Technical Report

UCAM-CL-TR-687
ISSN 1476-2986

Number 687

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

A marriage of rely/guarantee and separation logic

Viktor Vafeiadis, Matthew Parkinson

June 2007

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 marriage of rely/guarantee and separation logic

Viktor Vafeiadis
University of Cambridge

Matthew Parkinson
University of Cambridge

Abstract

In the quest for tractable methods for reasoning about concurrent algorithms both rely/guarantee logic and separation logic have made great advances. They both seek to tame, or control, the complexity of concurrent interactions, but neither is the ultimate approach. Rely-guarantee copes naturally with interference, but its specifications are complex because they describe the entire state. Conversely separation logic has difficulty dealing with interference, but its specifications are simpler because they describe only the relevant state that the program accesses.

We propose a combined system which marries the two approaches. We can describe interference naturally (using a relation as in rely/guarantee), and where there is no interference, we can reason locally (as in separation logic). We demonstrate the advantages of the combined approach by verifying a lock-coupling list algorithm, which actually disposes/frees removed nodes.

1 Introduction

Reasoning about shared variable concurrent programs is difficult, because the interference between the simultaneously executing threads must be taken into account. Our aim is to find methods that allow this reasoning to be done in a modular and composable way.

On the one hand, we have rely/guarantee, a well-established method, introduced by Jones, that is popular in the derivation and the post-hoc verification of concurrent algorithms [12]. RG provides a good way of describing interference by having two relations, the rely R and the guarantee G, which describe the state changes performed by the environment or by the program respectively. Its disadvantage is that the specification of interference is *global*: it must be checked against every state update, even if it is 'obvious' that the update cannot interfere with anything else. Even Jones [13] acknowledges this limitation and still considers the search for a satisfactory compositional approach to concurrency an 'open problem.'

On the other hand, the recent development of separation logic [20, 16] suggests that greater modularity is possible. There, the $*$ operator and the frame rule are used to carve all irrelevant state out of the specification and focus only on the state that matters for the execution of a certain component or thread. This makes specifications *local*; two components may interfere, only if they have overlapping specifications. Its disadvantage is that, in dealing with concurrent programs, it took the simplest approach and uses invariants to specify thread interaction. This makes expressing the relational nature of interference often quite difficult and requires many auxiliary variables [18]. Even O'Hearn acknowledges the weaknesses of separation logic, and asks if "a marriage between separation logic and rely-guarantee is also possible" [16].

Here we present such a marriage of rely/guarantee and separation logic, which combines their advantages and eliminates some of their weaknesses. We split the state into two disjoint parts: (*i*) the shared state which is accessible by all threads, and (*ii*) the local state which is accessible by a single component. Then, we use rely/guarantee to deal with the shared state, and separation logic to deal with the local state. This is best illustrated by our parallel composition rule:

$$\frac{\vdash C_1 \ \mathbf{sat}\ (p_1, R \cup G_2, G_1, q_1) \qquad \vdash C_2 \ \mathbf{sat}\ (p_2, R \cup G_1, G_2, q_2)}{\vdash C_1 \| C_2 \ \mathbf{sat}\ (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)}$$

This rule is identical to the standard rely/guarantee rule except for the use of $*$ instead of $\wedge$ in the pre- and post-conditions. In our specifications, the preconditions (e.g. p_1) and the postconditions (e.g. q_1) describe both the local and the shared state. The rely conditions (e.g. $R \cup G_2$) and the guarantee conditions (e.g. G_1) describe inter-thread interference: how the shared state gets modified.

The separating conjunction between assertions about both the local and the shared state splits local state (l) in two parts, but does not divide the shared state (s).

$$(p_1 * p_2)(l, s) \overset{\textbf{def}}{=} \exists l_1\, l_2.\ l = l_1 \uplus l_2 \wedge p_1(l_1, s) \wedge p_2(l_2, s)$$

The parallel composition rules of rely/guarantee and separation logic are special cases of our parallel composition rule. (1) When the local state is empty, then $p_1 * p_2 = p_1 \wedge p_2$ and we get the standard rely/guarantee rule. (2) When the shared state is empty, we do not need to describe its evolution (R and G are the identity relation). Then $p_1 * p_2$ has the same meaning as separation logic $*$, and we get the parallel rule of concurrent separation logic without resource invariants (see §2.2).

An important aspect of our approach is that the boundaries between the local state and the shared state are not fixed, but may change as the program runs. This “ownership transfer” concept is fundamental to proofs in concurrent separation logic.

In addition, as we encompass separation logic, we can cleanly reason about dynamically allocated data structures and explicit memory management, avoiding the need to rely on a garbage-collector. In §4, we demonstrate this by verifying a lock-coupling list algorithm, which actually disposes/frees removed nodes.

2 Technical background

In this paper, we reason about a parallel programming language with pointer operations. Let x, y and z range over logical variables, and x, y and z over program variables. We assume tid is a special variable that identifies the current thread. Commands C and expressions e are given by the following grammar,

$$\begin{aligned} C &::= \mathtt{x}\texttt{:=}e \mid \mathtt{x}\texttt{:=}[e] \mid [e_1]\texttt{:=}e_2 \mid \mathtt{x}\texttt{:=}\mathrm{cons}(e_1, \ldots, e_n) \mid \mathrm{dispose}(e) \\ &\quad \mid C_1; C_2 \mid C_1 \| C_2 \mid \mathbf{if}(b)\{C_1\}\,\mathbf{else}\,\{C_2\} \mid \mathbf{while}(b)\{C\} \mid \mathbf{atomic}(b)\{C\} \\ e &::= x \mid \mathrm{x} \mid e + e \mid n \end{aligned}$$

where b ranges over boolean expressions. Note that expressions e are *pure*: they do not refer to the heap. In the grammar, each assignment contains at most one heap access; assignments with multiple heap accesses can be performed using multiple assignments and temporary variables to store the intermediate results.

The semantics of $\mathbf{atomic}$ are that C will be executed in one indivisible step. This could be implemented through locking, hardware atomicity, transactional memories, etc. Choosing $\mathbf{atomic}$ over a given synchronisation primitive (e.g. locks) enables our reasoning to be applied at multiple abstraction levels. In any case, any synchronisation primitive can be encoded using $\mathbf{atomic}$.

In the rest of this section, we give a brief overview of the two logics we build on in this paper.

2.1 Interference – Rely/guarantee specifications

Rely/guarantee specifications [12] describe the interference between concurrently executing threads. These specifications are then used to prove concurrent algorithms in a compositional manner. Each component c is assigned a *rely* condition that describes the interference it can tolerate from its environment (namely, the other components of the system). In return, it is assigned a *guarantee* condition that characterises how it can interfere with the others.

The essence of rely/guarantee reasoning is its parallel composition rule. Two components (threads) may be placed in parallel, if and only if, the guarantee condition of the one component implies the rely condition of the other and vice versa.

$$\frac{\begin{array}{c}\vdash C_1 \ \mathbf{sat}\ (R \cup G_2, G_1) \\ \vdash C_2 \ \mathbf{sat}\ (R_2 \cup G_1, G_2)\end{array}}{\vdash C_1 \| C_2 \ \mathbf{sat}\ (R, G_1 \cup G_2)}$$

Since the interference experienced by thread C_1 can arise from C_2 or the environment of the parallel composition, we have to ensure that the total interference ($R \cup G_2$) is allowed. Similarly C_2 must be able to tolerate interference from C_1 and from the environment of the parallel composition. The interference caused by the parallel composition may be caused by either C_1 or C_2; so, the total interference must include the interferences caused by each component, G_1 and G_2 .

2.2 Local reasoning – Separation logic

In Hoare logic [10], assertions describe properties of the *whole* memory, and hence specifications, e.g. $\{P\}\ C\ \{Q\}$, describe a change of the whole memory. This is inherently *global reasoning*. Anything that is not explicitly preserved in the specification could be changed, for example $\{\mathtt{x} = 4\}$ `y:=5` $\{\mathtt{x} = 4\}$. Here `y` is allowed to change, even though it is not mentioned in the specification.[1]

The situation is different in *separation logic* [20]. Assertions describe properties of *part* of the memory, and hence specifications describe changes to *part* of the memory. The rest of the memory is guaranteed to be unchanged. This is the essence of *local reasoning*, specifications describe only the memory used by a command, its footprint.

The strength of separation logic comes from a new logical connective: the separating conjunction, $*$. $P * Q$ asserts the state can be split into two parts, one described by P and the other

[1] 'Modifies clauses' solve this problem, but they are neither pretty nor general.

```
{ArrSegBnd(a, first, last, min, max)}
qsort (a, first , last )   {
   local  pivot ;
   if ( first <last−1) {
      pivot  =  partition (a, first , last );
      qsort (a,   first  ,  pivot );
            || qsort(a, pivot,  last );
   }
}
{ArrSegSrt(a, first, last, min, max)}
```

Figure 1: Parallel Quicksort algorithm and specification

by Q. The separating conjunction allows us to formally capture the essence of *local reasoning* with the following rules:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \text{(Frame)} \qquad \frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}\ \text{(Par)}$$

The first rule says, if P is separate from R, and C transforms P into Q then if C finishes we have Q and separately still have R. The second rule says that if two threads have disjoint memory requirements, they can execute safely in parallel, and the postcondition is simply the composition of the two threads' postconditions.[2]

Example: Parallel Quicksort To motivate the use of separation logic, we verify parallel quicksort. Parallel quicksort uses disjoint concurrency, hence it is well suited to a separation logic proof: there is no interference.

We present the algorithm and specification in Figure 1. The algorithm's precondition, $\mathrm{ArrSegBnd}(\mathrm{a}, \mathrm{first}, \mathrm{last}, min, max)$, asserts that the heap contains a segment of array a, from index first to $\mathrm{last} - 1$, with values in the interval $[min, max]$. The postcondition denotes that this array segment is sorted. For simplicity, we omit saying that it is a permutation of the initial array segment. We specify the $\mathrm{partition}$ function as follows, but omit the source code and proof.

$$\begin{array}{l}
\{\mathrm{ArrSegBnd}(\mathrm{a}, \mathrm{first}, \mathrm{last}, min, max)\} \\
\mathrm{pivot} \ = \ \mathrm{partition}(\mathrm{a}, \mathrm{first}, \mathrm{last}) \\
\left\{\begin{array}{l} \exists X.\ \mathrm{ArrSegBnd}(\mathrm{a}, \mathrm{first}, \mathrm{pivot}, min, X) \\ * \ \mathrm{ArrSegBnd}(\mathrm{a}, \mathrm{pivot}, \mathrm{last}, X, max) \end{array}\right\}
\end{array}$$

The postcondition specifies the two segments are disjoint, hence, we can sort the two segments in parallel without interference.

$$\begin{array}{l}
\{\mathrm{ArrSegBnd}(\mathrm{a}, \mathrm{first}, \mathrm{pivot}, min, X) * \mathrm{ArrSegBnd}(\mathrm{a}, \mathrm{pivot}, \mathrm{last}, X, max)\} \\
\mathrm{qsort}(\mathrm{a}, \mathrm{first}, \mathrm{pivot});\ \|\ \mathrm{qsort}(\mathrm{a}, \mathrm{pivot}, \mathrm{last}); \\
\{\mathrm{ArrSegSrt}(\mathrm{a}, \mathrm{first}, \mathrm{pivot}, min, X) * \mathrm{ArrSegSrt}(\mathrm{a}, \mathrm{pivot}, \mathrm{last}, X, max)\}
\end{array}$$

[2] Originally, separation logic did not consider global variables as resource; hence the proof rules had nasty side-conditions. Later, this problem was solved by Bornat et al. [2]. By disallowing direct assignments to global variables, we avoid the problem.

To verify this algorithm with rely/guarantee, we would need to express that each parallel call modified disjoint elements of array. That is,

$$\begin{array}{ll} \text{guarantee:} & \forall i.(\text{first} \le i < \text{last}) \vee \text{a}[i] = \mathit{old}(\text{a}[i]) \\ \text{rely:} & \forall i.(\text{first} \le i < \text{last}) \Rightarrow \text{a}[i] = \mathit{old}(\text{a}[i]) \end{array}$$

In separation logic, however, we need not mention anything of this sort.

Brief details Separation logic has the following assertions for describing the heap, h:

$$P, Q, S ::= \mathbf{false} \mid \mathbf{emp} \mid e = e' \mid e \mapsto e' \mid \exists x.\, P \mid P \Rightarrow Q \mid P * Q \mid P \mathrel{-\!\!\circledast} Q$$

We encode $\neg, \wedge, \vee, \forall$, and $\mathbf{true}$ in the classical way. $\mathbf{emp}$ stands for the empty heap; $e \mapsto e'$ for the heap consisting of a single cell with address e and contents e'. Separating conjunction, $P * Q$, is the most important operator of separation logic. A heap h satisfies $P * Q$, if it can be split in two parts, one of which satisfies P and the other satisfies Q. We build heap descriptions of multiple cell heaps using $*$ and $\mapsto$. For example, $e \mapsto e' * f \mapsto f'$ describes two *separate* heap cells: it is impossible that e and f could be the same address (logically, if e and f are equal, then $e \mapsto f' * e \mapsto f'$ is false).

There remains one new connective to describe: *septraction*, $P \mathrel{-\!\!\circledast} Q$.[3] Intuitively, $P \mathrel{-\!\!\circledast} Q$ represents removing P from Q. Formally, it means the heap can be extended with a state satisfying P, and the extended state satisfies Q.

$$\begin{array}{rcl} h, i \vDash_{\text{SL}} (P * Q) & \overset{\textbf{def}}{=} & \exists h_1, h_2.\ (h_1 \uplus h_2 = h) \wedge h_1, i \vDash_{\text{SL}} P \wedge h_2, i \vDash_{\text{SL}} Q \\ h, i \vDash_{\text{SL}} (P \mathrel{-\!\!\circledast} Q) & \overset{\textbf{def}}{=} & \exists h_1, h_2.\ (h_1 \uplus h = h_2) \wedge h_1, i \vDash_{\text{SL}} P \wedge h_2, i \vDash_{\text{SL}} Q \end{array}$$

Finally, $e \mapsto e_1, \ldots, e_n$ is a shorthand for $(e \mapsto e_1) * \ldots * (e + n - 1 \mapsto e_n)$; and $e \mapsto \_$ means $\exists x \cdot e \mapsto x$.

Assignment to local variables are treated by the ordinary Hoare axiom, $\{Q[e/\text{x}]\}\ \text{x} := e\ \{Q\}$, where $Q[e/\text{x}]$ substitutes e for all occurrences of x in Q. The other axioms of separation logic are summarised below.

$$\begin{array}{c} \{e \mapsto \_\}\ [e] := e'\ \{e \mapsto e'\} \\ \{e = y \wedge e \mapsto z\}\ \text{x} := [e]\ \{y \mapsto z \wedge \text{x} = z\} \\ \{\mathbf{emp}\}\ \text{x} := \text{cons}(e_1, \ldots, e_n)\ \{\text{x} \mapsto e_1, \ldots, e_n\} \\ \{e \mapsto \_\}\ \text{dispose}(e)\ \{\mathbf{emp}\} \end{array}$$

(These are known as the small axioms, because they deal with the smallest heap affected by command. If there is more heap present, the frame rule says that it remains unaffected.)

- To write to a heap cell that cell must exist in the heap: i.e. you must own it.
- To read a cell $[e]$ you must own the cell; its contents are copied into variable x; the cell's contents are unchanged; and afterwards you still own it. (The logical variable y is used in case x occurs in e.)
- $\text{cons}(e_1, \ldots, e_n)$ allocates a new block of n heap cells. We require the heap is initially empty, and the postcondition contains the new block of cells.
- $\text{dispose}(e)$ deallocates a heap cell. We require the heap contains the cell being disposed; after disposal it is no longer contained in the heap.

[3] Sometimes called "existential magic wand", as it is the dual to "magic wand": $P \mathrel{-\!\!\circledast} Q \overset{\textbf{def}}{=} \neg(P \mathrel{-\!\!*} \neg Q)$. It has been used in the connection with modal logic in [4].

3 The combined logic

3.1 Describing interference

The strength of rely/guarantee is the careful description of interference between parallel processes. We describe interference in terms of actions $P \rightsquigarrow Q$ which describe the changes performed to the shared state. These resemble Morgan's *specification statements* [14], and P and Q will typically be linked with some existentially quantified logical variables. (We do not need to mention separately the set of modified shared locations, because these are all included in P.)

The meaning of an action $P \rightsquigarrow Q$ is that it replaces the part of the state that satisfies P before the action with a part satisfying Q. Its semantics is the following relation:

$$[\![P \rightsquigarrow Q]\!] = \{(h_1 \uplus h_0, h_2 \uplus h_0) \mid h_1, i \vDash_{\mathrm{SL}} P \wedge h_2, i \vDash_{\mathrm{SL}} Q\}$$

It relates some initial state h_1 satisfying the precondition P to a final state h_2 satisfying the postcondition. In addition, there may be some disjoint state h_0 which is not affected by the action. In the spirit of separation logic, we want action specifications as 'small' as possible, describing h_1 and h_2 but not h_0, and use the frame rule to perform the same update on a larger state.

The rely and guarantee conditions are simply sets of actions. Their semantics as a relation is the reflexive and transitive closure of the union of the semantics of each action in the set.

$$[\![P_1 \rightsquigarrow Q_1, \ldots, P_n \rightsquigarrow Q_n]\!] = \left(\bigcup_{i=1}^{n} [\![P_i \rightsquigarrow Q_i]\!]\right)^*$$

We shall write R for a syntactic rely condition (i.e. a set of actions) and $\mathcal{R}$ for a semantic rely condition (i.e. a binary relation).

3.2 Stability

Rely/guarantee reasoning requires that every pre- and post-condition in a proof is stable under environment interference. An assertion S is stable under interference of a relation $\mathcal{R}$ if and only if whenever S holds initially and we perform an update satisfying $\mathcal{R}$ then the resulting state still satisfies S.

Definition 1 (Stability). *$S; \mathcal{R} \Longrightarrow S$ iff for all s, s' and i such that $s, i \vDash_{\mathrm{SL}} S$ and $(s, s') \in \mathcal{R}$, then $s', i \vDash_{\mathrm{SL}} S$*

By representing the interference $\mathcal{R}$ as a set of actions, we reduce stability to a simple syntactic check. For a single action $[\![P \rightsquigarrow Q]\!]$, the following separation logic implication is necessary and sufficient:

Lemma 2. *$S; [\![P \rightsquigarrow Q]\!] \Longrightarrow S$ iff $\vDash_{\mathrm{SL}} (P \mathrel{-\!\!\circledast} S) * Q \implies S$.*

Informally, it says that if from a state that satisfies S, we subtract the part of the state satisfying P, and replace it with some state satisfying Q, then the result should still satisfy S. When the action cannot fire because there is no substate of S satisfying P, then $P \mathrel{-\!\!\circledast} S$ is false and the implication holds trivially.

An assertion S is stable under interference of a set of actions R when it is stable under interference of every action in R.

Lemma 3. $S; (\mathcal{R}_1 \cup \mathcal{R}_2)^* \Longrightarrow S$ *iff* $S; \mathcal{R}_1 \Longrightarrow S$ *and* $S; \mathcal{R}_2 \Longrightarrow S$.

Finally, we define $\text{wssa}_{\mathcal{R}}(Q)$ to be the weakest assertion that is stronger than Q and stable under $\mathcal{R}$.

Definition 4 (Weakest stable stronger assertion)**.** *(1)* $\textit{wssa}_{\mathcal{R}}(Q) \Rightarrow Q$*,*
(2) $\textit{wssa}_{\mathcal{R}}(Q); \mathcal{R} \Longrightarrow \textit{wssa}_{\mathcal{R}}(Q)$*, and*
(3) for all P*, if* $P; \mathcal{R} \Longrightarrow P$ *and* $P \Rightarrow Q$*, then* $P \Rightarrow \textit{wssa}_{\mathcal{R}}(Q)$*.*

3.3 Local and shared state assertions

We can specify a state using two assertions, one describing the local state and the other the shared state. However, this approach has some drawbacks: specifications are longer, and extending the logic to a setting with multiple disjoint regions of shared state is clumsy.

Instead, we consider a unified assertion language that describes both the local and the shared state. This is done by extending the positive fragment of separation logic assertions with 'boxed' terms. We could use boxes for both local and shared assertions: for example, $\boxed{P}_{\text{local}}$ and $\boxed{P}_{\text{shared}}$. However, since $\boxed{P}_{\text{local}} * \boxed{Q}_{\text{local}} \iff \boxed{P * Q}_{\text{local}}$ holds for *, and all the classical operators, we can omit the $\boxed{\_}_{\text{local}}$ and the "$_{\text{shared}}$" subscript. Hence the syntax of assertions is

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x.\ p \mid \forall x.\ p$$

Semantically, we split the state, σ, of the system into two components: the local state l, and the shared state s. Each component state may be thought to be a partial finite function from locations to values. We require that the domains of the two states are disjoint, so that the total state is simply the (disjoint) union of the two states. Assertions without boxes describe purely the local state l, whereas a boxed assertion $\boxed{P}$ describes the shared state s. Formally, we give the semantics with respect to a 'rely' condition R, a set of actions describing the environment interference:

$$\begin{array}{lcl}
l, s, i \vDash_R P & \iff & l, i \vDash_{\text{SL}} P \\
l, s, i \vDash_R \boxed{P} & \iff & l = \emptyset \wedge s, i \vDash_{\text{SL}} P \\
l, s, i \vDash_R p_1 * p_2 & \iff & \exists l_1, l_2.\ (l = l_1 \uplus l_2) \wedge (l_1, s, i \vDash_R p_1) \wedge (l_2, s, i \vDash_R p_2) \\
l, s, i \vDash_R p_1 \wedge p_2 & \iff & (l, s, i \vDash_R p_1) \wedge (l, s, i \vDash_R p_2) \\
l, s, i \vDash_R p_1 \vee p_2 & \iff & (l, s, i \vDash_R p_1) \vee (l, s, i \vDash_R p_2) \\
l, s, i \vDash_R \forall x.\ p & \iff & \forall v.\ (l, s, i[x \mapsto v] \vDash_R p) \\
l, s, i \vDash_R \exists x.\ p & \iff & \exists v.\ (l, s, i[x \mapsto v] \vDash_R p)
\end{array}$$

Note that $*$ is multiplicative over the local state, but additive over the shared state. Hence, $\boxed{P} * \boxed{Q} \Longrightarrow \boxed{P \wedge Q}$. The semantics of shared assertions, $\boxed{P}$, could alternatively be presented without $l = \emptyset$. This results in an equally expressive logic, but the definition above leads to shorter assertions in practice.

We use $\text{wssa}_{[\![R]\!]}(\_)$ to make assertions semantically resistant to interference:

Lemma 5. *If* $(l, s, i \vDash_R p)$*,* $s' \uplus l$ *defined and* $[\![R]\!](s, s')$ *then* $(l, s', i \vDash_R p)$*.*

We define an assertion to be syntactically stable if each of the assertions about the shared state is stable. By construction, any assertion about the local state of a component is unaffected by other components, because interference can happen only on the shared state. On the other hand, a boxed assertion $\boxed{S}$ may be affected.

Definition 6 (Stable assertion)**.** *1. P stable under R always;*

2. $\boxed{P}$ stable under R iff $P; [\![R]\!] \implies P$;

3. $(p\ op\ q)$ stable under R iff p stable under R and q stable under R; and

4. $(qu\,x.\ p)$ stable under R iff p stable under R
where $op ::= \wedge \mid \vee \mid *$ and $qu ::= \forall \mid \exists$.

This syntactic condition allows us to change the interpretation of a formula to a more permissive rely.

Lemma 7. *If $(l, s, i \vDash_R p)$, $[\![R]\!] \subseteq [\![R']\!]$ and p stable under R' then $(l, s, i \vDash_{R'} p)$. Note that $(l, s, i \vDash_R p)$ and $[\![R']\!] \subseteq [\![R]\!]$ then $(l, s, i \vDash_{R'} p)$.*

We present a few entailments for formulae involving shared states.

$$\frac{P \vdash_{\text{SL}} Q}{\boxed{P} \vdash \boxed{Q}} \qquad \boxed{P} \wedge \boxed{Q} \vdash \boxed{P \wedge Q} \qquad \boxed{P} \vee \boxed{Q} \vdash \boxed{P \vee Q} \qquad \boxed{P} * \boxed{Q} \vdash \boxed{P \wedge Q}$$

$$\forall x.\ \boxed{P} \vdash \boxed{\forall x.\ P} \qquad \exists x.\ \boxed{P} \vdash \boxed{\exists x.\ P} \qquad \boxed{P} \vdash \boxed{P} * \boxed{P} \qquad \boxed{P} \vdash \mathbf{emp}$$

Relationship to linear logic The use of two kinds of assertion perhaps calls to mind some presentations of linear logic, where there are two zones in sequents [9]. However, our two kinds of assertion, and the passage between them using Box, do not match the two kinds (linear and intuitionistic) in linear logic, and their passage using !. In particular, our box operator does not satisfy Dereliction ($!A \vdash A$) and Promotion ($\frac{!A \vdash B}{!A \vdash !B}$), although it does satisfy Weakening ($!A \vdash \mathbf{emp}$) and Contraction ($!A \vdash !A * !A$). It is as if we had two substructural zones, rather than one substructural and one additive.

3.4 Ownership transfer

Usually the precondition and postcondition of an action have the same heap footprint. For example, consider the action saying that x can be incremented:

$$\mathtt{x} \mapsto M \quad \leadsto \quad \mathtt{x} \mapsto N \wedge N \geq M \qquad \text{(Increment)}$$

If they have a different footprints, this indicates a transfer of ownership between the shared state and the local state of a thread. Consider a simple lock with two operations: (Acq) which changes the lock bit from 0 to 1, and removes the protected object, $list(y)$, from the shared state; and (Rel) which changes the lock bit from 1 to 0, and replaces the protected object into the shared state. We can represent these two operations formally as

$$(\mathtt{x} \mapsto 0) * list(\mathtt{y}) \leadsto \mathtt{x} \mapsto 1 \quad \text{(Acq)} \qquad\qquad \mathtt{x} \mapsto 1 \leadsto (\mathtt{x} \mapsto 0) * list(\mathtt{y}) \quad \text{(Rel)}$$

3.5 Specifications and proof rules

The judgement $\vdash C\ \mathbf{sat}\ (p, R, G, q)$ semantically says that any execution of C from an initial state satisfying p and under interference at most R, (*i*) does not fault (e.g. access unallocated memory), (*ii*) causes interference at most G, and, (*iii*) if it terminates, its final state satisfies q.

Hence, we get the familiar refinement rule.

$$\frac{\vdash C\ \mathbf{sat}\ (p', R', G', q') \quad \begin{matrix} R \Rightarrow R' & p \Rightarrow p' \\ G' \Rightarrow G & q' \Rightarrow q \end{matrix}}{\vdash C\ \mathbf{sat}\ (p, R, G, q)}$$

From separation logic, we inherit the frame rule. This rule says that a program safely running with initial state p can also be executed with additional state r. As the program runs safely without r, it cannot access r when it is present; hence, r is still true at the end. The additional premise is needed because r might mention the shared state and C might modify it in an $\mathbf{atomic}$.

$$\frac{\begin{matrix} \vdash C\ \mathbf{sat}\ (p, R, G, q) \\ \left(\begin{matrix} (r \text{ stable under } R \cup G) \\ \vee\ (C \text{ has no } \mathbf{atomic}\text{s}) \end{matrix}\right) \end{matrix}}{\vdash C\ \mathbf{sat}\ (p * r, R, G, q * r)}$$

We adopt all of the small axioms for local state from separation logic (not presented) [15]. Additionally, we have a read axiom for shared state, which allows a non-atomic read from a shared location if we can rely on its value not changing. Note that we do not need to check stability for this read.

$$\frac{Q = (P * X \mapsto Y) \quad x \notin fv(P)}{\vdash (x := [e])\ \mathbf{sat}\ (\boxed{Q} \wedge e = X, R, G, \boxed{Q} \wedge x = Y)}$$

The next rule is that of conditional critical regions $\mathbf{atomic}(b)\{C\}$. For clarity, we present the rule where the guard b is just $\mathbf{true}$. The general case, where b is non-trivial and may access the heap, just complicates the essential part of the rule. A simple rule for critical regions would be the following:

$$\frac{\vdash C\ \mathbf{sat}\ (P, \{\}, \{\}, Q) \quad (P \leadsto Q) \subseteq G \quad \boxed{Q} \text{ stable under } R}{\vdash (\mathbf{atomic}\{C\})\ \mathbf{sat}\ (\boxed{P}, R, G, \boxed{Q})}$$

As in RG, we must check that the postcondition is stable under interference from the environment, and that changing the shared state from P to Q is allowed by the guarantee G.

This rule is sound, but too weak in two ways. First, it does not allow critical regions to access any local state, as the precondition $\boxed{P}$ requires that the local state is empty. Second, it requires that the critical region changes the *entire* shared state from P to Q and that the guarantee condition allows such a change. Thus, we extend the rule by (*i*) adding a precondition P_2 and a postcondition Q_2 for the local state, and (*ii*) allowing the region to change a part P_1 of P into a part Q_1 of Q, ensuring that the rest F does not change. Additionally, we allow some existential quantifiers, $\overline{y}$ in the shared state to be pulled out over both the shared and local state.

$$\frac{\begin{matrix} \vdash C\ \mathbf{sat}\ (P_1 * P_2, \{\}, \{\}, Q_1 * Q_2) & \boxed{Q} \text{ stable under } R \\ \overline{y} \cap FV(P_2) = \emptyset \quad P \Rightarrow P_1 * F \quad Q_1 * F \Rightarrow Q & (P_1 \leadsto Q_1) \subseteq G \end{matrix}}{\vdash (\mathbf{atomic}\{C\})\ \mathbf{sat}\ (\boxed{\exists \overline{y}.\ P} * P_2, R, G, \exists \overline{y}.\ \boxed{Q} * Q_2)}$$

A specification, $P_1 \leadsto Q_1$ is allowed by a guarantee G if its effect is contained in G. Fig. 2 provides rules to approximate this definition in proofs. The rule G-SEQ allows actions to be sequenced and builds in a form of framing. Note that, if S is empty, then the rule is a parallel composition of two actions; if P_2 and Q_1 are empty, then the rule sequences the actions. It would be simpler, if we simply included the frame rule however this is unsound. In fact, the coframe rule G-COFRM is admissible. G-CONS is similar to the rule of consequence, but the second implication is reversed, $Q \Rightarrow Q'$. Semantically, the property is defined as follows:

$$\frac{}{x \mapsto y \rightsquigarrow x \mapsto y \subseteq G}\ \text{G-E}\textsc{xact} \qquad \frac{P \rightsquigarrow Q \in G}{P \rightsquigarrow Q \subseteq G}\ \text{G-A}\textsc{x}$$

$$\frac{P_1 \rightsquigarrow S * Q_1 \subseteq G \quad P_2 * S \rightsquigarrow Q_2 \subseteq G}{P_1 * P_2 \rightsquigarrow Q_1 * Q_2 \subseteq G}\ \text{G-S}\textsc{eq} \qquad \frac{P \rightsquigarrow Q \subseteq G}{P[e/x] \rightsquigarrow Q[e/x] \subseteq G}\ \text{G-S}\textsc{ub}$$

$$\frac{\models_{\mathrm{SL}} P' \Rightarrow P \quad P \rightsquigarrow Q \subseteq G \quad \models_{\mathrm{SL}} Q' \Rightarrow Q}{P' \rightsquigarrow Q' \subseteq G}\ \text{G-C}\textsc{ons} \qquad \frac{(P * F) \rightsquigarrow (Q * F) \subseteq G}{P \rightsquigarrow Q \subseteq G}\ \text{G-C}\textsc{o}\text{F}\textsc{rm}$$

Figure 2: Rules and axioms for guarantee allows an action.

Definition 8. $P \rightsquigarrow Q \subseteq G$ *iff* $[\![P \rightsquigarrow Q]\!] \subseteq [\![G]\!]$.

There is a side-condition to the atomic rule requiring that Q is a *precise* assertion. This is formally defined in §5 (Def. 17). This is a technical requirement inherited from concurrent separation logic. It ensures that the splitting of the resultant state into local and shared portions is unambiguous.

We reiterate the parallel composition rule from the introduction. As the interference experienced by thread C_1 can arise from C_2 or the environment of the parallel composition, we have to ensure that this interference $R \cup G_2$ is allowed. Similarly C_2 must be able to tolerate interference from C_1 and from the environment of the parallel composition.

$$\frac{\begin{array}{ll}\vdash C_1 \ \mathbf{sat}\ (p_1, R \cup G_2, G_1, q_1) & p_1 \ \mathsf{stable\ under}\ R \cup G_1 \\ \vdash C_2 \ \mathbf{sat}\ (p_2, R \cup G_1, G_2, q_2) & p_2 \ \mathsf{stable\ under}\ R \cup G_2\end{array}}{\vdash C_1 \| C_2 \ \mathbf{sat}\ (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)}$$

The precondition and postcondition of the composition are the separating conjunction, $*$, of the preconditions/postconditions of the individual threads. In essence, this is the conjunction of the shared state assertions, and the separating conjunction of the local state assertions (cf. the semantics of $*$ in §3.3).

The proof rules for sequential composition, conditional and iterative commands are completely standard.

$$\frac{\begin{array}{c}\vdash C_1 \ \mathbf{sat}\ (p \wedge b, R, G, q) \\ \vdash C_2 \ \mathbf{sat}\ (p \wedge \neg b, R, G, q)\end{array}}{\vdash (\mathbf{if}(b)\{C_1\}\ \mathbf{else}\ \{C_2\})\ \mathbf{sat}\ (p, R, G, q)} \qquad \frac{\begin{array}{c}\vdash C_1 \ \mathbf{sat}\ (p, R, G, r) \\ \vdash C_2 \ \mathbf{sat}\ (r, R, G, q)\end{array}}{\vdash (C_1; C_2)\ \mathbf{sat}\ (p, R, G, q)}$$

$$\frac{}{\mathbf{skip\ sat}\ (p, R, G, p)} \qquad \frac{\vdash C \ \mathbf{sat}\ (p \wedge b, R, G, p)}{\vdash (\mathbf{while}(b)\{C\})\ \mathbf{sat}\ (p, R, G, p \wedge \neg b)}$$

We can also extend our logic to deal with parameterless procedures in the usual way, where the environment Γ maps procedure names to their specifications. The previous rules all simply pass Γ around unmodified. When encountering a procedure call, we apply the following rules.

$$\frac{\Gamma, \mathit{proc}\ \mathbf{sat}\ (p, R, G, q) \vdash C \ \mathbf{sat}\ (p, R, G, q)}{\Gamma \vdash \mathit{proc}\ \mathbf{sat}\ (p, R, G, q)} \ \text{provided } \mathit{proc} \text{ has body } C.$$

$$\Gamma, \mathit{proc}\ \mathbf{sat}\ (p, R, G, q) \vdash \mathit{proc}\ \mathbf{sat}\ (p, R, G, q)$$

We omit parameters and return values to procedures/functions as they can be encoded into the heap.

```
lock(p) {
  atomic(p.lock==0){
    p.lock = tid;
 //p.oldn = p.next;
  }
}
unlock(p) {
  atomic(true) {
    p.lock = 0;
  }
}
```

```
locate(e) {
  local p,c;
  p = Head;
  lock(p);
  c = p.next;
  while(c.value<e){
    lock(c);
    unlock(p);
    p = c;
    c = p.next;
  }
  return (p,c);
}
```

```
add(e) {
  local x,y,z;
  (x,z)=locate(e);
  if(z.value!=e){
    y = cons(0,e,z);
    x.next = y;
  }
  unlock(x);
}
```

```
remove(e) {
  local x,y,z;
  (x,y)=locate(e);
  if(y.value==e){
    lock(y);
    z = y.next;
    x.next = z;
    unlock(x);  // A
    dispose(y);
  } else {
    unlock(x);
  }
}
```

Figure 3: Source code for lock coupling list operations. For clarity, we use a field notation, hence we encode $\mathrm{p.lock}$, $\mathrm{x.value}$, $\mathrm{x.next}$ and $\mathrm{p.oldn}$ as $[\mathrm{p}]$, $[\mathrm{x}+1]$, $[\mathrm{x}+2]$ and $[\mathrm{p}+3]$, respectively. Commented code is auxiliary, that is, required only for the proof. We use heap reads in conditional tests for ifs and whiles, which can be encoded using an additional local variable for the heap read.

4 Example

This section uses the new logic to verify a fine-grained concurrent linked list implementation of a mutable set data structure (see Fig. 3). It has operations add which adds an element to the set, and remove which removes an element from the set.

The algorithm associates one lock per list node rather than have a single lock for the entire list. Traversing the list uses *lock coupling*: the lock on one node is not released until the next node is locked. Somewhat like a person climbing a rope “hand-over-hand,” you always have at least one hand on the rope.

An element is added to the set by inserting it in the appropriate position, while holding the lock of its previous node. It is removed by redirecting the previous node’s pointer, while both the previous and the current node are locked. This ensures that deletions and insertions can happen concurrently in the same list. The algorithm makes two assumptions about the list: (1) it is sorted; and (2) the first and last elements have values $-\infty$ and $+\infty$ respectively. This allows us to avoid checking for the end of the list.

Node predicates We use three predicates to represent a node in the list: (1) $N_s(x,v,y)$, for a node at location x with contents v and tail pointer y and with the lock status set to s; (2) $U(x,v,y)$ for an unlocked node at location x with contents v and tail pointer y; and (3) $L_t(x,v,y)$ for a node locked with thread identifier t. We use $N_{\_}(x,v,y)$ for a node that may or may not be locked.

$$N_s(x,v,y) \overset{\textbf{def}}{=} x \mapsto s, v * \left(\begin{array}{l}(s=0 \land x+2 \mapsto y, \_)\\ \lor\,(s \neq 0 \land x+3 \mapsto y)\end{array}\right) \land x \bmod 4 = 0$$

$$U(x,v,y) \overset{\textbf{def}}{=} N_0(x,v,y) \qquad\qquad L_t(x,v,y) \overset{\textbf{def}}{=} N_t(x,v,y) \land t > 0$$

We assume nodes are aligned, $x \bmod 4 = 0$, and $\mathbf{cons}$ returns aligned nodes.[4] The thread identifier parameter in the locked node is required to specify that a node can only be unlocked by the thread that locked it. The fourth field/cell is auxiliary. It is used to store the last value of the nodes tail before it was locked. Once a node is locked its tail field is released to the locking thread, allowing it to mutate the field outside of critical sections, the auxiliary field is used in the proof to track the list structure.

Actions The algorithm does four kinds of actions: (1) lock, which locks a node, (2) unlock, which unlocks a node, (3) add, which inserts a new node to the list, and (4) delete, which removes a node from the list. All of these actions are parameterised with a set of thread identifiers, T. This allows us to use the actions to represent both relies and guarantees. In particular, we take a thread with identifier tid to have the guarantee with $T = \{\mathrm{tid}\}$, and the rely to use the complement of this set. Let $I(T)$ be the set of these four actions.

The first two actions are straightforward:

$$t \in T \wedge U(x, v, n) \rightsquigarrow L_t(x, v, n) \tag{lock}$$

$$t \in T \wedge L_t(x, v, n) \rightsquigarrow U(x, v, n) \tag{unlock}$$

Now, consider adding a node to the list. We begin by describing an action that ignores the sorted nature of the list:

$$t \in T \wedge L_t(x, u, n) \rightsquigarrow L_t(x, u, m) * U(m, v, n)$$

To add an element to the list, we must have locked the previous node, and then we can swing the tail pointer to the added node. The added node must have the same tail as previous node before the update. To preserve the sorted order of the list, the actual add action must also mention the next node: the inserted value must be between the previous and the next values.

$$\begin{aligned}&(t \in T) \wedge (u < v < w) \wedge (L_t(x, u, n) * N_s(n, w, y))\\ &\qquad\qquad \rightsquigarrow L_t(x, u, m) * U(m, v, n) * N_s(n, w, y)\end{aligned} \tag{add}$$

The final action we allow is removing an element from the list. We must lock the node we wish to delete, n, and its previous node, x. The tail of the previous node must be updated to the deleted node's tail, m.

$$(v < \infty) \wedge (t \in T) \wedge (L_t(x, u, n) * L_t(n, v, m)) \rightsquigarrow L_t(x, u, m) \tag{delete}$$

We summarise these actions pictorially in figure 4.

List predicate We use separation to describe the structure of the shared list. The predicate $ls(x, A, y)$ describes a list segment starting at location x with the final tail value of y, and with contents A. We use $\cdot$ as a list separator.

$$ls(x, \emptyset, x) \overset{\text{def}}{=} \mathbf{emp} \qquad ls(x, v{\cdot}B, y) \overset{\text{def}}{=} (\exists z.\ x \neq y \wedge N_{\_}(x, v, z) * ls(z, B, y))$$

[4] Without this restriction a node could be formed by parts of two adjacent nodes. Instead of assuming alignment, this problem can also be solved by allowing contexts in actions, for example the node is reachable from the head.

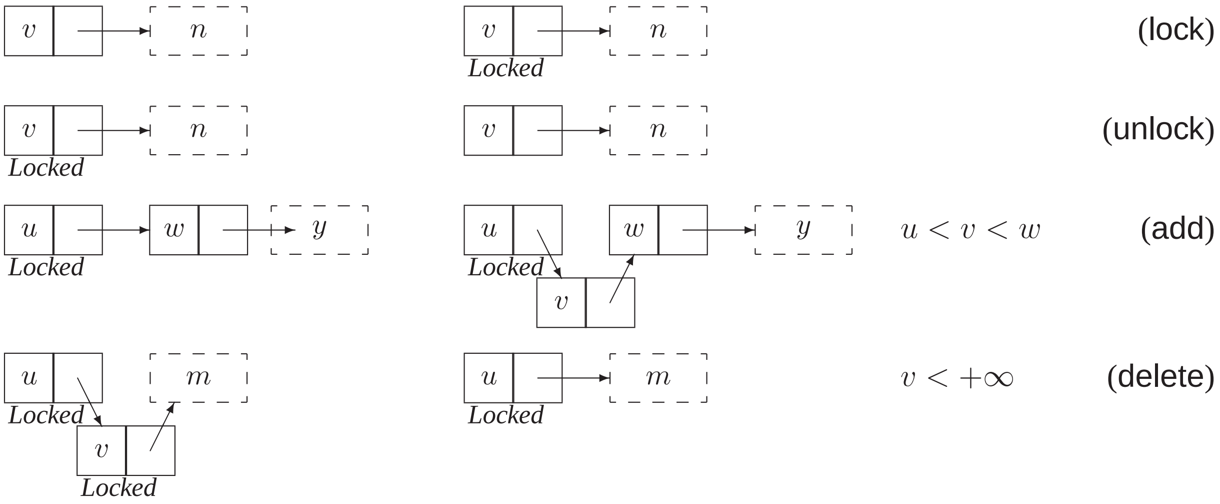

Figure 4: Pictorial representation of the actions

Note, as we use separation logic we do not need any reachability predicates, our predicate is simply a recursively defined predicate. The use of $*$ and the inequality $x \neq y$ ensures the list is acyclic.

We have three basic properties of a list segment: (1) it does not contain the end marker; (2) an element can be added to the end, provided its tail pointer does not point to anything in the list; and (3) two lists can be appended, provided the end marker of the second list is not contained in the first.

Definition 9. $P\lfloor_x \overset{\textbf{def}}{=} P \wedge \neg(x \mapsto \_ * true)$

Lemma 10. *1.* $ls(w, A, z) \iff ls(w, A, z)\lfloor_z$
2. $ls(w, A, x)\lfloor_y * N_s(x, v, y) \Rightarrow ls(w, A{\cdot}v, y)$
3. $ls(w, A, x)\lfloor_y * ls(x, B, y) \Rightarrow ls(w, A{\cdot}B, y)$

Finally, we give a lemma that enables us to delete a node from a list segment.

Proposition 11. $(N_s(x, v, y) \multimap\!\!\circledast\ ls(w, A, z))$ *is equivalent to* $\exists BC.\,(A = B{\cdot}v{\cdot}C) \wedge w \neq z \wedge \big(ls(w, B, x)\lfloor_z * ls(y, C, z)\lfloor_x\big)$

The algorithm works on sorted lists with the first and last values being $-\infty$ and $+\infty$ respectively. $s(A)$ represents this restriction on a logical list A.

$$srt(+\infty{\cdot}\epsilon) \overset{\textbf{def}}{=} \textbf{emp} \qquad srt(a{\cdot}b{\cdot}A) \overset{\textbf{def}}{=} srt(b{\cdot}A) \wedge a < b \qquad s(-\infty{\cdot}A) \overset{\textbf{def}}{=} srt(A)$$

Main proof Appendix A contains the proof outlines. The outline presents the intermediate assertions in the proof. We present one step of the verification of remove function in detail: the unlock action labelled “A” in Fig. 3. For simplicity, we inline the unlock body.

$\left\{\boxed{\exists AB.\, ls(\mathrm{Head}, A, \mathrm{x}) * L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) * ls(\mathrm{z}, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}B)} * (\mathrm{x}{+}2 \mapsto \mathrm{z})\right\}$
atomic$\{\{L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) * (\mathrm{x}{+}2 \mapsto \mathrm{z})\}$x.lock = 0;$\{U(\mathrm{x}, u, \mathrm{z}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z})\}\}$
$\left\{\boxed{\exists A.\, ls(\mathrm{Head}, A, \mathrm{nil}) * s(A)} * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z})\right\}$

We must prove four things: (1) the body meets its specification; (2) the body's specification is allowed by the guarantee; (3) the outer specification's postcondition is stable; and (4) find a frame, F, that satisfies the two implications:

1. $\{L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) * (\mathrm{x}{+}2 {\mapsto} \mathrm{z})\}$ x.lock = 0; $\{U(\mathrm{x}, u, \mathrm{z}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z})\}$

2. $L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) \rightsquigarrow U(\mathrm{x}, u, \mathrm{z}) \subseteq I(\{\mathrm{tid}\})$

3. $\boxed{\exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) * s(A))}$ stable under $I(\overline{\{\mathrm{tid}\}})$

4. $ls(\mathrm{Head}, A, \mathrm{x}) * L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) * ls(\mathrm{z}, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}B)$
$\qquad\qquad \Longrightarrow L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) * F$
$U(\mathrm{x}, u, \mathrm{z}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) * F \Longrightarrow \exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) * s(A)$

The first is a simple proof in separation logic. The second follows as:

$$\frac{\begin{array}{l} L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) \rightsquigarrow L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{z}) \subseteq I(\{\mathrm{tid}\}) \\ L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{z}) \rightsquigarrow U(\mathrm{x}, u, V z) \subseteq I(\{\mathrm{tid}\}) \end{array}}{L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) \rightsquigarrow U(\mathrm{x}, u, \mathrm{z}) \subseteq I(\{\mathrm{tid}\})} \ \textsc{G-Seq}$$

Third, to show $\boxed{\exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) * s(A)}$ is stable, we use Lemma 2 for the four actions in the rely: lock, unlock, add and delete.

(lock): We must show that:

$$((U(x, v, n) \mathbin{-\!\!\circledast} ls(y, A, z)) * L_t(x, v, n)) \Rightarrow ls(y, A, z)$$
$$((U(x, v, n) \mathbin{-\!\!\circledast} s(A{\cdot}u{\cdot}v{\cdot}B)) * L_t(x, v, n)) \Rightarrow s(A{\cdot}u{\cdot}v{\cdot}B))$$

The first follows as

$$\begin{array}{ll} & (U(x, v, n) \mathbin{-\!\!\circledast} ls(y, A, z)) * L_t(x, v, n) \\ \Rightarrow & (ls(y, B, x)\rfloor_z * ls(n, C, z) * L_t(x, v, n)) \wedge (A = B{\cdot}v{\cdot}C) \\ \Rightarrow & (ls(y, B, x)\rfloor_z * ls(x, v{\cdot}C, z)) \wedge (A = B{\cdot}v{\cdot}C) \Rightarrow ls(y, A, z) \end{array}$$

and the second as

$$(U(x, v, n) \mathbin{-\!\!\circledast} s(A)) * L_t(x, v, n) \quad \Rightarrow \quad \mathbf{false} * L_t(x, v, n) \ \Rightarrow \ \mathbf{false} \ \Rightarrow \ s(A)$$

(unlock): This follows in a very similar way to (lock).

(add): We omit the case where we delete from $s(A)$ as this follows trivially. Assume $u < v < w$ and simplifying because $L_t(x, u, m) * U(m, v, n) * N_s(n, w, y) \Rightarrow ls(x, u{\cdot}v{\cdot}w, y)$

$$\begin{array}{ll} & \left(\begin{pmatrix} L_t(x, u, n) \\ * \ N_s(n, w, y) \end{pmatrix} \mathbin{-\!\!\circledast} ls(\mathrm{Head}, A, \mathrm{nil})\right) \\ & \qquad\qquad * \ s(A) * ls(x, u{\cdot}v{\cdot}w, y) \\ \Rightarrow & \left(N_s(n, w, y) \mathbin{-\!\!\circledast} \begin{pmatrix} ls(\mathrm{Head}, B, x) \\ * \ ls(n, C, \mathrm{nil}) \end{pmatrix}\right) \\ & \qquad\qquad * \ s(B{\cdot}u{\cdot}C) * ls(x, u{\cdot}v{\cdot}w, y) \\ \Rightarrow & \left(N_s(n, w, y) \mathbin{-\!\!\circledast} \begin{pmatrix} ls(\mathrm{Head}, B, x) \\ * \ N_{s'}(n, w', y') \\ * \ ls(y', C', \mathrm{nil}) \end{pmatrix}\right) \\ & \qquad\qquad * \ s(B{\cdot}u{\cdot}w'{\cdot}C') * ls(x, u{\cdot}v{\cdot}w, y) \\ \Rightarrow & ls(\mathrm{Head}, B, x) * ls(y, C', \mathrm{nil}) \\ & \qquad\qquad * \ s(B{\cdot}u{\cdot}w{\cdot}C') * ls(x, u{\cdot}v{\cdot}w, y) \\ \Rightarrow & ls(\mathrm{Head}, B{\cdot}u{\cdot}v{\cdot}w{\cdot}C', \mathrm{nil}) * s(B{\cdot}u{\cdot}w{\cdot}C') \\ \Rightarrow & ls(\mathrm{Head}, B{\cdot}u{\cdot}v{\cdot}w{\cdot}C', \mathrm{nil}) * s(B{\cdot}u{\cdot}v{\cdot}w{\cdot}C') \\ \Rightarrow & \exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) * s(A) \end{array}$$

(delete):

We omit the case where we delete from $s(A)$ as this follows trivially. Assume $u < +\infty$ and simplifying because $L_t(x, u, y) \Rightarrow \mathit{ls}(x, u, y)$

$$\begin{array}{rl}
& \left(\begin{pmatrix} L_t(x,u,n) \\ * \, L_t(n,w,y) \end{pmatrix} \mathrel{-\!\!\circledast} \mathit{ls}(\mathrm{Head}, A, \mathrm{nil}) \right) \\
& \hfill * \, s(A) * \mathit{ls}(x,u,y) \\
\Rightarrow & \left(L_t(n,w,y) \mathrel{-\!\!\circledast} \begin{pmatrix} \mathit{ls}(\mathrm{Head}, B, x) \\ * \, \mathit{ls}(n, C, \mathrm{nil}) \end{pmatrix} \right) \\
& \hfill * \, s(B{\cdot}u{\cdot}C) * \mathit{ls}(x,u,y) \\
\Rightarrow & \left(L_t(n,w,y) \mathrel{-\!\!\circledast} \begin{pmatrix} \mathit{ls}(\mathrm{Head}, B, x) \\ * \, N_{s'}(n, w', y') \\ * \, \mathit{ls}(y', C', \mathrm{nil}) \end{pmatrix} \right) \\
& \hfill * \, s(B{\cdot}u{\cdot}w'{\cdot}C') * \mathit{ls}(x,u,y) \\
\Rightarrow & \mathit{ls}(\mathrm{Head}, B, x) * \mathit{ls}(y, C', \mathrm{nil}) \\
& \hfill * \, s(B{\cdot}u{\cdot}w{\cdot}C') * \mathit{ls}(x,u,y) \\
\Rightarrow & \mathit{ls}(\mathrm{Head}, B{\cdot}u{\cdot}C', \mathrm{nil}) * s(B{\cdot}u{\cdot}w{\cdot}C') \\
\Rightarrow & \mathit{ls}(\mathrm{Head}, B{\cdot}u{\cdot}C', \mathrm{nil}) * s(B{\cdot}u{\cdot}C') \\
\Rightarrow & \exists A.\ \mathit{ls}(\mathrm{Head}, A, \mathrm{nil}) * s(A)
\end{array}$$

The proof of stability is long, but the proof steps are largely automatic. We can automate these checks [6].

Finally, we define F as $\mathit{ls}(\mathrm{Head}, A, \mathrm{x}) * \mathit{ls}(\mathrm{z}, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}B)$

Theorem 12. *The algorithm in Fig. 3 is safe and keeps the list always sorted.*

5 Semantics and soundness

Our semantics follows the abstract semantics for separation logic of Calcagno, O'Hearn and Yang [5]. Rather than presenting the semantics with respect to a particular model of the heap, we use a partial commutative cancellative[5] monoid $(M, \uplus, \emptyset)$ as an abstract notion of a heap. We use m, l, s and o to range over elements of M.

Our logic explicitly deals with the separation between a thread's own local state (l) and the shared state (s), and hence implicitly the environment's own state (o). Our semantics are given with respect to a structured heap, which separates these three components.[6] This splitting is only used to prove the soundness of the logic. There is an obvious erasure to a semantics without a splitting.

Definition 13 (Structured heaps)**.** *Heaps* $\overset{\text{def}}{=} \{(l, s, o) \mid \{l, s, o\} \subseteq M \wedge l \uplus s \uplus o$ *is defined*$\}$

Definition 14. $(l_1, s_1, o_1) \uplus (l_2, s_2, o_2)$ *defined as* (l, s, o), *iff* $s_1 = s_2 = s$, $l_1 \uplus l_2 = l$, $o_1 = l_2 \uplus o$, *and* $o_2 = l_1 \uplus o$*; otherwise it is undefined.*

We use σ to range over these structured heaps. Again following [5], we use abstract commands, A, and abstract boolean tests, b, for our abstract heap model. Note that by encoding each primitive command onto a pair of abstract commands, we can give our language a grainless semantics [21].

[5] If $m_1 \uplus m = m_2 \uplus m$, then $m_1 = m_2$.

[6] The assertions simply ignore the environment.

$$\frac{\begin{array}{c} l \uplus s = l_1 \quad b(l_1) \quad l' \uplus s' = l_2 \quad Q(s') \\ \eta \vdash (C, (l_1, \emptyset, o)) \xrightarrow{\mathbf{Emp}}{}^{*} (\mathbf{skip}, (l_2, \emptyset, o')) \end{array}}{\eta \vdash (\mathbf{atomic}_Q(b)\{C\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} (\mathbf{skip}, (l', s', o'))} \qquad \frac{\begin{array}{c} l \uplus s = l_1 \quad b(l_1) \\ \eta \vdash (C, (l_1, \emptyset, o)) \xrightarrow{\mathbf{Emp}}{}^{*} \mathbf{fault} \end{array}}{\eta \vdash (\mathbf{atomic}_Q(b)\{C\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}$$

$$\frac{\begin{array}{c} l \uplus s = l_1 \quad b(l_1) \quad l_2 \not\models Q * \mathbf{true} \\ \eta \vdash (C, (l_1, \emptyset, o)) \xrightarrow{\mathbf{Emp}}{}^{*} (\mathbf{skip}, (l_2, \emptyset, o')) \end{array}}{\eta \vdash (\mathbf{atomic}_Q(b)\{C\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}} \qquad \frac{l \uplus s = l_1 \quad b(l_1) = \mathbf{fault}}{\eta \vdash (\mathbf{atomic}_Q(b)\{C\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}$$

$$\frac{A(l, l') \quad (l', s, o) \in \mathsf{Heaps}}{\eta \vdash (A, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} (\mathbf{skip}, (l', s, o))} \qquad \frac{(\neg \exists l'.\ A(l, l'))}{\eta \vdash (A, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}$$

$$\frac{\mathcal{R}(s, s') \quad (l, s', o') \in \mathsf{Heaps}}{\eta \vdash (C, (l, s, o)) \xrightarrow[\mathbf{e}]{\mathcal{R}} (C, (l, s', o'))} \qquad \frac{}{\eta \vdash (\mathbf{skip}; C, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C, \sigma)}$$

$$\frac{}{\eta \vdash (\mathbf{while}(b)\{C\}, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (\mathbf{if}(b)\{C; \mathbf{while}(b)\{C\}\}\,\mathbf{else}\,\{\mathbf{skip}\}, \sigma)} \qquad \frac{\eta\, \mathit{proc} \ = \ C}{\eta \vdash (\mathit{proc}, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C, \sigma)}$$

$$\frac{}{\eta \vdash (\mathbf{skip} \| \mathbf{skip}, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (\mathbf{skip}, \sigma)} \qquad \frac{\eta \vdash (C, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C_1, \sigma')}{\eta \vdash (C; C', \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C_1; C', \sigma')}$$

$$\frac{\eta \vdash (C_1, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C_1', \sigma')}{\eta \vdash (C_1 \| C_2, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C_1' \| C_2, \sigma')} \qquad \frac{\eta \vdash (C_2, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C_2', \sigma')}{\eta \vdash (C_1 \| C_2, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C_1 \| C_2', \sigma')}$$

$$\frac{b(l)}{\eta \vdash (\mathbf{if}(b)\{C_1\}\,\mathbf{else}\,\{C_2\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C_1, (l, s, o))}$$

$$\frac{\neg b(l)}{\eta \vdash (\mathbf{if}(b)\{C_1\}\,\mathbf{else}\,\{C_2\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C_2, (l, s, o))} \qquad \frac{b(l) = \mathbf{fault}}{\eta \vdash (\mathbf{if}(b)\{C_1\}\,\mathbf{else}\,\{C_2\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}$$

$$\frac{\eta \vdash (C, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}{\eta \vdash (C; C_2, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}} \qquad \frac{\eta \vdash (C, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}{\eta \vdash (C \| C_2, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}} \qquad \frac{\eta \vdash (C, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}{\eta \vdash (C_2 \| C, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}$$

Figure 5: Operational semantics: Config_1 reduces to Config_2 $\quad \eta \vdash \mathit{Config}_1 \xrightarrow[\lambda]{\mathcal{R}} \mathit{Config}_2$

Definition 15. *Primitive commands A are represented by a subset of $M \times M$, satisfying: (1) If $A(l_1 \uplus l, l_2)$, then either there exists l_2' such that $A(l_1, l_2')$ and $l_2 = l \uplus l_2'$, or $\neg \exists l.\ A(l_1, l)$; and (2) If $\neg \exists l_2.\ A(l_1 \uplus l, l_2)$, then $\neg \exists l_2.\ A(l_1, l_2)$.*

Definition 16. *Boolean expressions b are represented by $M \to \{\mathbf{true}, \mathbf{false}, \mathbf{fault}\}$, satisfying: if $b(l_1 \uplus l) = v$, then either $b(l_1) = v$ or $b(l_1) = \mathbf{fault}$.*

We present the semantics of the abstract programming language in Figure 5. We define a reduction step $\eta \vdash \mathrm{Config}_1 \xrightarrow[\lambda]{\mathcal{R}} \mathrm{Config}_2$, as configuration Config_1 makes a reduction step to Config_2 with possible interference $\mathcal{R}$ and label λ in a procedure context η. The label indicates whether this is a program action, $\mathbf{p}$, or an environment action, $\mathbf{e}$. Configurations are either $\mathbf{fault}$ or a pair of a command and a structured heap, (C, σ). We use $\xrightarrow{\mathcal{R}}{}^{n}$ as the n-step reduction relation, and$\xrightarrow{\mathcal{R}}{}^{*}$ as the transitive and reflex closure of the reduction relation. A procedure context η maps procedure names to commands.

We alter the syntax of $\mathbf{atomic}$ to have a postcondition annotation Q, to specify how the state is split between shared and local on exit from the block. In CSL the resource invariant does this job, but we do not have a single resource invariant in this logic. Each of these postconditions must be precise, so there is a unique splitting.

Definition 17 (Precise assertion)**.** *P is precise iff for every $l \in M$, there exists at most one l_P such that $l_P \vDash_{\mathrm{SL}} P$ and $\exists l'.\, l_P \uplus l' = l$.*

Consider the semantics of $\mathbf{atomic}$ (Figure 5). The non-faulting rule (1) combines the thread's local state with the shared state to create a new local state, $l \uplus s = l_1$, (2) checks the guard holds of this new state, $b(l_1)$, (3) executes the command with no interference on the shared state $(\mathbf{Emp})$, (4) splits the resulting local state into a new shared and local state, $l' \uplus s' = l_2$, and (5) finally checks the postcondition Q holds of the shared state s'. As Q is precise, it uniquely specifies the splitting in step (4). There are three more rules for $\mathbf{atomic}$ where the program faults on the evaluation of the body, the evaluation of the guard, or fails to find a splitting to satisfy the postcondition.

The next three rules concern abstract commands and environment transitions. The abstract command A executes correctly, if it runs correctly by accessing only the local state. Otherwise, A faults. Its execution does not affect the shared and environment states. An environment transition can happen anytime and affects only the shared state and the environment state, provided that the shared-state change describes the rely relation, $\mathcal{R}$; the local state is unchanged.

The remaining rules deal with the standard language constructs: sequence, parallel, conditional, skip and loop. Note that our semantics has the reduction $\eta \vdash (\mathbf{skip} \| \mathbf{skip}, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (\mathbf{skip}, \sigma)$ instead of the reduction $\eta \vdash (\mathbf{skip} \| C, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} (C, \sigma)$ and its symmetric version. This simplifies stating some of the following lemmas.

We extend the standard separation logic notion of safety with a guarantee observed by each program action.

Definition 18 (Guarantee)**.**
(1) $\eta \vdash (C, \sigma, \mathcal{R})$ guars_0 $\mathcal{G}$ always holds; and
(2) $\eta \vdash (C, \sigma, \mathcal{R})$ guars_{n+1} $\mathcal{G}$ iff if $\eta \vdash (C, \sigma) \xrightarrow[\lambda]{\mathcal{R}} \mathrm{Config}$ then there exist C' σ' such that $\mathrm{Config} = (C', \sigma')$; $\eta \vdash (C', \sigma', \mathcal{R})$ guars_n $\mathcal{G}$; and if $\lambda = \mathbf{p}$ then $(\sigma, \sigma') \in \mathcal{G}$.
We define $\eta \vdash (C, \sigma, \mathcal{R})$ guars $\mathcal{G}$ as a shorthand for $\forall n. \eta \vdash (C, \sigma, \mathcal{R})$ guars_n $\mathcal{G}$.

Lemma 19 (Locality)**.**

1. *If $\eta \vdash (C, \sigma_1 \uplus \sigma') \xrightarrow{\mathcal{R}}{}^{*} (C', \sigma_2)$ then either there exists σ_2' such that $\eta \vdash (C, \sigma_1) \xrightarrow{\mathcal{R}}{}^{*} (C', \sigma_2')$ and $\sigma_2' \uplus \sigma' = \sigma_2$, or $\eta \vdash (C, \sigma_1) \xrightarrow{\mathcal{R}}{}^{*} \mathbf{fault}$; and*
2. *If $\eta \vdash (C, \sigma_1 \uplus \sigma') \xrightarrow{\mathcal{R}}{}^{*} \mathbf{fault}$ then $\eta \vdash (C, \sigma_1) \xrightarrow{\mathcal{R}}{}^{*} \mathbf{fault}$.*

Proof. Proved in Coq. □

We use the following two lemmas about relies and guarantees.

Lemma 20.
1. *If* $\eta \vdash (C, \sigma, \mathcal{R})$ guars $\mathcal{G}$*, then* $\eta \vdash (C, \sigma \uplus \sigma', \mathcal{R})$ guars $\mathcal{G}'$
2. *If* $\eta \vdash (C, \sigma, \mathcal{R})$ guars $\mathcal{G}$ *and* $\mathcal{G} \subset \mathcal{G}'$*, then* $\eta \vdash (C, \sigma, \mathcal{R})$ guars $\mathcal{G}'$

Proof. Proved in Coq. □

Lemma 21. *If* $\eta \vdash (C, \sigma) \xrightarrow{\mathcal{R}}{}^{*} (C', \sigma')$ *and* $\mathcal{R} \subset \mathcal{R}'$*, then* $\eta \vdash (C, \sigma) \xrightarrow{\mathcal{R}'}{}^{*} (C', \sigma')$

Proof. Proved in Coq. □

We use the following two lemmas about the operational semantics to simplify proofs.

Lemma 22. $\eta \vdash (C, \sigma) \xrightarrow{\mathcal{R}}{}^{*} (C', \sigma')$ *iff* $\eta \vdash (C \| \mathbf{skip}, \sigma) \xrightarrow{\mathcal{R}}{}^{*} (C' \| \mathbf{skip}, \sigma')$.

Proof. Proved in Coq. □

Lemma 23.
1. $\eta \vdash (C, \sigma) \xrightarrow{\mathcal{R}}{}^{*} (C, \sigma')$ *iff* $\eta \vdash (C, \sigma) \xrightarrow[\mathbf{e}]{\mathcal{R}} (C, \sigma')$
2. $\eta \vdash (C, \sigma) \xrightarrow{\mathcal{R}}{}^{*} \mathbf{fault}$ *iff* $\eta \vdash (C, \sigma) \xrightarrow[\mathbf{e}]{\mathcal{R}} (C, \sigma'') \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}$
3. $\eta \vdash (C, \sigma) \xrightarrow{\mathcal{R}}{}^{*} (\mathbf{skip}, \sigma')$ *iff* $\eta \vdash (C, \sigma) \xrightarrow[\mathbf{e}]{\mathcal{R}} (C, \sigma'') \xrightarrow[\mathbf{p}]{\mathcal{R}} (\mathbf{skip}, \sigma''') \xrightarrow[\mathbf{e}]{\mathcal{R}} (\mathbf{skip}, \sigma')$

where $C = \mathbf{atomic}_Q(b)\{C'\}$ *or* $C = A$

Proof. Proved in Coq. □

To prove the soundness of the parallel composition rule, we require the following: (1) if we have the guarantee of two commands, C_1 and C_2, then we have the guarantee of their parallel composition; and (2) if the parallel composition of two commands can make a reduction, then the two commands can also make that reduction given an extended rely condition.

Lemma 24. *If* $\eta \vdash (C_1, \sigma_1, (\mathcal{R} \cup \mathcal{G}_2))$ guars $\mathcal{G}_1$*,* $\eta \vdash (C_2, \sigma_2, (\mathcal{R} \cup \mathcal{G}_1))$ guars $\mathcal{G}_2$ *and* $\sigma_1 \uplus \sigma_2 = \sigma$ *then* $\eta \vdash (C_1 \| C_2, \sigma, \mathcal{R})$ guars $\mathcal{G}_1 \cup \mathcal{G}_2$

Proof. Proved in Coq. □

Lemma 25. *If* $\eta \vdash (C_1, \sigma_1, (\mathcal{R} \cup \mathcal{G}_2))$ guars $\mathcal{G}_1$*,* $\eta \vdash (C_2, \sigma_2, (\mathcal{R} \cup \mathcal{G}_1))$ guars $\mathcal{G}_2$*,* $\sigma_1 \uplus \sigma_2 = \sigma$ *and* $\eta \vdash (C_1 \| C_2, \sigma) \xrightarrow{\mathcal{R}}{}^{*} (C_1' \| C_2', \sigma')$ *then there exists* σ_1' *and* σ_2' *such that* $\eta \vdash (C_1, \sigma_1) \xrightarrow{\mathcal{R} \cup \mathcal{G}_2}{}^{*} (C_1', \sigma_1')$*,* $\eta \vdash (C_2, \sigma_2) \xrightarrow{\mathcal{R} \cup \mathcal{G}_1}{}^{*} (C_2', \sigma_2')$ *and* $\sigma_1' \uplus \sigma_2' = \sigma'$*.*

Proof. Proved in Coq. □

The following two lemmas are used in the soundness of the sequencing rule.

Lemma 26. *If* $\eta \vdash (C_1, \sigma, \mathcal{R})$ guars $\mathcal{G}$ *and for all* σ' *such that* $\eta \vdash (C_1, \sigma) \xrightarrow{\mathcal{R}}{}^{*} (\mathbf{skip}, \sigma') \Rightarrow (C_2, \sigma', \mathcal{R})$ guars $\mathcal{G}$*, then* $\eta \vdash (C_1; C_2, \sigma, \mathcal{R})$ guars_n $\mathcal{G}$

Proof. Proved in Coq. □

Lemma 27. *If* $\eta \vdash (C_1; C_2, \sigma) \xrightarrow{\mathcal{R}}{}^{n} (\mathbf{skip}, \sigma'')$ *then* $\eta \vdash (C_1, \sigma) \xrightarrow{\mathcal{R}}{}^{*} (\mathbf{skip}, \sigma')$ *and* $\eta \vdash (C_2, \sigma') \xrightarrow{\mathcal{R}}{}^{*} (\mathbf{skip}, \sigma'')$.

Proof. Proved in Coq. □

We are now in a position to state and prove the soundness of this logic.

Definition 28. $\eta \models_n C\ \mathbf{sat}\ (p, R, G, q)$ *iff for all* $R' \subseteq R$ *and* $\sigma \vDash_{R'} p$*, then*

1. $\eta \vdash (C, \sigma, [\![R']\!])$ guars $[\![G]\!]$*; and*
2. *if* $\eta \vdash (C, \sigma) \xrightarrow{[\![R]\!]'}{}^{n} (\mathbf{skip}, \sigma')$ *then* $\sigma' \vDash_{[\![R]\!]'} q$*.*

Definition 29. $\Gamma \models_n \eta$ *iff* $\forall (proc\ \mathbf{sat}\ (p, R, G, q)) \in \Gamma$ *then* $\eta \models_n (\eta(proc))\ \mathbf{sat}\ (p, R, G, q)$

Definition 30. $\Gamma \models_n C\ \mathbf{sat}\ (p, R, G, q)$ *iff forall* η *if* $\Gamma \models_n \eta$ *then* $\eta \models_{n+1} C\ \mathbf{sat}\ (p, R, G, q)$

Theorem 31 (Soundness)**.** *If* $\Gamma \vdash C\ \mathbf{sat}\ (p, R, G, q)$*, then* $\forall n. \Gamma \models_n C\ \mathbf{sat}\ (p, R, G, q)$

Proof. By induction on the proof rules. Let $[\![R]\!] = \mathcal{R}$.

- Atomic command:

  $$\frac{\begin{array}{c} \vDash_{\mathrm{SL}} P_1 * P_2 \Longrightarrow (b)\ \mathrm{def} \\ \vdash C\ \mathbf{sat}\ ((P_1 * P_2) \wedge b, \{\}, \{\}, \exists \overline{y}.\ Q_1 * Q_2) \\ \boxed{Q}\ \mathsf{stable\ under}\ R \qquad (P_1 \rightsquigarrow Q_1) \subseteq G \\ \overline{y} \cap FV(P_2) = \emptyset \qquad \vDash_{\mathrm{SL}} P \Rightarrow P_1 * F \qquad \vDash_{\mathrm{SL}} Q_1 * F \Rightarrow Q \end{array}}{\vdash (\mathbf{atomic}_Q(b)\{C\})\ \mathbf{sat}\ (\boxed{\exists \overline{y}.\ P} * P_2, R, G, \exists \overline{y}.\ \boxed{Q} * Q_2)}$$

  It suffices to consider three possible reduction sequences (Lemma 23). We can ignore the environment actions due to Lemma 5. Hence we can assume $(\sigma, i) \vDash_R \boxed{P} * P_2$ and $(\mathbf{atomic}_Q(b)\{C\}, \sigma) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathrm{Config}$ and prove there exists σ' st $\mathrm{Config} = (\mathbf{skip}, \sigma')$, $(\sigma, \sigma') \in [\![G]\!]$ and $(\sigma', i) \vDash_R \boxed{Q} * Q_2$. Let $(l, s, o) = \sigma$ and $l_1 = l * s$. Note that, if it cannot reduce then it holds trivially.

  Case: $l_1, i \vDash_{\mathrm{SL}} b$ does not hold. As ($l_1, i \vDash_{\mathrm{SL}} (b)$ def), there are no reduction rules that apply. Hence it holds trivially.

  Case: $l_1, i \vDash_{\mathrm{SL}} b$ holds. Therefore, we know $l_1, i' \vDash_{\mathrm{SL}} (P * P_2) \wedge B$ where $\exists \overline{v}. i' = i[\overline{y} \mapsto \overline{v}]$.

  Hence, by assumption we know $l_1, i' \vDash_{\mathrm{SL}} (P_1 * F * P_2) \wedge b$, and as b defined by $P_1 * P_2$, we get $l_1, i' \vDash_{\mathrm{SL}} ((P_1 * P_2) \wedge b) * F$.

  So $l_1', i' \vDash_{\mathrm{SL}} (P_1 * P_2) \wedge b$ and $s_1, i' \vDash_{\mathrm{SL}} F$ where $l_1' \uplus s_1 = l_1$. By assumption we have:

  $$(C, (l_1', \emptyset, o), \{\})\ \mathsf{guars}\ \{\}$$

  $$\begin{array}{c} \forall l_2.\ (C, (l_1', \emptyset, o)) \xrightarrow{\{\}}{}^{*} (\mathbf{skip}, (l_2', \emptyset, o')) \\ \Rightarrow (l_2', \emptyset, o') \vDash_R Q_1 * Q_2 \end{array}$$

  By Lemma 19 and 20, we know

  $$(C, (l_1, \emptyset, o), \{\})\ \mathsf{guars}\ \{\} \tag{1}$$

  $$\begin{array}{c} \forall l_2.\ (C, (l_1, \emptyset, o)) \xrightarrow{\{\}}{}^{*} (\mathbf{skip}, (l_2, \emptyset, o')) \\ \Rightarrow (l_2, \emptyset, o') \vDash_R Q_1 * Q_2 * F \wedge l_2 = l_2' \uplus s_1 \end{array} \tag{2}$$

  We now proceed by case analysis on the possible reduction rule:

  1.

$$\dfrac{l \uplus s = l_1 \quad b(l_1) = \mathbf{fault}}{\eta \vdash (\mathbf{atomic}_Q(b)\{C\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}$$

We know $l_1, i' \models_{\mathrm{SL}} P * P_2$, therefore $b(l_1) \neq \mathbf{fault}$, so this rule does not apply.

2.

$$\dfrac{\begin{array}{c} l \uplus s = l_1 \quad b(l_1) \\ \eta \vdash (C, (l_1, \emptyset, o)) \xrightarrow{\mathbf{Emp}}{}^{*} \mathbf{fault} \end{array}}{\eta \vdash (\mathbf{atomic}_Q(b)\{C\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}$$

From (1) we know the body cannot fault, so this rule does not apply.

3.

$$\dfrac{\begin{array}{c} l \uplus s = l_1 \quad b(l_1) \quad l_2 \not\models Q * \mathbf{true} \\ \eta \vdash (C, (l_1, \emptyset, o)) \xrightarrow{\mathbf{Emp}}{}^{*} (\mathbf{skip}, (l_2, \emptyset, o')) \end{array}}{\eta \vdash (\mathbf{atomic}_Q(b)\{C\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathbf{fault}}$$

We know that $l_2, i' \models_{\mathrm{SL}} Q * Q_2$ and hence $l_2 \models_{\mathrm{SL}} Q * \mathbf{true}$. Therefore this rule does not apply.

4.

$$\dfrac{\begin{array}{c} l \uplus s = l_1 \quad b(l_1) \quad l' \uplus s' = l_2 \quad Q(s') \\ \eta \vdash (C, (l_1, \emptyset, o)) \xrightarrow{\mathbf{Emp}}{}^{*} (\mathbf{skip}, (l_2, \emptyset, o')) \end{array}}{\eta \vdash (\mathbf{atomic}_Q(b)\{C\}, (l, s, o)) \xrightarrow[\mathbf{p}]{\mathcal{R}} (\mathbf{skip}, (l', s', o'))}$$

We know that $l_2, i' \models_{\mathrm{SL}} Q_1 * F * Q_2$ and $l_2' \uplus s_1 = l_2$.
By assumption we know $l_2, i' \models_{\mathrm{SL}} Q * Q_2$.
As Q is precise we know $l' \models_{\mathrm{SL}} Q_2$.
As Q is stable under R and Lemma 7, we know $(\emptyset, s', o'), i' \models_R \boxed{Q}$.
Thus, $(l', s', o'), i' \models_R \boxed{Q} * Q_2$, and therefore $(l', s', o'), i \models_R \exists \overline{y}.\ \boxed{Q} * Q_2$.

We know $l_2', i' \models_{\mathrm{SL}} Q_1 * Q_2$. Therefore there exist s'' and l_3 such that $s'' \uplus l_3 = l_2'$ and $s'', i' \models_{\mathrm{SL}} Q_1$ and $l_3, i' \models_{\mathrm{SL}} Q_2$. Therefore $s'' \uplus s_1, i' \models_{\mathrm{SL}} Q_1 * F$ and hence $s'' \uplus s_1, i' \models_{\mathrm{SL}} Q$. As Q is precise, $l' \uplus s' = l_1$ and $s'' \uplus s_1 \uplus l_3 = l_1$, we know $s_1 \uplus s'' = s'$. Hence the step $(s_1' \uplus s_1, s'' \uplus s_1)$ is in $[\![P_1 \rightsquigarrow Q_1]\!]$, and hence in G as required.

- Sequential composition: Follows from Lemmas 26 and 27.
- Parallel composition: Follows from Lemmas 24 and 25, and using the stability assumptions with Lemma 7.
- Skip: Trivial
- Basic action:

$$\dfrac{\mathbf{sat}_{\ SL}\{P\}A\{Q\}}{A\ \mathbf{sat}\ (P, \mathcal{R}, \mathcal{G}, Q)}$$

It suffices to consider three possible reduction sequences (Lemma 23). Assume $\sigma \models_R P$, and prove

1. $(A, \sigma, \mathcal{R})$ guars $\mathcal{G}$; and
2. if $(A, \sigma) \xrightarrow{\mathcal{R}}{}^{*} (\mathbf{skip}, \sigma')$ then $\sigma' \models_R Q$.

To prove (1), assume $(A, \sigma) \xrightarrow[\mathbf{e}]{\mathcal{R}} (A, \sigma') \xrightarrow[\mathbf{p}]{\mathcal{R}} \mathrm{Config}$ and prove exists σ'' st $\mathrm{Config} = (\mathbf{skip}, \sigma'')$ and $(\sigma', \sigma'') \in \mathcal{G}$. Let $(l, s, e) = \sigma$ and $(l', s', e') = \sigma'$, by first reduction, we know $l = l'$, and as P only depends on local state, then $\sigma' \vDash_R P$. Therefore Config is not a fault, and hence $\sigma'' = (l''', s', e')$, so $(\sigma', \sigma'') \in \mathcal{G}$.

To prove (2), assume $(A, \sigma) \xrightarrow[\mathbf{e}]{\mathcal{R}} (A, \sigma') \xrightarrow[\mathbf{p}]{\mathcal{R}} (\mathbf{skip}, \sigma'') \xrightarrow[\mathbf{e}]{\mathcal{R}} (\mathbf{skip}, \sigma''')$. By construction, we know $\sigma'' \vDash_R Q$. Let $(l'', s'', e'') = \sigma''$ and $(l''', s''', e''') = \sigma'''$, hence $l'' = l'''$ therefore $\sigma''' \vDash_R Q$ as required.

- Frame: As C and $\mathbf{skip}\|C$ are equivalent with respect to the operational semantics by Lemma 22. We can derive the frame rule from the parallel rule. If C contains critical regions:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vDash C \ \mathbf{sat}\ (p, R, G, q)}{\vDash C \ \mathbf{sat}\ (\mathsf{wssa}_{\mathcal{R}}(p), R, G, q)} \quad L \text{ stable under } R \cup G \quad \dfrac{}{\vDash \mathbf{skip}\ \mathbf{sat}\ (L, R \cup G, \{\}, L)}}{\vDash (C \parallel \mathbf{skip})\ \mathbf{sat}\ (\mathsf{wssa}_{\mathcal{R}}(p) * L, R, G, q * L)}}{\vDash (C \parallel \mathbf{skip})\ \mathbf{sat}\ (p * L, R, G, q * L)}}{\vDash C \ \mathbf{sat}\ (p * L, R, G, q * L)}$$

and C does not contain critical regions:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vDash C \ \mathbf{sat}\ (p, R, \{\}, q)}{\vDash C \ \mathbf{sat}\ (\mathsf{wssa}_{\mathcal{R}}(p), R, \{\}, q)} \quad \dfrac{\dfrac{}{\vDash \mathbf{skip}\ \mathbf{sat}\ (L, R, \{\}, L)}}{\vDash \mathbf{skip}\ \mathbf{sat}\ (\mathsf{wssa}_{\mathcal{R}}(L), R, \{\}, L)}}{\vDash (C \parallel \mathbf{skip})\ \mathbf{sat}\ (\mathsf{wssa}_{\mathcal{R}}(p) * \mathsf{wssa}_{\mathcal{R}}(L), R, G, q * L)}}{\vDash (C \parallel \mathbf{skip})\ \mathbf{sat}\ (p * L, R, G, q * L)}}{\vDash C \ \mathbf{sat}\ (p * L, R, G, q * L)}$$

where

  - $\mathsf{wssa}_{\mathcal{R}}(\boxed{P}) \stackrel{\mathbf{def}}{=} \boxed{\mathsf{wssa}_{\mathcal{R}}(P)}$,
  - $\mathsf{wssa}_{\mathcal{R}}(P) \stackrel{\mathbf{def}}{=} P$,
  - $\mathsf{wssa}_{\mathcal{R}}(p \ op\ q) \stackrel{\mathbf{def}}{=} \mathsf{wssa}_{\mathcal{R}}(p) \ op \ \mathsf{wssa}_{\mathcal{R}}(q)$, and
  - $\mathsf{wssa}_{\mathcal{R}}(qu\, x.\ p) \stackrel{\mathbf{def}}{=} qu\, x.\ \mathsf{wssa}_{\mathcal{R}}(p)$;

  and hence $\vDash_R \mathsf{wssa}_{[\![R]\!]}(p) \Leftrightarrow p$; and $\mathsf{wssa}_{[\![R]\!]}(p)$ stable under R.
- Consequence: Follows from Lemmas 21 and 20.
- While rule: Follows directly from induction on number of reduction steps.
- Procedure rule: Follows directly from induction on number of reduction steps.

□

We must prove the read axiom separately, as it depends on particular model of separation logic. We can view a non-atomic read as two atomic reads, which fault if the they read different values: local temp; **atomic**{x=[e]}; **atomic**{temp=[e]}; **if** (x!=temp) fault . As

$$\boxed{e \mapsto e' * P} \Leftrightarrow \boxed{\mathsf{wssa}_{\mathcal{R}}(e \mapsto e' * P)} \qquad\qquad \mathsf{wssa}_{\mathcal{R}}(e \mapsto e' * P) \Rightarrow e \mapsto e' * P$$

We can derive it as follows:

$\{\boxed{e \mapsto e' * P}\}$
$\{\boxed{\mathsf{wssa}_{\mathcal{R}}(e \mapsto e' * P)}\}$
atomic{x=[e]};
$\{\boxed{\mathsf{wssa}_{\mathcal{R}}(e \mapsto e' * P)} \wedge x = e'\}$

atomic{temp=[e]};
$\left\{\boxed{\mathrm{wssa}_{\mathcal{R}}(e \mapsto e' * P)} \land x = e' \land temp = e'\right\}$
if (x!=temp) fault
$\left\{\boxed{\mathrm{wssa}_{\mathcal{R}}(e \mapsto e' * P)} \land x = e'\right\}$
$\left\{\boxed{e \mapsto e' * P} \land x = e'\right\}$

6 Late stability checks

The rules presented so far check stability at the forks of parallel composition and on the exits of atomic blocks.

We can provide a similar semantics where we delay the stability checks to the entry of atomics and joins of parallel compositions. This semantics uses the strongest stable weaker assertion.

Definition 32 (Strongest stable weaker assertion)**.**

- *$q \Rightarrow sswa_{\mathcal{R}}(q)$;*
- *$sswa_R(q); \mathcal{R} \Longrightarrow sswa_R(q)$; and*
- *for all p, if $p; \mathcal{R} \Longrightarrow \mathcal{R}$ and $q \Rightarrow p$, then $sswa_{\mathcal{R}}(q) \Rightarrow p$.*

We use this to define the semantics of shared assertions as:

$$l, s, i \vDash_R \boxed{P} \iff l = \emptyset \land s, i \vDash_{\mathrm{SL}} \mathrm{sswa}_{[\![R]\!]}(P)$$

This definition also ensures that assertions are semantically resistant to interference.

Lemma 33. *If $(l, s, i \vDash_R p)$, $s' \uplus l$ defined and $[\![R]\!](s, s')$ then $(l, s', i \vDash_R p)$.*

However, this semantics reverses the direction of Lemma 7:

Lemma 34. *If $(l, s, i \vDash_R p)$, $[\![R]\!] \supseteq [\![R']\!]$ and p stable under R' then $(l, s, i \vDash_{R'} p)$. Note that $(l, s, i \vDash_R p)$ and $[\![R']\!] \supseteq [\![R]\!]$ then $(l, s, i \vDash_{R'} p)$.*

7 Multiple regions

Concurrent separation logic is defined with multiple resource names for critical regions. We can trivially extend our treatment to this setting by:

1. associating a region name to each boxed/shared assertion, i.e. $\boxed{P}_r$;
2. instead of a single R and a single G, having a set of rely/guarantee pairs index by resource name, namely $\mathcal{I} = r_1{:}(R_1, G_1), \ldots, r_n{:}(R_n, G_n)$; and
3. annotating each atomic with the relevant set of region names.

Here are the changes in more detail:

Each boxed assertion is now subscripted with the name of the region it describes. The shared state, s, of our model is a function from region names to separation logic states, such that all the states in its range are disjoint. The meaning of a boxed assertion is now as follows:

$$l, s \vDash_R \boxed{P}_r \overset{\mathbf{def}}{=} dom(l) = \emptyset \ \land \ s(r) \vDash_{\mathrm{SL}} P.$$

It asserts that the local state is empty and that the relevant region of the shared state satisfies P. The other regions of the shared state could be anything.

Judgements now have the form $\vdash C\ \mathbf{sat}\ (p, \mathcal{I}, q)$, where $\mathcal{I}$ is a mapping from resource names r_i to rely/guarantee pairs (R_i, G_i). We shall use the $\mathcal{I}_R(r_i)$ (resp. $\mathcal{I}_R(r_i)$) notation for accessing the rely (resp. guarantee) component of $\mathcal{I}(r_i)$.

Here is a simple rule for atomic blocks that access a single shared region r:

$$\frac{\begin{array}{c}\vdash C\ \mathbf{sat}\ (P' * P'', \{\}, Q' * Q'') \\ \boxed{P}\ \text{stable under}\ \mathcal{I}_R(r) \qquad \boxed{Q}\ \text{stable under}\ \mathcal{I}_R(r) \\ P' \rightsquigarrow Q' \Longrightarrow P \rightsquigarrow Q \qquad (P \rightsquigarrow Q) \subseteq \mathcal{I}_G(r)\end{array}}{\vdash (\mathbf{atomic}\{C\})\ \mathbf{sat}\ (\boxed{P}_r * P'', \mathcal{I}, \boxed{Q}_r * Q'')}$$

We can extend this rule to atomic blocks that access multiple shared regions as follows:

$$\frac{\begin{array}{c}\vdash C\ \mathbf{sat}\ (P'_1 * \ldots * P'_n * P'', \{\}, Q'_1 * \ldots * Q'_n * Q'') \\ \forall i \in \{1, \ldots, n\} \begin{pmatrix} \boxed{P_i}\ \text{stable under}\ \mathcal{I}_R(r_i) \qquad \boxed{Q_i}\ \text{stable under}\ \mathcal{I}_R(r_i) \\ P'_i \rightsquigarrow Q'_i \Longrightarrow P_i \rightsquigarrow Q_i \qquad (P_i \rightsquigarrow Q_i) \subseteq \mathcal{I}_G(r_i)\end{pmatrix}\end{array}}{\vdash (\mathbf{atomic}\{C\})\ \mathbf{sat}\ (\boxed{P_1}_{r_1} * \ldots * \boxed{P_n}_{r_n} * P'', \mathcal{I}, \boxed{Q_1}_{r_1} * \ldots * \boxed{Q_n}_{r_n} * Q'')}$$

As in CSL, we can create a statically scoped shared region of state,

$$\frac{\vdash C\ \mathbf{sat}\ (\boxed{P}_r * p, \mathcal{I} \uplus \{r{:}(R, G)\}, \boxed{Q}_r * q) \qquad r \notin \mathit{dom}(\mathcal{I}) \qquad p, q\ \text{contain no}\ \boxed{\_}_r}{\vdash C\ \mathbf{sat}\ (P * p, \mathcal{I}, Q * q)}$$

Our semantics are almost unchanged to before, just with more structure to represent the multiple shared regions of memory.

Definition 35 (Structured heap)**.** *For a set of resource names $\mathbb{R}$, we define a structured heap, σ, as a triple, (l, s, e), where $l, e : M$ and $s : (\mathbb{R} \rightharpoonup M)$, such that $l \uplus e \uplus \circledast_{r \in \mathbb{R}} s(r)$ is defined.*

Joining two structured heaps is defined exactly as it was defined in Section 5 for a single shared region:

Definition 36. *$(l_1, s_1, e_1) \uplus (l_2, s_2, e_2)$ defined as (l, s, e), iff $s_1 = s_2 = s$, $l_1 \uplus l_2 = l$, $e_1 = l_2 \uplus e$, and $e_2 = l_1 \uplus e$; otherwise it is undefined.*

We can trivially extend the operational semantics to this new form of state. The lemmas and the rest of the definitions are unchanged.

8 Local guards

The local state of a thread has a more subtle rôle in controlling interference than we have seen so far. It acts as a token, a permission to perform a certain action, and as a guard, a prohibition that the environment does some action.

In the operational semantics, an environment transition (Figure 5, last rule) requires that the resulting state is well formed, that the new shared state is disjoint from the local state. In essence, the existence of the local state *restricts* what the environment can do (e.g. it cannot allocate an existing memory address).

Besides its prohibitive rôle as to what other threads can do, the local state has a permissive rôle. Its presence allows a thread to do more actions than it would otherwise be able to do (e.g. in some algorithms, a mutex can be unlocked only by the thread that locked it).

So far, our proof rules have ignored these two rôles of the local state that are present in the semantics. We can, however, extend our assertion language with guarded boxed assertions $\boxed{L|S}$, where L is an assertion about the local state, whose presence is used in the stability proof of S. Similarly, guarded actions $G \mid P \leadsto Q$ use the guard G to stand for the local state that must be owned for the action to occur.

Definition 37. *The assertion* $\boxed{L|S}$ *is stable under interference from* $G \mid P \leadsto Q$*, if and only if,*

$$((P -\!\circledast S) * Q \Rightarrow S) \vee \neg(P * G * L) \vee \neg(Q * L) \vee \neg(S * L)$$

The three new cases are when the action cannot execute. The local state that protects the stability of a shared assertion cannot be accessed directly, because the shared assertion might become unstable. We must each time redo a stability check.

9 Related work

Owicki & Gries [17] introduced the concept of non-interference between the proofs of parallel threads. Their method is not compositional and does not permit top-down development of a proof because the final check of interference-freedom may fail rendering the whole development useless.

To address this problem, Jones [12] introduced the compositional rely/guarantee method. In the VDM-style, Jones opted for ‘two-state’ postconditions; other authors [23, 19] have chosen single-state postconditions. Several authors have proved the soundness and relative completeness of rely/guarantee [23, 19, 7]; Prensa’s proof [19] is machine checked by the Isabelle theorem prover. The completeness results are all modulo the introduction of auxiliary variables. Abadi and Lamport [1] have adapted RG to temporal logic and have shown its soundness for safety specifications.

Separation logic [20, 16] takes a different approach to interference by forbidding it except in critical regions [11]. An invariant, I, is used to describe the shared state. This is a simple case of our system where the interference specifications (i.e. R and G) are restricted to a very simple relation, $I \leadsto I$. Brookes has shown concurrent separation logic to be sound [3].

There have been attempts to verify fine-grained concurrent algorithms using both separation logic and rely/guarantee. Vafeiadis *et al.* [22] verify several list algorithms using rely/guarantee. Their proofs require reachability predicates to describe lists and they cannot deal with the disposal of nodes. Parkinson *et al.* [18] verify a non-blocking stack algorithm using concurrent separation logic. Their proof requires a lot of auxiliary state to encode the possible interference. With the logic presented in this paper much of the auxiliary state can be removed, and hence the proof becomes clearer.

Concurrently with our work, Feng, Ferreira and Shao [8] proposed a different combination of rely/guarantee and separation logic, SAGL. Both our approach and SAGL partition memory into shared and private parts. However, in SAGL, every primitive command is assumed to be atomic. Our approach is more flexible and allows one to specify what is atomic; everything else is considered non-atomic. By default, non-atomic commands cannot update shared state, so we only need stability checks when there is an atomic command: in the lock coupling list only at the lock and unlock operations. On the other hand, SAGL must check stability after every single command. Moreover, in SAGL, the rely and guarantee conditions are relations and stability checks are semantic implications. We instead provide convenient syntax for writing down these relations, and reduces the semantic implication into a simple logic implication. This allowed us to automated our logic [6], and hence automatically verify the safety of a collection of fine-grained list algorithms.

SAGL is presented as a logic for assembly code, and is thus hard to apply at different abstraction levels. It does not contain separation logic as a proper subsystem, as it lacks the standard version of the frame rule [20]. This means that it cannot prove the usual separation logic specification of procedures such as `copy_tree` [15]. In contrast, our system subsumes SL [20], as well as the single-resource variant of CSL [16]: hence, the same proofs there (for a single resource) go through directly in our system. Of course, the real interest is the treatment of additional examples, such as lock coupling, that neither separation logic nor rely/guarantee can prove tractably. Our system also includes a rely-guarantee system, which is why we claim to have produced a marriage of the two approaches. It may be possible to extend SAGL to include the frame rule for procedures, but we understand that such extension is by no means obvious.

With this all being said, there are remarkable similarities between our work and SAGL; that they were arrived at independently is perhaps encouraging as to the naturalness of the basic ideas.

10 Conclusion

We have presented a marriage of rely/guarantee with separation logic. We proved soundness with respect to an abstract operational semantics in the style of abstract separation logic [5]. Hence, our proof can be reused with different languages and with different separation logics, e.g. permissions and variables as resource [2]. Our logic allows us to give a clear and simple proof of the lock-coupling list algorithm, which includes memory disposal. Moreover, our logic can be efficiently automated [6].

Acknowledgements We thank Josh Berdine, Richard Bornat, Cristiano Calcagno, Joey Coleman, Tony Hoare, Cliff Jones, Xinyu Feng, Alan Mycroft, Peter O'Hearn, Uday Reddy, John Reynolds and Hongseok Yang for discussions and feedback on this work. We acknowledge funding from an RAEng/EPSRC fellowship (Parkinson) and a Gates scholarship (Vafeiadis).

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Syst.*, 17(3):507–534, May 1995.

[2] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. *ENTCS*, 155:247–276, 2006.

[3] S. D. Brookes. A semantics for concurrent separation logic. In *CONCUR*, volume 3170 of *LNCS*, pages 16–34. Springer, 2004.

[4] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In *POPL*, pages 123–134. ACM Press, 2007.

[5] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. to appear in LICS, 2007.

[6] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*. LNCS, 2007.

[7] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. Technical Report CS-TR-987, Newcastle University, Oct. 2006.

[8] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proceedings of ESOP*, 2007.

[9] J.-Y. Girard. On the unity of logic. *Ann. Pure Appl. Logic*, 59(3):201–217, 1993.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[11] C. A. R. Hoare. Towards a theory of parallel programming. *Operating Systems Techniques*, 1971.

[12] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[13] C. B. Jones. *Wanted: a compositional approach to concurrency*, pages 5–15. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[14] C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[15] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, pages 1–19, 2001.

[16] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004.

[17] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[18] M. J. Parkinson, R. Bornat, and P. W. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, 2007.

[19] L. Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In *ESOP*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003.

[20] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[21] J. C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In *FSTTCS*, pages 35–48, 2004.

[22] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*. ACM Press, 2006.

[23] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

A Proof outlines

```
locate(e) { local p,c,t;
$\{\boxed{\exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) * s(A)} \wedge -\infty < \mathrm{e}\}$
  p = Head;
$\{\boxed{\exists ZB.\ ls(\mathrm{Head}, \{\}, \mathrm{p}) * N(\mathrm{p}, -\infty, Z) * ls(Z, B, \mathrm{nil}) * s(-\infty{\cdot}B)} \wedge -\infty < \mathrm{e}\}$
  lock(p);
$\{\exists Z.\ \boxed{\exists B.\ ls(\mathrm{Head}, \{\}, \mathrm{p}) * L_{\mathrm{tid}}(\mathrm{p}, -\infty, Z) * ls(Z, B, \mathrm{nil}) * s(-\infty{\cdot}B)} * (\mathrm{p}{+}2 \mapsto Z) \wedge -\infty < \mathrm{e}\}$
  c = p.next;
  t = c.value;
  while (t < e) {
  $\left\{\exists u.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\mathrm{Head}, A, \mathrm{p}) * L_{\mathrm{tid}}(\mathrm{p}, u, \mathrm{c}) \\ * N(\mathrm{c}, \mathrm{t}, Z) * ls(\mathrm{c}, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\mathrm{t}{\cdot}B)\end{array}} * (\mathrm{p}{+}2 \mapsto \mathrm{c}) \wedge u < \mathrm{e} \wedge \mathrm{t} < \mathrm{e}\right\}$
    lock(c);
    $\left\{\exists uZ.\ \boxed{\begin{array}{l}\exists AB.\ ls(\mathrm{Head}, A, \mathrm{p}) * L_{\mathrm{tid}}(\mathrm{p}, u, \mathrm{c}) \\ * L_{\mathrm{tid}}(\mathrm{c}, \mathrm{t}, Z) * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\mathrm{t}{\cdot}B)\end{array}} * (\mathrm{p}{+}2 \mapsto \mathrm{c}) * (\mathrm{c}{+}2 \mapsto Z) \wedge \mathrm{t} < \mathrm{e}\right\}$
    unlock(p);
    $\left\{\exists Z.\ \boxed{\begin{array}{r}\exists AB.\ ls(\mathrm{Head}, A, \mathrm{c}) * L_{\mathrm{tid}}(\mathrm{c}, \mathrm{t}, Z) \\ * ls(Z, B, \mathrm{nil}) * s(A{\cdot}\mathrm{t}{\cdot}B)\end{array}} * (\mathrm{c}{+}2 \mapsto Z) \wedge \mathrm{t} < \mathrm{e}\right\}$
    p = c;
    c = p.next;
    t = c.value;
    $\left\{\exists u.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\mathrm{Head}, A, \mathrm{p}) * L_{\mathrm{tid}}(\mathrm{p}, u, \mathrm{c}) \\ * N(\mathrm{c}, \mathrm{t}, Z) * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\mathrm{t}{\cdot}B)\end{array}} * (\mathrm{p}{+}2 \mapsto \mathrm{c}) \wedge u < \mathrm{e}\right\}$
  }
$\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\mathrm{Head}, A, \mathrm{p}) * L_{\mathrm{tid}}(\mathrm{p}, u, \mathrm{c}) \\ * N(\mathrm{c}, v, Z) * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)\end{array}} * (\mathrm{p}{+}2 \mapsto \mathrm{c}) \wedge u < \mathrm{e} \wedge \mathrm{e} \leq v\right\}$
  return (p,c);
}

add(e) { local x,y,z,t;
$\{\boxed{\exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) \wedge s(A)} \wedge -\infty < \mathrm{e}\}$
  (x,z) = locate(e);
$\left\{\begin{array}{l}\exists uv.\ \boxed{\exists ZAB.\ ls(\mathrm{Head}, A, \mathrm{x}) * L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{z}) * N(\mathrm{z}, v, Z) * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)} \\ * (\mathrm{x}{+}2 \mapsto \mathrm{z}) \wedge u < \mathrm{e} \wedge \mathrm{e} \leq v\end{array}\right\}$
  t = z.value; if (t != e) {
    y = cons(0,e,z);
    x.next = y;
$\left\{\exists uv.\ \boxed{\begin{array}{r}\exists ZAB.\ ls(\mathrm{Head}, A, \mathrm{x}) * L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{z}) \\ * ls(\mathrm{z}, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\mathrm{e}{\cdot}B)\end{array}} * (\mathrm{x}{+}2 \mapsto \mathrm{y}) * U(\mathrm{y}, \mathrm{e}, \mathrm{z})\right\}$
```

```
  }
  unlock(x);
$\{\exists v.\ \boxed{\exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) * s(A)}\}$
}

remove(e) { local x,y,z,t;
$\{\boxed{\exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) * s(A)} \wedge -\infty < \mathrm{e} \wedge \mathrm{e} < +\infty\}$
  (x,y) = locate(e);
$\left\{\begin{array}{l}\exists uv.\ \boxed{\exists ZAB.\ ls(\mathrm{Head}, A, \mathrm{x}) * L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * N(\mathrm{y}, v, Z) * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)}\\ * (\mathrm{x}{+}2{\mapsto}\mathrm{y}) \wedge u < \mathrm{e} \wedge \mathrm{e} \leq v \wedge \mathrm{e} < +\infty\end{array}\right\}$
  t = y.value; if (t == e) {
$\left\{\begin{array}{l}\exists u.\ \boxed{\exists ZAB.\ ls(\mathrm{Head}, A, \mathrm{x}) * L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * N(\mathrm{y}, \mathrm{e}, Z) * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\mathrm{e}{\cdot}B)}\\ * (\mathrm{x}{+}2{\mapsto}\mathrm{y}) \wedge \mathrm{e} < +\infty\end{array}\right\}$
    lock(y);
$\left\{\begin{array}{l}\exists uZ.\ \boxed{\exists AB.\ ls(\mathrm{Head}, A, \mathrm{x}) * L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, Z) * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\mathrm{e}{\cdot}B)}\\ * (\mathrm{x}{+}2{\mapsto}\mathrm{y}) * (\mathrm{y}{+}2{\mapsto}Z) \wedge \mathrm{e} < +\infty\end{array}\right\}$
    z = y.next; x.next = z;
$\left\{\begin{array}{l}\exists u.\ \boxed{\exists AB.\ ls(\mathrm{Head}, A, \mathrm{x}) * L_{\mathrm{tid}}(\mathrm{x}, u, \mathrm{y}) * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) * ls(\mathrm{z}, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}B)}\\ *(\mathrm{x}{+}2{\mapsto}\mathrm{z}) * (\mathrm{y}{+}2{\mapsto}\mathrm{z})\end{array}\right\}$
    unlock(x);
$\{\boxed{\exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) * s(A)} * L_{\mathrm{tid}}(\mathrm{y}, \mathrm{e}, \mathrm{z}) * (\mathrm{y}{+}2{\mapsto}\mathrm{z})\}$
    dispose(y);
  } else { unlock(x); }
$\{\boxed{\exists A.\ ls(\mathrm{Head}, A, \mathrm{nil}) * s(A)}\}$
}
```

B Heap-reading expressions

The examples of this paper use a more complex set of expressions which dereference the heap.

$$E ::= n \mid x \mid [E] \mid E + E \mid E - E \mid \ldots$$

We can, however, translate these *impure* expressions into a formula asserting that they evaluate to a given value, v. We write $\llbracket E \rrbracket_v$ for such a translation.

$$\begin{aligned}
\llbracket n \rrbracket_v &\stackrel{\textbf{def}}{=} v = n\\
\llbracket x \rrbracket_v &\stackrel{\textbf{def}}{=} v = x\\
\llbracket [E] \rrbracket_v &\stackrel{\textbf{def}}{=} \exists y.\ y \mapsto v \wedge \llbracket E \rrbracket_y\\
\llbracket E_1 + E_2 \rrbracket_v &\stackrel{\textbf{def}}{=} \exists x_1\, x_2.\ v = x_1 + x_2 \wedge \llbracket E_1 \rrbracket_{x_1} \wedge \llbracket E_2 \rrbracket_{x_1}
\end{aligned}$$

We say that the expression E is defined in the current heap, if $\llbracket E \rrbracket_x$ is true for some value x.

$$(E)\ \mathrm{def} \iff \exists x. \llbracket E \rrbracket_x$$

Similarly, we can allow boolean tests to access the heap:

$$b ::= E{==}E \mid \ldots$$

These boolean tests can be lifted to formulae in the logic by

$$[\![E_1{=}{=}E_2]\!] \overset{\textbf{def}}{=} \exists x_1, x_2.\ [\![E_1]\!]_{x_1} \wedge [\![E_2]\!]_{x_2} \wedge x_1 = x_2$$

A boolean test is defined in the current heap if all its expressions are defined.

$$(E_1{=}{=}E_2)\ \mathsf{def} \overset{\textbf{def}}{=} \exists x_1, x_2.\ [\![E_1]\!]_{x_1} \wedge [\![E_2]\!]_{x_2}$$

Finally, we can give the actual atomic rule with a heap-reading guard as

$$\frac{\begin{array}{c} \vDash_{\mathrm{SL}} P_1 * P_2 \Longrightarrow (b)\ \mathsf{def} \\ \vdash C\ \mathbf{sat}\ ((P_1 * P_2) \wedge b, \{\}, \{\}, \exists\overline{y}.\ Q_1 * Q_2) \\ \boxed{Q}\ \mathsf{stable\ under}\ R \qquad (P_1 \rightsquigarrow Q_1) \subseteq G \\ \overline{y} \cap FV(P_2) = \emptyset \qquad \vDash_{\mathrm{SL}} P \Rightarrow P_1 * F \qquad \vDash_{\mathrm{SL}} Q_1 * F \Rightarrow Q \end{array}}{\vdash (\mathbf{atomic}_Q(b)\{C\})\ \mathbf{sat}\ (\boxed{\exists\overline{y}.\ P} * P_2, R, G, \exists\overline{y}.\ \boxed{Q} * Q_2)}$$

C Properties of septraction ($-\!\!\circledast$)

The following properties are direct consequences of the definitions.

$$\begin{aligned} \mathbf{emp} -\!\!\circledast P &\iff P \\ (P * Q) -\!\!\circledast R &\iff P -\!\!\circledast (Q -\!\!\circledast R) \\ P -\!\!\circledast Q &\iff P -\!\!\circledast (Q \wedge (P * \mathbf{true})) \end{aligned}$$

In addition, septraction distributes over $\vee$, and semi-distributes over $\wedge$.

$$\begin{aligned} P -\!\!\circledast (Q \vee R) &\iff (P -\!\!\circledast Q) \vee (P -\!\!\circledast R) \\ (P \vee Q) -\!\!\circledast R &\iff (P -\!\!\circledast R) \vee (Q -\!\!\circledast R) \\ P -\!\!\circledast (Q \wedge R) &\Longrightarrow (P -\!\!\circledast Q) \wedge (P -\!\!\circledast R) \\ (Q \wedge R) -\!\!\circledast P &\Longrightarrow (Q -\!\!\circledast P) \wedge (R -\!\!\circledast P) \end{aligned}$$

If P is *exact* (i.e. for all h_1, h_2, and i, if $h_1, i \vDash_{\mathrm{SL}} P$ and $h_2, i \vDash_{\mathrm{SL}} P$ then $h_1 = h_2$), the last two properties become equivalences. When we are septracting a single memory cell, $x \mapsto y$, then futher properties hold:

$$\begin{aligned} x \mapsto y -\!\!\circledast P &\iff (x \mapsto y -\!\!\circledast P)\lfloor_x \\ x \mapsto y -\!\!\circledast z \mapsto w &\iff x = z \wedge y = w \wedge \mathbf{emp} \\ x \mapsto y -\!\!\circledast (P * Q) &\iff ((x \mapsto y -\!\!\circledast P) * Q\lfloor_x) \\ &\qquad \vee ((x \mapsto y -\!\!\circledast Q) * P\lfloor_x) \\ x \mapsto y -\!\!\circledast \mathbf{emp} &\iff \mathbf{false} \end{aligned}$$

where $P\lfloor_x \overset{\textbf{def}}{=} P \wedge \neg(\exists y.\ x \mapsto y * true)$. Intuitively, if we remove a memory cell from P, the result does not contain the removed cell. If we remove a memory cell from another memory cell, the two memory cells must be identical and the resulting state is empty. Removing a memory cell from a separating conjuction of two formulae generates a case split: the cell could belong either to the first conjunct or to the second. This equivalence is reminiscent of the chain rule of differentiation ($\frac{d(yz)}{dx} = \frac{dy}{dx}z + y\frac{dz}{dx}$). Finally, removing a cell from the empty heap is impossible.