{n1.owner} = \text{n3.owner} = \mathsf{self} \wedge \neg\text{n1.marked} \wedge \neg\text{n3.marked}\ \}$
if n3.val $\neq e$ **then**
$\{\ I \wedge \text{Head} \rightarrow^* \text{n1} \rightarrow \text{n3} \wedge \text{n1.val} < e < \text{n3.val}$
$\wedge\ \text{n1.owner} = \text{n3.owner} = \mathsf{self} \wedge \neg\text{n1.marked} \wedge \neg\text{n3.marked}\ \}$
n2 := **new** $\mathit{Node}(e)$
$\{\ \ldots \wedge \mathsf{PrivateNode}(\text{n2}) \wedge \text{n2.val} = e \wedge \neg\text{n2.marked}\ \}$
n2.next := n3
$\{\ \ldots \wedge \mathsf{PrivateNode}(\text{n2}) \wedge \text{n2.val} = e \wedge \text{n2.next} = \text{n3} \wedge \neg\text{n2.marked}\ \}$
$\Longrightarrow$
$\{\ I \wedge e \notin \mathit{Abs} \wedge \text{Head} \rightarrow^* \text{n1} \rightarrow \text{n3} \wedge \text{n1.val} < \text{n2.val} < \text{n3.val}$
$\wedge\ \text{n1.owner} = \text{n3.owner} = \mathsf{self} \wedge \neg\text{n1.marked} \wedge \neg\text{n3.marked}$
$\wedge\ \mathsf{PrivateNode}(\text{n2}) \wedge \text{n2.val} = e \wedge \text{n2.next} = \text{n3} \wedge \neg\text{n2.marked}\ \}$
$\mathsf{makePublic}(\text{n2})$
n1.next := n2
$\mathit{AbsResult} := e \notin \mathit{Abs}$
$\mathit{Abs} := \mathit{Abs} \cup \{e\}$
$\{\ I \wedge \mathit{AbsResult} \wedge \text{n1.owner} = \text{n3.owner} = \mathsf{self}\ \}$
$\mathit{Result} := \mathsf{true}$
$\{\ I \wedge \mathit{Result} = \mathit{AbsResult} \wedge \text{n1.owner} = \text{n3.owner} = \mathsf{self}\ \}$
else
$\{\ I \wedge \text{Head} \rightarrow^* \text{n1} \rightarrow \text{n3} \wedge e = \text{n3.val}$
$\wedge\ \text{n1.owner} = \text{n3.owner} = \mathsf{self} \wedge \neg\text{n1.marked} \wedge \neg\text{n3.marked}\ \}$
$\Longrightarrow$
$\{\ I \wedge e \in \mathit{Abs} \wedge \text{n1.owner} = \text{n3.owner} = \mathsf{self}\ \}$
$\mathit{AbsResult} := e \notin \mathit{Abs}$
$\mathit{Abs} := \mathit{Abs} \cup \{e\}$
$\mathit{Result} := \mathsf{false}$
$\{\ I \wedge \mathit{Result} = \mathit{AbsResult} \wedge \text{n1.owner} = \text{n3.owner} = \mathsf{self}\ \}$

$\{\ I \wedge \mathit{Result} = \mathit{AbsResult} \wedge \text{n1.owner} = \text{n3.owner} = \mathsf{self}\ \}$
n1.unlock()
$\{\ I \wedge \mathit{Result} = \mathit{AbsResult} \wedge \text{n3.owner} = \mathsf{self}\ \}$
n3.unlock()
$\{\ I \wedge \mathit{Result} = \mathit{AbsResult}\ \}$
return *Result*
$\{\ I \wedge \mathit{Result} = \mathit{AbsResult}\ \}$

All intermediate conditions are preserved, because:

- n1, n2, n3, e, *Result*, *AbsResult* are local variables
- $R \Rightarrow \text{Preserve}(I)$
- $R \Rightarrow \text{ID}(\text{n1.owner} = \mathsf{self}, \text{n3.owner} = \mathsf{self})$

- $R \wedge \text{n1}.\overleftarrow{\text{owner}} = \mathsf{self} \Rightarrow \text{ID}(\text{n1.next}, \text{n1.marked})$
- $R \wedge \text{n3}.\overleftarrow{\text{owner}} = \mathsf{self} \Rightarrow \text{ID}(\text{n3.marked})$
- $R \wedge \neg\mathsf{Initialised}(\text{n2}) \Rightarrow \text{ID}(\text{n2.val}, \text{n2.next}, \text{n2.marked})$

The guarantee condition is valid, because:

- Preserve($ListInv$) – $ListInv$ holds at all steps
- ID(n.val, n.marked) $\wedge$ n.$LockGuar$ – by code inspection, no assignments to these for public nodes
- n.owner $\neq$ self $\Rightarrow$ ID(n.next) – the only updates to .next are the assignment to n2.next and n1.next; but at those points n1 is locked and n2 is owned.

□

7.5 Remove

- Precondition: $ListInv \wedge -\infty < e < +\infty$
- Postcondition: $ListInv \wedge Result = AbsResult$
- Rely: R
- Guarantee: G

The annotated algorithm:

$remove(e)$:
  n1, n2 := $locate(e)$;
  if $\langle$n2.val $= e\rangle$ **then**
    $\langle$n2.marked := true ;
     $AbsResult := e \in Abs$;
     $Abs := Abs \setminus \{e\}\rangle$
    $\langle$n3 := n2.next$\rangle$;
    $\langle$n1.next := n3$\rangle$;
    $\langle Result :=$ true$\rangle$
  else
    $\langle Result :=$ false ;
     $AbsResult := e \in Abs$;
     $Abs := Abs \setminus \{e\}\rangle$
  endif ;
  n1.unlock() ;
  n2.unlock() ;
  return $Result$

Proof. First we do a sequential proof; we consider R-G conditions later.

$\{\ \mathit{ListInv} \wedge -\infty < e < +\infty\ \}$
n1, n2 := $\mathit{locate}(e)$
$\{\ \mathit{ListInv} \wedge \mathrm{Head} \rightarrow^* \mathrm{n1} \rightarrow \mathrm{n2} \wedge \mathrm{n1.val} < e \leq \mathrm{n2.val} \wedge e < +\infty$
$\wedge\ \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self} \wedge \neg \mathrm{n1.marked} \wedge \neg \mathrm{n2.marked}\ \}$

if n2.val = e **then**
$\{\ \mathit{ListInv} \wedge \mathrm{Head} \rightarrow^* \mathrm{n1} \rightarrow \mathrm{n2} \wedge \mathrm{n1.val} < e = \mathrm{n2.val} < +\infty$
$\wedge\ \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self} \wedge \neg \mathrm{n1.marked} \wedge \neg \mathrm{n2.marked}\ \}$
$\mathit{n2.marked} := \mathit{true}$
$\mathit{AbsResult} := e \in \mathit{Abs}$
$\mathit{Abs} := \mathit{Abs} \setminus \{e\}$
$\{\ \mathit{ListInv} \wedge \mathrm{Head} \rightarrow^* \mathrm{n1} \rightarrow \mathrm{n2} \wedge \mathrm{n1.val} < e = \mathrm{n2.val} < +\infty$
$\wedge\ \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self} \wedge \neg \mathrm{n1.marked} \wedge \mathrm{n2.marked}\ \}$
by ListInv
$\{\ \mathit{ListInv} \wedge \mathrm{Head} \rightarrow^* \mathrm{n1} \rightarrow \mathrm{n2} \wedge \mathrm{n2} \rightarrow \mathrm{n2.next}$
$\wedge\ \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self} \wedge \neg \mathrm{n1.marked} \wedge \mathrm{n2.marked}\ \}$
n3 := n2.next
$\{\ \mathit{ListInv} \wedge \mathrm{Head} \rightarrow^* \mathrm{n1} \rightarrow \mathrm{n2} \rightarrow \mathrm{n3}$
$\wedge\ \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self} \wedge \neg \mathrm{n1.marked} \wedge \mathrm{n2.marked}\ \}$
n1.next := n3
$\{\ \mathit{ListInv} \wedge \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self}\ \}$
Result := true
$\{\ \mathit{ListInv} \wedge \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self}\ \}$

else
$\{\ \mathit{ListInv} \wedge \mathrm{Head} \rightarrow^* \mathrm{n1} \rightarrow \mathrm{n2} \wedge \mathrm{n1.val} < e < \mathrm{n2.val}$
$\wedge\ \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self} \wedge e < +\infty\ \}$
weakening...
$\{\ \mathit{ListInv} \wedge \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self}\ \}$
Result := false
$\{\ \mathit{ListInv} \wedge \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self}\ \}$
endif

$\{\ \mathit{ListInv} \wedge \mathrm{n1.owner} = \mathrm{n2.owner} = \mathsf{self}\ \}$
n1.unlock()
$\{\ \mathit{ListInv} \wedge \mathrm{n2.owner} = \mathsf{self}\ \}$
n2.unlock()
$\{\ \mathit{ListInv}\ \}$
return *Result*
$\{\ \mathit{ListInv}\ \}$

All intermediate conditions are preserved, because:
(i) $R \Rightarrow \mathit{locate.rely}$,
(ii) $R \Rightarrow \mathrm{Preserve}(\mathit{ListInv})$,
(iii) $R \Rightarrow \mathrm{Preserve}(\mathrm{n.owner} = \mathsf{self})$,

(iv) $R \wedge \text{n.}\overleftarrow{\text{owner}} = \mathsf{self} \Rightarrow \text{ID}(\text{n.next}, \text{n.marked})$, *and*
(v) $R \wedge \text{n.}\overleftarrow{\text{owner}} = \mathsf{self} \wedge \mathit{ListInv} \Rightarrow \text{Preserve}(\text{Head} \rightarrow^* \text{n})$.

The guarantee condition is valid, because:

- Preserve($\mathit{ListInv}$), as $\mathit{ListInv}$ holds throughout the sequential proof.
- Preserve(n.marked); the only assignment to n.marked makes it true.
- $\mathsf{Mod}(\text{n.next}, \text{n.owner}, \text{n.marked})$ – by code inspection.
- $\text{n.}\mathit{LockGuar}$ – by code inspection.
- $\text{n.owner} \neq \mathsf{self} \Rightarrow \text{ID}(\text{n.next}, \text{n.marked})$ – by inspecting the sequential proof; the only such updates are to n1.next and n2.marked, but n1 and n2 are locked at these points.

□

7.6 Contains

The method $\mathit{Result} := \mathit{contains}(e)$ has the specification:

- Precondition: $\mathit{ListInv} \wedge -\infty < e < +\infty$
- Postcondition: $\mathit{ListInv} \wedge (\mathit{Result} = \mathit{AbsResult})$
- Rely: R
- Guarantee: G

As we explained in Section 7.5 of the extended abstract [4], this cannot be proved with the formal rules we have discussed so far without introducing a lot of abstract state. While there exists a proof making use of additional book-keeping state, we confined ourselves to an informal argument in the extended abstract. Here we present a proof of a much weaker property. We prove the post-condition:

$$\mathit{ListInv} \wedge (\mathit{Result} \Rightarrow (\mathit{AbsResult}_1 \wedge \mathit{AbsResult}_2))$$

If the function returns true, then e was in the list at the two semi-linearisation points: (i) the last loop iteration, and (ii) the test of curr.marked. In fact, it must also be in the list during the whole time period between the two points.

The algorithm, desugared and annotated:

$\mathit{contains}(e)$:
  $\langle \text{curr} := \text{Head} \rangle$;
  while $(\langle \text{curr.val} < e \rangle)$
    $\langle \text{curr} := \text{curr.next} ; \mathit{AbsResult}_1 := e \in \mathit{Abs} \rangle$;
  if $\text{curr.val} = e$ **then**
    $\langle b := \text{curr.marked} ; \mathit{AbsResult}_2 := e \in \mathit{Abs} \rangle$
    return $\neg b$
  else
    return false

Proof. First, note that the guarantee condition holds, as the code only modifies the local variables curr, *Result* and *AbsResult*. In particular, this implies Preserve(*ListInv*).

Let

$$I \stackrel{\text{def}}{=} \mathit{ListInv} \wedge -\infty < e < +\infty$$

which is preserved by interference.

In order to show that the correct result is returned, we establish the invariant:

$$J \stackrel{\text{def}}{=} (e \neq \text{curr.val} \vee \text{curr.marked} \vee \mathit{AbsResult}_1)$$

which is preserved by interference.

J becomes true after assignments to curr, because:

$\{\ I \wedge \text{n.val} < e\ \}$
$\Longleftrightarrow$
$\{\ I \wedge \text{n.val} < e \wedge \mathsf{Node}(\text{n.next})\ \}$
curr := n.next
$\mathit{AbsResult}_1 := e \in \mathit{Abs}$
$\{\ I \wedge \text{n.val} < e \wedge \text{n} \rightarrow \text{curr} \wedge \mathsf{Node}(\text{curr}) \wedge \mathit{AbsResult}_1 = (e \in \mathit{Abs})\ \}$

From *ListInv*,
$e \in \mathit{Abs} \Rightarrow \mathit{AbsResult}_1$
$\Longrightarrow (\exists\, \text{n.}\ \mathsf{Node}(\text{n}) \wedge \neg\text{n.marked} \wedge e = \text{n.val}) \Rightarrow \mathit{AbsResult}_1$
$\Longrightarrow (\mathsf{Node}(\text{curr}) \wedge \neg\text{curr.marked} \wedge e = \text{curr.val}) \Rightarrow \mathit{AbsResult}_1$
$\Longrightarrow \neg\mathsf{Node}(\text{curr}) \vee \text{curr.marked} \vee e \neq \text{curr.val} \vee \mathit{AbsResult}_1$
$\Longrightarrow e \neq \text{curr.val} \vee \text{curr.marked} \vee \mathit{AbsResult}_1$
as $\mathsf{Node}(\text{curr})$.
Thus by weakening, we get:
$\{\ I \wedge J \wedge \mathsf{Node}(\text{curr})\ \}$

$\{\ I\ \}$
curr := Head.next
$\mathit{AbsResult} := e \in \mathit{Abs}$
$\{\ I \wedge J \wedge \mathsf{Node}(\text{curr})\ \}$
while (curr.val $< e$)
$\{\ I \wedge J \wedge \mathsf{Node}(\text{curr}) \wedge \text{curr.val} < e\ \}$
$\Longrightarrow$
$\{\ I \wedge J \wedge \mathsf{Node}(\text{curr.next})\ \}$
curr := curr.next
$\mathit{AbsResult}_1 := e \in \mathit{Abs}$
$\{\ I \wedge J \wedge \mathsf{Node}(\text{curr})\ \}$

Therefore, after the loop:
$\{\ I \wedge J \wedge \mathsf{Node}(\text{curr}) \wedge e \leq \text{curr.val}\ \}$
if curr.val $= e$ **then**
$\{\ I \wedge \mathsf{Node}(\text{curr}) \wedge (\text{curr.marked} \vee \mathit{AbsResult}_1)\ \}$
$b :=$ curr.marked
$\mathit{AbsResult}_2 := e \in \mathit{Abs}$

$\{\ I \wedge (b \vee (AbsResult_1 \wedge AbsResult_2))\ \}$
return $\neg b$
$\{\ I \wedge Result \Rightarrow (AbsResult_1 \wedge AbsResult_2)\ \}$
else
$\{\ I \wedge J \wedge \mathsf{Node}(\text{curr}) \wedge e < \text{curr.val}\ \}$
return false
$\{\ I \wedge Result \Rightarrow (AbsResult_1 \wedge AbsResult_2)\ \}$

And, as usual, all the intermediate conditions are preserved by the rely condition. □

7.7 Final step

Now we can apply the lifting rule for the methods *add*, *remove* and get the required specifications. We can do the same for *contains*, modulo that we have not proved the post-condition $AbsResult = Result$ formally. Hence, we get the specifications:

- Precondition: $ListInv \wedge -\infty < e < +\infty$
- Postcondition: $AbsAdd(e)$ (resp. $AbsRemove(e)$, $AbsContains(e)$)
- Rely condition R; guarantee G

8 Conclusion

We have proved the safety of a practical list-based set implementation due to Heller et al [2] using rely-guarantee reasoning. Our proof demonstrates the power and applicability of R-G reasoning to fine-grain concurrency, as well as its limitations when a linearisation point cannot be precisely located. As future work, we would like to automate this proof, preferably by constructing a tool which would use R-G reasoning to prove the safety of an annotated program. Finally, we have only considered safety properties of the algorithm; reasoning about its liveness properties remains an open problem.

References

[1] J. Dingel. Computer-assisted assume/guarantee reasoning with VeriSoft. In *Proc. Int. Conference on Software Engineering (ICSE-25)*, pages 138–148, Portland, Oregon, USA, May 2003. IEEE Computer.

[2] S. Heller, M. Herlihy, V. Luchangco, M. Moir, B. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, Dec. 2005.

[3] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, July 1990.

[4] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proc. Symp. on Principles and Practice of Parallel Programming*. ACM Press, 2006.