RGSep Lecture Notes

Viktor Vafeiadis

May 13, 2014

1 Introduction

In this document, we shall introduce RGSep, a logic that combines rely/guarantee reasoning and separation logic. The presentation here is largely based on [1, Chapter 3] (correcting a number of errors and adapting it to the setting of the programming language introduced in the CSL lecture notes).

2 RGSep Assertions

RGSep divides the heap into two logical components: the local heap, h_{L}, and the shared heap, h_{S}, such that their combination, $h_{\mathrm{L}} \uplus h_{\mathrm{S}}$, is defined.

Its assertion language describes both the local and the shared part of the heap. Here is the syntax of RGSep assertions:

$p, q, r ::=$	P	Local assertion
	$\boxed{P}$	Shared assertion
	$p * q$	Separating conjunction
	$p \wedge q$	Normal conjunction
	$p \vee q$	Disjunction
	$\forall x.\ p$	Universal quantification
	$\exists x.\ p$	Existential quantification

where P stands for any separation logic assertion. We will often call $\boxed{P}$ assertions as boxed assertions. Note, however, that boxes are not modalities in the usual sense as they cannot be nested.

Formally, the semantics of assertions are given as a function of type:

$$[\![\_]\!] : \mathsf{RGSepAssn} \to \mathsf{Stack} \times \mathsf{Heap} \times \mathsf{Heap} \to \{\mathit{true}, \mathit{false}\}$$

This function is defined recursively over the syntax of RGSep assertions:

$$\begin{aligned}
[\![P]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}}) &\overset{\text{def}}{\iff} [\![P]\!]_{\mathrm{SL}}(s, h_{\mathrm{L}}) \wedge \mathsf{def}(h_{\mathrm{L}} \uplus h_{\mathrm{S}})\\
[\![\boxed{P}]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}}) &\overset{\text{def}}{\iff} [\![P]\!]_{\mathrm{SL}}(s, h_{\mathrm{S}}) \wedge (h_{\mathrm{L}} = \mathbf{emp})\\
[\![p * q]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}}) &\overset{\text{def}}{\iff} \exists h_1, h_2.\ (h_{\mathrm{L}} = h_1 \uplus h_2) \wedge [\![p]\!](s, h_1, h_{\mathrm{S}}) \wedge [\![q]\!](s, h_2, h_{\mathrm{S}})\\
[\![p \wedge q]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}}) &\overset{\text{def}}{\iff} [\![p]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}}) \wedge [\![q]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}})\\
[\![p \vee q]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}}) &\overset{\text{def}}{\iff} [\![p]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}}) \vee [\![q]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}})\\
[\![\forall x.\ p]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}}) &\overset{\text{def}}{\iff} \forall v.\ [\![p]\!](s[x := v], h_{\mathrm{L}}, h_{\mathrm{S}})\\
[\![\exists x.\ p]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}}) &\overset{\text{def}}{\iff} \exists v.\ [\![p]\!](s[x := v], h_{\mathrm{L}}, h_{\mathrm{S}})
\end{aligned}$$

Note that the definition of $*$ splits the local state, but not the shared state. We say that $*$ is multiplicative over the local state, but additive over the shared state. In particular, $\boxed{P} * \boxed{Q} \iff \boxed{P \wedge Q}$. The semantics of shared assertions, $\boxed{P}$, could alternatively be presented without l=u. This results in an equally expressive logic, but the definition above leads to shorter assertions in practice.

RGSep formulas include the separation logic formulas and overload the definition of some separation logic operators ($*$, $\wedge$, $\vee$, $\exists$ and $\forall$) to act on RGSep assertions. This overloading is intentional and justified by the following Lemma (writing $\mathsf{Local}(P)$ for the first RGSep assertion kind):

Lemma 1 (Properties of local assertions)**.**

$$\begin{aligned}
\mathsf{Local}(B) &\iff B\\
(\mathsf{Local}(P) * \mathsf{Local}(Q)) &\iff \mathsf{Local}(P * Q)\\
(\mathsf{Local}(P) \wedge \mathsf{Local}(Q)) &\iff \mathsf{Local}(P \wedge Q)\\
(\mathsf{Local}(P) \vee \mathsf{Local}(Q)) &\iff \mathsf{Local}(P \vee Q)\\
(\exists x.\ \mathsf{Local}(P)) &\iff \mathsf{Local}(\exists x.\ P)\\
(\forall x.\ \mathsf{Local}(P)) &\iff \mathsf{Local}(\forall x.\ P)
\end{aligned}$$

These follow directly from the semantic definitions. Because of this lemma, we can reduce the notational overhead by making the Local implicit. This should not cause any confusion, because according to Lemma 1, the RGSep operators and the separation logic operators coincide for local assertions.

3 Describing Interference using RGSep Actions

The strength of rely/guarantee is the relational description of interference between parallel processes. Instead of using relations directly, RGSep describes interference in terms of actions $\vec{x}.\ P \rightsquigarrow Q$ that describe the changes performed to the shared state. Typically, P and Q will 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 $\vec{x}.\ P \rightsquigarrow Q$ is that it replaces the part of the shared state that satisfies P prior to the action with a part satisfying Q without changing the rest of the shared state. For example, consider the following action:

$$M, N.\quad \mathtt{x} \mapsto M \ \rightsquigarrow\ \mathtt{x} \mapsto N \wedge N \geq M \qquad \textsf{(Increment)}$$

It specifies that the value in the heap cell `x` may be changed, but its value is never decremented. The logical variables M and N are existentially bound with scope ranging over both the precondition and the postcondition. In this action, the heap footprints of the precondition and of the postcondition both consist of the location `x`. The footprints of the precondition and the postcondition, however, need not be the same. When they are different, this indicates a transfer of ownership between the shared state and the local state of a thread. For instance, consider a simple lock with two operations: Acquire which changes the lock bit from 0 to 1, and removes the protected object, $list(y)$, from the shared state; and Release which changes the lock bit from 1 to 0, and puts the protected object back into the shared state. We can represent these two operations formally as

$$(\mathtt{x} \mapsto 0) * list(\mathtt{y}) \rightsquigarrow \mathtt{x} \mapsto 1 \qquad \textsf{(Acquire)}$$
$$\mathtt{x} \mapsto 1 \rightsquigarrow (\mathtt{x} \mapsto 0) * list(\mathtt{y}) \qquad \textsf{(Release)}$$

An action $P \rightsquigarrow Q$ represents the modification of some shared state satisfying P to some state satisfying Q. Its semantics is the following relation:

$$[\![\vec{x}.\ P \rightsquigarrow Q]\!] \stackrel{\text{def}}{=} \{(s, h_1 \uplus h_0, h_2 \uplus h_0) \mid \exists \vec{v}.\ [\![P[\vec{v}/\vec{x}]]\!](s, h_1) \wedge [\![Q[\vec{v}/\vec{x}]]\!](s, h_2)\}$$

It relates some initial shared heap h_1 satisfying the precondition P to a final heap h_2 satisfying the postcondition. In addition, there may be some disjoint shared heap h_0 which is not affected by the action. In the spirit of separation logic, we want the action specification 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 existential quantification over the interpretation, $\exists\vec{v}$, allows P and Q to have shared logical variables, such as M and N in Increment.

RGSep represents the rely and guarantee conditions as sets of actions. The relational semantics of a set of actions is the reflexive and transitive closure of the union of the semantics of each action in the set. This allows each action to run any number of times in any interleaved order with respect to the other actions.

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

A specification, $P_1 \leadsto Q_1$ is allowed by a guarantee G if its effect is contained in G.

Definition 2. $(\vec{x}.\ P \leadsto Q) \subseteq G \quad \overset{\text{def}}{\iff} \quad [\![\vec{x}.\ P \leadsto Q]\!] \subseteq [\![G]\!]$.

4 Stability of assertions

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

Definition 3 (Stability). $\mathsf{sem\_stable}(S, R)$ *if and only if for all s, h, and h' such that $[\![S]\!]_{\mathrm{SL}}(s, h)$ and $(s, h, h') \in R$, then $[\![S]\!]_{\mathrm{SL}}(s, h, h')$.*

By representing the interference R as a set of actions, we can reduce stability to a simple syntactic check.

Lemma 4 (Checking stability)**.**

- $\mathsf{sem\_stable}(S, [\![\vec{x}.\ P \leadsto Q]\!])$ *iff* $((P[\vec{v}/\vec{x}] \mathrel{-\!\circledast} S) * Q[\vec{v}/\vec{x}]) \Rightarrow S$ *for all* $\vec{v}$.
- $\mathsf{sem\_stable}(P, (R_1 \cup R_2)^*)$ *iff* $\mathsf{sem\_stable}(S, R_1)$ *and* $\mathsf{sem\_stable}(S, R_2)$.

Informally, the first property says that if from a state that satisfies S, we remove the part of the state satisfying P, and replace it with some state satisfying Q, then this should imply that S holds again. In the case when the action cannot run, because there is no sub-state 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 if and only if it is stable under interference by every action in R.

RGSep forbids interference on the local state, but permits interference on the shared state. Hence, given an RGSep assertion, only the parts of it that describe the shared state may be affected by interference. We shall say that an RGSep assertion is (syntactically) stable under R, if all the boxed assertions it contains are stable under R.

Definition 5. *Let p* stable under *R be defined by induction on p as follows*

- P stable under R *always holds.*
- $\boxed{P}$ stable under R *if and only if* $(P; R) \Rightarrow P$.
- *For* $op ::= * \mid \wedge \mid \vee$, *let* $(p_1\ op\ p_2)$ stable under R *if and only if* p_1 stable under R *and* p_2 stable under R.
- *For* $Q ::= \forall \mid \exists$, *let* $(Qx.\ p)$ stable under R *if and only if* p stable under R.

If an assertion is syntactically stable, then it is also semantically stable in the following sense:

Lemma 6. *If p stable under R and $[\![p]\!](s, h_{\mathrm{L}}, h_{\mathrm{S}})$ and $(s, h_{\mathrm{S}}, h'_{\mathrm{S}}) \in R$, then $[\![p]\!](s, h_{\mathrm{L}}, h'_{\mathrm{S}})$.*

The converse is not true. For example, let

$$R \stackrel{\text{def}}{=} (M, N.\ x \mapsto M \ \rightsquigarrow\ x \mapsto N)$$

The relation, R, writes an arbitrary value to x without changing the rest of the heap. Then, $\boxed{\exists n.\ x \mapsto n}$ is stable under R, whereas the assertions $P_1 = \boxed{\exists n.\ x \mapsto n \wedge n \leq 0}$ and $P_2 = \boxed{\exists n.\ x \mapsto n \wedge n > 0}$ are not. Therefore, $P_1 \vee P_2$ is not syntactically stable although it is semantically equivalent to $\boxed{\exists n.\ x \mapsto n}$.

5 Specifications and proof rules

Specifications of a command C are quadruples (p, R, G, q), where

- The precondition p describes the set of initial states in which C might be executed (both its local and shared parts).
- The rely R is a relation (i.e. a set of actions) describing the interference caused by the environment.
- The guarantee G is a relation describing the changes to the shared state, caused by the program.
- The postcondition q describes the possible resulting local and shared states, should the execution of C terminate.

The judgement C **sat** (p, R, G, q) says that any execution of C from an initial state satisfying p and under environment interference R (*i*) does not fault (e.g., by accessing unallocated memory), (*ii*) causes interference at most G, and, (*iii*) if it terminates, its final state satisfies q. Its formal definition is given in the next section.

First, we have the familiar specification weakening rule:

$$\frac{\begin{array}{ccc} & R \subseteq R' & p \Rightarrow p' \\ C \textbf{ sat } (p', R', G', q') & G' \subseteq G & q' \Rightarrow q \end{array}}{C \textbf{ sat } (p, R, G, q)} \quad (\textsc{Weaken})$$

Next, we have the disjunction and existential quantification rules:

$$\frac{\begin{array}{c} C \textbf{ sat } (p_1, R, G, q) \\ C \textbf{ sat } (p_2, R, G, q) \end{array}}{C \textbf{ sat } (p_1 \vee p_2, R, G, q)} \ (\textsc{Disj}) \qquad \frac{\begin{array}{c} x \notin \mathsf{fv}(q, C, R, G) \\ C \textbf{ sat } (p, R, G, q) \end{array}}{C \textbf{ sat } (\exists x.\ p, R, G, q)} \ (\textsc{Ex})$$

From separation logic, RGSep inherits the frame rule: If a program runs safely with initial state p, it can also run with additional state r lying around. Since the program runs safely without r, it cannot access the additional state; hence, r is still true at the end. Since the frame, r, may also specify the shared state, the FRAME rule checks that r is stable under interference from both the program and its environment. Otherwise, they may invalidate r during their execution. In the simple case when r does not mention the shared state, the stability check is trivially satisfied. Similarly, as in CSL, we require that the frame not to mention any variables modified by the command.

$$\frac{\begin{array}{c} C\ \mathbf{sat}\ (p, R, G, q) \\ r\ \mathsf{stable\ under}\ (R \cup G) \qquad \mathsf{writes}(C) \cap \mathsf{fv}(r) = \emptyset \end{array}}{C\ \mathbf{sat}\ (p * r, R, G, q * r)} \quad (\text{FRAME})$$

Then, there is a proof rule for each construct in the language. The rules for the empty program, sequential composition, non-deterministic choice, and loops are completely standard. Similarly to FRAME, SKIP checks that the precondition, p, is stable under the rely, R. Because the empty program does not change the state, p is trivially stable under interference from the program itself.

$$\frac{p\ \mathsf{stable\ under}\ R}{\mathbf{skip}\ \mathbf{sat}\ (p, R, G, p)} \ (\text{SKIP}) \qquad \frac{\begin{array}{c} p\ \mathsf{stable\ under}\ R \\ C\ \mathbf{sat}\ (p, R, G, p) \end{array}}{C^*\ \mathbf{sat}\ (p, R, G, p)} \ (\text{LOOP})$$

$$\frac{\begin{array}{c} C_1\ \mathbf{sat}\ (p, R, G, r) \\ C_2\ \mathbf{sat}\ (r, R, G, q) \end{array}}{(C_1; C_2)\ \mathbf{sat}\ (p, R, G, q)} \ (\text{SEQ}) \qquad \frac{\begin{array}{c} C_1\ \mathbf{sat}\ (p, R, G, q) \\ C_2\ \mathbf{sat}\ (p, R, G, q) \end{array}}{(C_1 \oplus C_2)\ \mathbf{sat}\ (p, R, G, q)} \ (\text{CHOICE})$$

For primitive commands, c, that do not access the shared state, we adopt the separation logic rules. We have the following rule scheme:

$$\frac{\vdash_{\mathsf{SL}} \{P\}\ c\ \{Q\} \qquad \mathsf{writes}(c) \cap \mathsf{fv}(R, G) = \emptyset}{c\ \mathbf{sat}\ (P, R, G, Q)} \quad (\text{PRIM})$$

The sidecondition that c does not modify any variables appearing in R and G is a general requirement that our rules enforce. (This is because for simplicity we assume that there are no shared updateable variables between threads.)

The parallel composition rule of RGSep is very similar to the parallel composition rule of rely/guarantee. Its crucial difference is that the precondition and postcondition of the composition are the separating conjunction of the preconditions and postconditions of the individual threads. In essence, this is the normal conjunction of the shared state assertions, and the separating conjunction of the local state assertions.

$$\frac{\begin{array}{ll} C_1\ \mathbf{sat}\ (p_1, R \cup G_2, G_1, q_1) & \mathsf{writes}(C_1) \cap \mathsf{fv}(C_2, p_2, q_2) = \emptyset \\ C_2\ \mathbf{sat}\ (p_2, R \cup G_1, G_2, q_2) & \mathsf{writes}(C_2) \cap \mathsf{fv}(C_1, p_1, q_1) = \emptyset \end{array}}{(C_1 \| C_2)\ \mathbf{sat}\ (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)} \quad (\textsc{Par})$$

As the interference experienced by thread C_1 can come from C_2 or from the environment of the parallel composition, we have to ensure that both interferences $(R \cup G_2)$ are allowed. Similarly C_2 must be able to tolerate interference from C_1 and from the environment, R.

The most complex rule is that of atomic commands, **atomic** C. Instead of tackling the general case directly, it is easier if we have two rules. The first rule checks that the atomic block meets its specification in an empty environment, and then checks that the precondition and the postcondition are stable with respect to the actual environment, R. This reduces the problem from an arbitrary rely condition to an empty rely condition.

$$\frac{\begin{array}{ll} (\mathbf{atomic}\ C)\ \mathbf{sat}\ (p, \emptyset, G, q) & p\ \mathsf{stable\ under}\ R \\ \mathsf{writes}(c) \cap \mathsf{fv}(R) = \emptyset & q\ \mathsf{stable\ under}\ R \end{array}}{(\mathbf{atomic}\ C)\ \mathbf{sat}\ (p, R, G, q)} \quad (\textsc{AtomR})$$

The second rule is somewhat trickier. Here is a first attempt:

$$\frac{\begin{array}{c} C\ \mathbf{sat}\ (P * P', \emptyset, \emptyset, Q * Q') \\ (P \rightsquigarrow Q) \subseteq G \qquad \mathsf{writes}(C) \cap \mathsf{fv}(G) = \emptyset \end{array}}{(\mathbf{atomic}\ C)\ \mathbf{sat}\ (\boxed{P} * P', \emptyset, G, \boxed{Q} * Q')}$$

Within an atomic block, we can access the shared state $\boxed{P}$, but we must check that changing the shared state from P from Q is allowed by the guarantee G. This rule is sound, but too weak in practice. It requires that the critical region changes the *entire* shared state from P to Q and that the guarantee condition allows such a change. We can extend the rule by allowing the region to change only *part* of the shared state P into Q, leaving the rest of the shared state (F) unchanged, and checking that the guarantee permits the small change $P \rightsquigarrow Q$.

$$\frac{\begin{array}{c} C\ \mathbf{sat}\ (P * P', \emptyset, \emptyset, Q * Q') \\ (P \rightsquigarrow Q) \subseteq G \qquad \mathsf{writes}(C) \cap \mathsf{fv}(G) = \emptyset \end{array}}{(\mathbf{atomic}\ C)\ \mathbf{sat}\ (\boxed{P * F} * P', \emptyset, G, \boxed{Q * F} * Q')} \quad (\textsc{Atom})$$

From these two rules for atomic blocks, we can derive the following more elaborate rule:

$$\frac{\begin{array}{cc} (P' \rightsquigarrow Q') \subseteq G & C\ \textbf{sat}\ (P' * P'', \emptyset, \emptyset, Q' * Q'') \\ \mathsf{fv}(P'') \cap \{\overline{y}\} = \emptyset & P \Rightarrow P' * F \quad Q' * F \Rightarrow Q \\ \mathsf{writes}(C) \cap \mathsf{fv}(R, G) = \emptyset & \mathsf{sem\_stable}(\exists \overline{y}.P, R) \quad \mathsf{sem\_stable}(Q, R) \end{array}}{(\textbf{atomic}\ C)\ \textbf{sat}\ (\boxed{\exists \overline{y}.\ P} * P'', R, G, \exists \overline{y}.\ \boxed{Q} * Q'')}$$

(Atomic)

This rule is derivable by applying the rules AtomR, Conseq, Ex, and Atom. In the other direction, Atom is derivable from this complex rule, but AtomR is not.

$$\dfrac{(S) \qquad \dfrac{(I) \qquad \dfrac{\dfrac{C\ \textbf{sat}\ (P' * P'', \emptyset, G, Q' * Q'') \qquad (G)}{(\textbf{atomic}\ C)\ \textbf{sat}\ (\boxed{P' * F} * P'', \emptyset, G, \boxed{Q' * F} * Q'')}\ \text{Atom}}{(\textbf{atomic}\ C)\ \textbf{sat}\ (\exists \overline{y}.\ \boxed{P} * P'', \emptyset, G, \boxed{Q} * Q'')}\ \text{Ex}}{(\textbf{atomic}\ C)\ \textbf{sat}\ (\boxed{\exists \overline{y}.\ P} * P'', \emptyset, G, \exists \overline{y}.\ \boxed{Q} * Q'')}\ \text{Conseq}}{(\textbf{atomic}\ C)\ \textbf{sat}\ (\boxed{\exists \overline{y}.\ P} * P'', R, G, \exists \overline{y}.\ \boxed{Q} * Q'')}\ \text{AtomR}$$

In the derivation, (S) stands for $\mathsf{sem\_stable}(\exists \overline{y}.P, R)$, $\mathsf{sem\_stable}(Q, R)$, and $\mathsf{writes}(C) \cap \mathsf{fv}(R) = \emptyset$; (I) stands for $\mathsf{fv}(P'') \cap \{\overline{y}\} = \emptyset$, $P \Rightarrow P' * F$, and $Q' * F \Rightarrow Q$; and (G) stands for $(P' \rightsquigarrow Q') \subseteq G$ and $\mathsf{writes}(C) \cap \mathsf{fv}(G) = \emptyset$.

6 Encodings of SL and RG

Separation logic and rely/guarantee are trivial special cases of RGSep. This is best illustrated by the parallel composition rule.

Plain rely/guarantee: When all the state is shared, we get the standard rely/guarantee rule. In this case, as the local state is empty, we get $p_1 * p_2 \iff p_1 \wedge p_2$ and $q_1 * q_2 \iff q_1 \wedge q_2$.

Formally, we encode $\vdash C\ \mathsf{sat}_{\mathsf{RG}}\ (P, R, G, Q)$ as $C\ \textbf{sat}\ (\boxed{P}, R, G, \boxed{Q})$.

Plain separation logic: When all the state is local, we get the separation logic rules. (Since there is no shared state, we do not need to describe its evolution: R and G are simply the identity relation.)

Formally, we encode $\vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\}$ as $C\ \textbf{sat}\ (P, \emptyset, \emptyset, Q)$.

Concurrent separation logic: We simply thread the resource invariant J through the assertions, and make the rely and guarantee conditions assert that it remains unaffected by interference.

Formally, we encode $J \vdash_{\mathsf{SL}} \{P\}\ C\ \{Q\}$ as $C\ \textbf{sat}\ (P * \boxed{J},\ R,\ R,\ Q * \boxed{J})$, where $R = \{J \rightsquigarrow J\}$.

7 Soundness

As with CSL, we define the meaning of RGSep judgments in terms of a safe predicate. Here, the safe predicate records not only the local heap, h_L, but also the shared heap, h_S, as this is needed for R and G:

Definition 7. $\mathsf{safe}_0(C, s, h_\mathrm{L}, h_\mathrm{S}, R, G, q)$ *holds always.*
$\mathsf{safe}_{n+1}(C, s, h_\mathrm{L}, h_\mathrm{S}, R, G, q)$ *if and only if*
(i) if $C = \mathbf{skip}$*, then* $[\![q]\!](s, h_\mathrm{L}, h_\mathrm{S})$*; and*
(ii) for all h_F*,* $\langle C, s, h_\mathrm{L} \uplus h_\mathrm{S} \uplus h_\mathrm{F}\rangle \not\rightarrow \mathbf{abort}$*; and*
(iii) whenever $\langle C, s, h_\mathrm{L} \uplus h_\mathrm{S} \uplus h_\mathrm{F}\rangle \rightarrow \langle C', s', h'\rangle$*, then there exist* h'_L *and* h'_S *such that* $h' = h'_\mathrm{L} \uplus h'_\mathrm{S} \uplus h_\mathrm{F}$ *and* $(s, h_\mathrm{S}, h'_\mathrm{S}) \in [\![G]\!]$ *and* $\mathsf{safe}_n(C', s', h'_\mathrm{L}, h'_\mathrm{S}, R, G, q)$*;*
(iv) whenever $(s, h_\mathrm{S}, h'_\mathrm{S}) \in [\![R]\!]$ *and* $\mathsf{def}(h_\mathrm{L} \uplus h'_\mathrm{S})$*, then* $\mathsf{safe}_n(C, s, h_\mathrm{L}, h'_\mathrm{S}, R, G, q)$*.*

A configuration is safe for $n + 1$ steps if (i) whenever it is a terminal configuration, it satisfies the postcondition; and (ii) it does not abort; and (iii) whenever it performs a transition, its change to the shared state satisfies the guarantee and the new configuration remains safe for n steps; and finally (iv) whenever the environment changes the shared state according to the rely, the resulting configuration remains safe for another n steps.

The semantics of RGSep judgments is then defined as follows:

Definition 8. C **sat** (p, R, G, q) *holds iff* $\mathit{writes}(C) \cap \mathsf{fv}(R, G) = \emptyset$ *and* $\neg\mathsf{locked}(C)$ *and* $\forall s, h_\mathrm{L}, h_\mathrm{S}, n.\ [\![P]\!](s, h_\mathrm{L}, h_\mathrm{S}) \implies \mathsf{safe}_n(C, s, h_\mathrm{L}, h_\mathrm{S}, R, G, q)$*.*

The definition checks the basic syntactic properties of programs that are RGSep judgments ensure (i.e., that C does not modify the values of variables appearing in R and G, and that the command is a valid user command—not containing the intermediate **inatom** _ marker). It then also requires that any initial configuration of C satisfying the precondition is safe for an arbitrary number of steps.

8 Example: Lock-Coupling List

This section demonstrates a fine-grained concurrent linked list implementation of a mutable set data structure. Instead of having a single lock for the entire list, there is one lock per list node.

Locks Here is the source code for locking and unlocking a node:

```
lock(p)    { atomic { t := p.lock; assume(t = 0); p.lock := TID; } }
unlock(p) { atomic { p.lock := 0; } }
```

These locks store the identifier of the thread that acquired the lock: `TID` represents the thread identifier of the current thread. Storing thread identifiers is not necessary for the algorithm's correctness, but it facilitates the

```
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);
}
```

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

```
add(e) {
  local x, y, z;
  (x, z) := locate(e);
  if (z.value ≠ e) {
    y := new Node();
    y.lock := 0;
    y.value := e;
    y.next := z;
    x.next := y;
  }
  unlock(x);
}
```

Figure 1: Source code for lock coupling list operations.

proof. Similarly, `unlock` uses an `atomic` block to indicate that the write to `p.lock` must be done indivisibly.

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

$$\begin{array}{lcl} N_s(x,v,y) & \overset{\text{def}}{=} & x \mapsto \{.\text{lock}{=}s; .\text{value}{=}v; .\text{next}{=}y\} \\ U(x,v,y) & \overset{\text{def}}{=} & N_0(x,v,y) \\ L_t(x,v,y) & \overset{\text{def}}{=} & N_t(x,v,y) \wedge t > 0 \end{array}$$

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.

$$t \in T \wedge U(x,v,n) \leadsto L_t(x,v,n) \tag{\textsf{Lock}}$$

$$t \in T \wedge L_t(x,v,n) \leadsto U(x,v,n) \tag{\textsf{Unlock}}$$

The Lock and Unlock 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 $T = \{\texttt{TID}\}$, and the rely to use the complement on this set.

From the Atomic rule, we can derive the following rules for the lock primitives. (Arguably, these specifications are not as simple as one might hope. More advanced logics can give better specifications to `lock` and `unlock`.)

$$\frac{\begin{array}{cc} P \text{ stable under } R & Q \text{ stable under } R \\ P \Rightarrow N_{\_}(\mathrm{p}, v, n) * F & L_{\mathrm{TID}}(\mathrm{p}, v, n) * F \Rightarrow Q \end{array}}{\vdash \texttt{lock(p)} \textbf{ sat } (\boxed{P}, R, G, \boxed{Q})} \quad (\textsc{Lock})$$

$$\frac{\begin{array}{cc} P \text{ stable under } R & Q \text{ stable under } R \\ P \Rightarrow L_{\mathrm{TID}}(\mathrm{p}, v, n) * F & U(\mathrm{p}, v, n) * F \Rightarrow Q \end{array}}{\vdash \texttt{unlock(p)} \textbf{ sat } (\boxed{P}, R, G, \boxed{Q})} \quad (\textsc{Unlock})$$

The Algorithm Now we build a fine-grained concurrent linked list implementation of a set using the lock mechanism we have defined. The list has operations `add`, which adds an element to the set, and `remove`, which removes an element from the set. 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.

Figure 1 contains the source code. 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 sentinel values $-\infty$ and $+\infty$ respectively. This allows us to avoid checking for the end of the list.

First, consider the action of adding a node to the list. Here is 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 ensure that the sorted order of the list is preserved, the actual action must be specified with respect to the next node as well. We ensure the value we add is between the previous and next values.

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

The final permitted action is to remove an element from the list. The action specifies that to remove node n from the list, both n and the previous node (x) must be locked. The tail of the previous node is then updated to the removed node's tail, m.

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

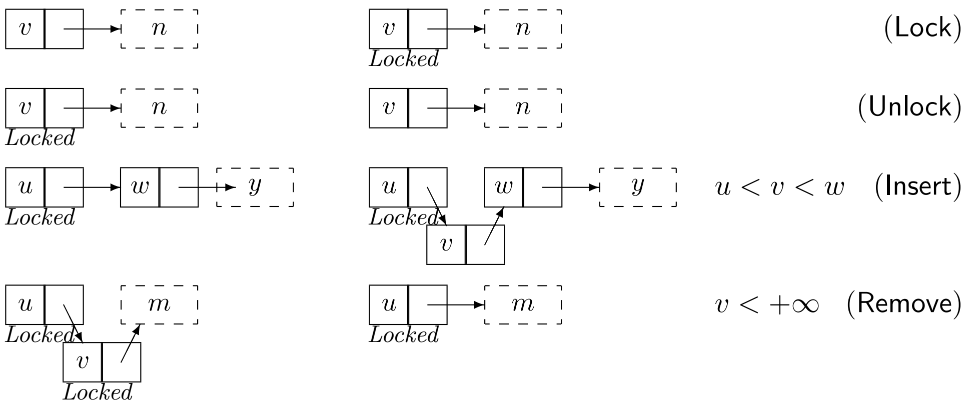

Figure 2: Pictorial representation of the actions

We define $\mathcal{I}(T)$ as the four actions given above: Lock, Unlock, Insert and Remove. These are depicted in Figure 2. $\mathcal{I}(\{t\})$ allows the thread t: (i) to lock an unlocked node, (ii) to unlock a node that it had locked, (iii) to insert a node in the list immediately after a node that it had locked, and (iv) if two adjacent nodes in the list are locked by t, to remove the second node from the list by swinging a pointer past it. For a thread with thread identifier t, take $R = \mathcal{I}(\overline{\{t\}})$ and $G = \mathcal{I}(\{t\})$.

We can use separation to describe the structure of the shared list. The following predicate, $\mathit{ls}(x, A, y)$, describes a list segment starting at location x with the final tail value of y, and with contents A. We write ϵ for the empty sequence and $\cdot$ for the concatenation of two sequences.

$$\begin{aligned} \mathit{ls}(x, A, y) \overset{\text{def}}{=}\ & (x = y \wedge A = \epsilon \wedge \mathbf{emp}) \\ & \vee (\exists vzB.\ x \neq y \wedge A = v{\cdot}B * N_{\_}(x, v, z) * \mathit{ls}(z, B, y)) \end{aligned}$$

Because of separation logic, we do not need any reachability predicates. Instead, the 'list segment' predicate is simply a recursively defined predicate. The definition above ensures that the list does not contain any cycles.

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

$$\mathit{sorted}(A) \overset{\text{def}}{=} \begin{cases} \mathit{true} & \textbf{if } (A = \epsilon) \vee (A = a{\cdot}\epsilon) \\ (a < b) \wedge \mathit{sorted}(b.B) & \textbf{if } (A = a{\cdot}b{\cdot}B) \end{cases}$$

$$s(A) \overset{\text{def}}{=} (\exists B.\ A = -\infty{\cdot}B{\cdot}{+\infty}) \wedge \mathit{sorted}(A) \wedge \mathbf{emp}$$

Figures 3 and 4 contain the proof outlines of `locate`, `add`, and `remove`. The outline presents the intermediate assertions in the proof. Further, we must prove that every shared state assertion is stable under the rely. These

```
locate(e) {
  local p, c, t;
  $\left\{\boxed{\exists A.\ \mathit{ls}(\mathtt{Head}, A, \mathrm{nil}) * s(A)} \wedge -\infty < \mathtt{e}\right\}$
  p := Head;
  $\left\{\boxed{\begin{array}{l}\exists ZB.\ \mathit{ls}(\mathtt{Head}, \epsilon, \mathtt{p}) * N(\mathtt{p}, -\infty, Z)\\ * \mathit{ls}(Z, B, \mathrm{nil}) * s(-\infty \cdot B)\end{array}} \wedge -\infty < \mathtt{e}\right\}$
  lock(p);
  $\left\{\exists Z.\ \boxed{\begin{array}{l}\exists B.\ \mathit{ls}(\mathtt{Head}, \epsilon, \mathtt{p}) * L(\mathtt{p}, -\infty, Z)\\ * \mathit{ls}(Z, B, \mathrm{nil}) * s(-\infty \cdot B)\end{array}} \wedge -\infty < \mathtt{e}\right\}$
  $\langle$c := p.next;$\rangle$
  $\left\{\boxed{\begin{array}{l}\exists B.\ \mathit{ls}(\mathtt{Head}, \epsilon, \mathtt{p}) * L(\mathtt{p}, -\infty, \mathtt{c})\\ * \mathit{ls}(\mathtt{c}, B, \mathrm{nil}) * s(-\infty \cdot B)\end{array}} \wedge -\infty < \mathtt{e}\right\}$
  $\langle$t := c.value;$\rangle$
  $\left\{\exists u.\ \boxed{\begin{array}{l}\exists ABZ.\ \mathit{ls}(\mathtt{Head}, A, \mathtt{p}) * L(\mathtt{p}, u, \mathtt{c})\\ * N(\mathtt{c}, \mathtt{t}, Z) * \mathit{ls}(\mathtt{c}, B, \mathrm{nil}) * s(A \cdot u \cdot \mathtt{t} \cdot B)\end{array}} \wedge u < \mathtt{e}\right\}$
  while (t < e) {
    $\left\{\exists u.\ \boxed{\begin{array}{l}\exists ABZ.\ \mathit{ls}(\mathtt{Head}, A, \mathtt{p}) * L(\mathtt{p}, u, \mathtt{c})\\ * N(\mathtt{c}, \mathtt{t}, Z) * \mathit{ls}(\mathtt{c}, B, \mathrm{nil}) * s(A \cdot u \cdot \mathtt{t} \cdot B)\end{array}} \wedge u < \mathtt{e} \wedge \mathtt{t} < \mathtt{e}\right\}$
    lock(c);
    $\left\{\exists uZ.\ \boxed{\begin{array}{l}\exists AB.\ \mathit{ls}(\mathtt{Head}, A, \mathtt{p}) * L(\mathtt{p}, u, \mathtt{c})\\ * L(\mathtt{c}, \mathtt{t}, Z) * \mathit{ls}(Z, B, \mathrm{nil}) * s(A \cdot u \cdot \mathtt{t} \cdot B)\end{array}} \wedge \mathtt{t} < \mathtt{e}\right\}$
    unlock(p);
    $\left\{\exists Z.\ \boxed{\begin{array}{l}\exists AB.\ \mathit{ls}(\mathtt{Head}, A, \mathtt{c}) * L(\mathtt{c}, \mathtt{t}, Z)\\ * \mathit{ls}(Z, B, \mathrm{nil}) * s(A \cdot \mathtt{t} \cdot B)\end{array}} \wedge \mathtt{t} < \mathtt{e}\right\}$
    p := c;
    $\left\{\exists uZ.\ \boxed{\begin{array}{l}\exists AB.\ \mathit{ls}(\mathtt{Head}, A, \mathtt{p}) * L(\mathtt{p}, u, Z)\\ * \mathit{ls}(Z, B, \mathrm{nil}) * s(A \cdot u \cdot B)\end{array}} \wedge u < \mathtt{e}\right\}$
    $\langle$c := p.next;$\rangle$
    $\left\{\exists u.\ \boxed{\begin{array}{l}\exists AB.\ \mathit{ls}(\mathtt{Head}, A, \mathtt{p}) * L(\mathtt{p}, u, \mathtt{c})\\ * \mathit{ls}(\mathtt{c}, B, \mathrm{nil}) * s(A \cdot u \cdot B)\end{array}} \wedge u < \mathtt{e}\right\}$
    $\langle$t := c.value;$\rangle$
    $\left\{\exists u.\ \boxed{\begin{array}{l}\exists ABZ.\ \mathit{ls}(\mathtt{Head}, A, \mathtt{p}) * L(\mathtt{p}, u, \mathtt{c})\\ * N(\mathtt{c}, \mathtt{t}, Z) * \mathit{ls}(Z, B, \mathrm{nil}) * s(A \cdot u \cdot \mathtt{t} \cdot B)\end{array}} \wedge u < \mathtt{e}\right\}$
  }
  $\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ABZ.\ \mathit{ls}(\mathtt{Head}, A, \mathtt{p}) * L(\mathtt{p}, u, \mathtt{c})\\ * N(\mathtt{c}, v, Z) * \mathit{ls}(Z, B, \mathrm{nil}) * s(A \cdot u \cdot v \cdot B)\end{array}} \wedge u < \mathtt{e} \wedge \mathtt{e} \leq v\right\}$
  return (p, c);
}
```

Figure 3: Outline verification of `locate`.

```
add(e) { local x, y, z, t;
  $\{\boxed{\exists A.\ ls(\mathtt{Head}, A, \mathrm{nil}) * s(A)} \wedge -\infty < \mathtt{e}\}$
  (x, z) := locate(e);
  $\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\mathtt{Head}, A, \mathtt{x}) * L(\mathtt{x}, u, \mathtt{z}) * N(\mathtt{z}, v, Z)\\ * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)\end{array}} \wedge u < \mathtt{e} \wedge \mathtt{e} \le v\right\}$
  ⟨t = z.value;⟩  if(t ≠ e) {
    $\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\mathtt{Head}, A, \mathtt{x}) * L(\mathtt{x}, u, \mathtt{z}) * N(\mathtt{z}, v, Z)\\ * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)\end{array}} \wedge u < \mathtt{e} \wedge \mathtt{e} < v\right\}$
    y = cons(0, e, z);
    $\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\mathtt{Head}, A, \mathtt{x}) * L(\mathtt{x}, u, \mathtt{z}) * N(\mathtt{z}, v, Z)\\ * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)\end{array}} * U(\mathtt{y}, \mathtt{e}, \mathtt{z}) \wedge u < \mathtt{e} \wedge \mathtt{e} < v\right\}$
    ⟨x.next = y;⟩
    $\{\exists uv.\ \boxed{\exists ZAB.\ ls(\mathtt{Head}, A, \mathtt{x}) * L(\mathtt{x}, u, \mathtt{y}) * N(\mathtt{y}, \mathtt{e}, Z) * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\mathtt{e}{\cdot}B)}\}$
  }
  unlock(x);
  $\{\exists v.\ \boxed{\exists A.\ ls(\mathtt{Head}, A, \mathrm{nil}) * s(A)}\}$
}

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

Figure 4: Outline verification of add and remove.

proofs involve reasoning about the septraction operator ($-\!\circledast$); they are long, but straightforward.

Theorem 9. *The lock coupling algorithm is safe, and maintains the sorted nature of the list.*

References

[1] Vafeiadis, V., *Modular fine-grained concurrency verification*. PhD thesis. University of Cambridge (2007) See chapter 3.