Local Rely-Guarantee Reasoning

Xinyu Feng

Toyota Technological Institute at Chicago
Chicago, IL 60637, U.S.A.
feng@tti-c.org

Abstract

Rely-Guarantee reasoning is a well-known method for verification of shared-variable concurrent programs. However, it is difficult for users to define rely/guarantee conditions, which specify threads' behaviors over the whole program state. Recent efforts to combine Separation Logic with Rely-Guarantee reasoning have made it possible to hide thread-local resources, but the shared resources still need to be globally known and specified. This greatly limits the reuse of verified program modules.

In this paper, we propose LRG, a new Rely-Guarantee-based logic that brings local reasoning and information hiding to concurrency verification. Our logic, for the first time, supports a frame rule over rely/guarantee conditions so that specifications of program modules only need to talk about the resources used locally, and the verified modules can be reused in different threads without redoing the proof. Moreover, we introduce a new hiding rule to hide the resources shared by a subset of threads from the rest in the system. The support of information hiding not only improves the modularity of Rely-Guarantee reasoning, but also enables the sharing of dynamically allocated resources, which requires adjustment of rely/guarantee conditions.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification — Correctness proofs, Formal methods; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Concurrency, Rely-Guarantee Reasoning, Separation Logic, Local Reasoning, Information Hiding

1. Introduction

With the development and wide use of multi-core processors, concurrency has become a crucial element in software systems. However, the correctness of concurrent programs is notoriously difficult to verify because of the non-deterministic interleaving of running threads and the exponential size of state spaces. Compositionality is of particular importance for scalable concurrency verification.

Rely-Guarantee reasoning (Jones 1983) is a well-known method for verification of shared-variable concurrent programs. It lets each thread specify its expectation (the rely condition) of state transitions made by its environment, and its guarantee to the environment about transitions made by itself. Since the rely condition specifies all possible behaviors that might interfere with the thread, we do not need to consider the exponential size of possible interleavings during the verifications. However, the rely/guarantee conditions are difficult to formulate in practice, because they need to specify the global program state and these global conditions need to be checked during the execution of the whole thread. Specifically, the composionality and applicability of Rely-Guarantee reasoning are greatly limited by the following problems:

- The *whole* program state is viewed as shared resource and need to be specified in the rely/guarantee conditions, even if a part of the state might be privately owned by a single thread. The thread-private resource has to be exposed in the specifications.
- As part of the specifications of program modules, the rely and guarantee conditions need to specify all the shared resources, even if the module accesses only part of them locally. This limits the reuse of verified program modules in different applications with different shared resources.
- Since the shared resources need to be globally known, it is difficult to support the sharing of dynamically allocated resources, which are not known until they are allocated.
- Some resources might be shared only by a subset of threads, but there is no way to hide them from the rest of threads in the system.

These problems are part of the reasons why Jones (2003) calls for a more compositional approach to concurrency.

Recent works on SAGL (Feng et al. 2007) and RGSep (Vafeiadis and Parkinson 2007) have tried to combine the Rely-Guarantee reasoning with Separation Logic (Ishtiaq and O'Hearn 2001; Reynolds 2002) for better composionality. They split the whole state into thread-private and shared parts. The partition of resources enforced by Separation Logic ensures that each thread cannot touch the private parts of others. The rely and guarantee conditions now only need to specify the part that is indeed shared. These combinations, however, only address the first problem mentioned above. Since they also require the shared resources to be globally known, the last three problems remain unsolved.

In this paper, we propose a new program logic, LRG, for Local Rely-Guarantee reasoning. By addressing all these open problems, our logic makes local reasoning and information hiding a reality in concurrency verification. Our work is based on previous works on Rely-Guarantee reasoning and Separation Logic, and SAGL and RGSep in particular, but makes the following new contributions:

- As an extension of Separation Logic, we introduce the separating conjunction of rely/guarantee conditions. Unlike assertions in Separation Logic, rely/guarantee conditions are binary relations of program states and they specify state transitions.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

The new separating conjunction allows us to formalize two subtransitions conducted over disjoint resources, which is the basis to bring in all the nice ideas developed in Separation Logic for local reasoning and modularity.

- Our logic, for the first time, supports a frame rule over rely and guarantee conditions so that the sharing of resources no longer needs to be globally known. Specifications of program modules only need to talk about the resource used locally, therefore the verified modules can be reused in different contexts without redoing the proof.
- We propose a new rule for hiding the shared resources from the environment. It allows the local sharing of resources among a subset of threads without exposing them to others in the system. In particular, using the hiding rule we can derive a more general rule for parallel composition such that a thread's private resource can be shared by its children without being visible by its siblings. The hiding rule also gives us a way to change the rely and guarantee conditions, so that the sharing of dynamically allocated resources can be supported.
- In addition to these extensions, our work also greatly simplifies SAGL and RGSep. We split program states *conceptually* into thread-private and shared parts, but do not need explicit distinction of them either syntactically in assertions (as in SAGL and RGSep) or semantically in program states and operational semantics (as in RGSep). This gives us a simpler semantic model and makes the logic more flexible to use.
- Treating variables as resources (Parkinson et al. 2006), our work is very general and the same ideas work for traditional Rely-Guarantee-based logics where only variables are used and heaps are not dealt with.
- Our logic can also be viewed as an extension of the Concurrent Separation Logic (O'Hearn 2007) with the more expressive Rely-Guarantee-based specifications, but without sacrificing its compositionality.

In the rest of this paper, we first give an overview of the technical background, and use an example to explain the problems and challenges in Sec. 2. Then, before diving into the formal technical development, we explain informally our approaches in Sec. 3. We present the programming language in Sec. 4, the assertion language in Sec. 5 and the LRG logic in Sec. 6. As an example, in Sec. 7 we show how the program presented in Sec. 2 can be verified in our logic. We discuss related work and conclude in Sec. 8.

2. Background and Challenges

In this section, we give an overview of Rely-Guarantee reasoning, Concurrent Separation Logic, and recent works on combining them (Feng et al. 2007; Vafeiadis and Parkinson 2007). Then we use an example to show the problems with existing approaches.

2.1 Rely-Guarantee Reasoning

In Rely-Guarantee reasoning, each thread views the set of all other threads in the system as its environment. The interface between the thread and its environment is specified using a pair of rely and guarantee conditions. The rely condition R specifies the thread's expectations of state transitions made by its environment. The guarantee G specifies the state transitions made by the thread itself. R and G are predicates over a pair of states, *i.e.,* the initial one before the transition and the resulting one after the transition. The specification of a thread is a quadruple (p,R,G,q), where p and q are pre- and post-conditions. A thread satisfies its specification if, given an initial state satisfying p and an environment whose behaviors satisfy R, each atomic transition made by the thread satisfies G and the state at the end satisfies q.

Parallel Composition. To ensure two parallel threads can collaborate without interference, we need to check that their interfaces are compatible in the sense that the rely condition of each thread is implied by the guarantee of the other. Below is the rule for the parallel composition $C_1 \parallel C_2$:

$$\frac{C_1\ \mathbf{sat}\ (p, R\vee G_2, G_1, q_1) \quad C_2\ \mathbf{sat}\ (p, R\vee G_1, G_2, q_2)}{C_1 \parallel C_2\ \mathbf{sat}\ (p, R, G_1\vee G_2, q_1\wedge q_2)}$$

It shows that, to verify $C_1 \parallel C_2$, we can verify the children C_1 and C_2 separately. The rely condition of each child captures the behavior of both its parent's environment (R) and its sibling (G_1 or G_2). It is easy to check that the rely and guarantee conditions for C_1 and C_2 are compatible, *i.e.,* $G_1 \Rightarrow (R\vee G_1)$ and $G_2 \Rightarrow (R\vee G_2)$.

Stability. Each thread is verified with respect to its specification in a similar way as the verification of sequential programs in Hoare Logic, except that we also need to ensure the behavior of every atomic operation satisfies the guarantee, and the precondition at each step is *stable* with respect to the rely condition.

The stability means, if the current state satisfies the precondition p and the current thread is preempted by its environment, p still holds when the current thread resumes its execution in a new state as long as the transition made by the environment satisfies its rely condition R. The stability check is essential to ensure the non-interference between the thread and its environment, but it requires R to capture all possible behaviors of the environment, which makes R (and G) difficult to define and limits the compositionality of the Rely-Guarantee reasoning.

2.2 Separation Logic and Concurrency Verification

Separation Logic (Ishtiaq and O'Hearn 2001; Reynolds 2002) is an extension of Hoare Logic with effective reasoning about memory aliasing. The separating conjunction $p * p'$ in the assertion language specifies program states that can be split into two disjoint parts satisfying p and p' respectively. Because of the separation, update of the part satisfying p does not affect the validity of p' over the other part. The frame rule, as shown below (with some side conditions elided), supports local reasoning of program modules:

$$\frac{\{p\}C\{q\}}{\{p*r\}C\{q*r\}}$$

The specifications p and q for C need to only talk about states accessed locally by C. When C is composed with other modules in different contexts, different r can be added in the specification by applying the frame rule, without redoing the proof.

O'Hearn has proposed Concurrent Separation Logic (CSL), which applies Separation Logic to reason about concurrent programs (O'Hearn 2007). Unlike Rely-Guarantee reasoning, CSL ensures non-interference by enforcing the separation of resources accessible by different threads. The parallel composition rule in CSL is as follows:

$$\frac{\{p_1\}C_1\{q_1\} \quad \{p_2\}C_2\{q_2\}}{\{p_1 * p_2\}C_1 \parallel C_2\{q_1 * q_2\}}$$

Verification of each sequential thread in CSL is the same as in Separation Logic. The frame rule is also sound in CSL. CSL also allows threads to share resources, but only in conditional critical regions that can be entered by only one thread at a time. The well-formedness of shared resources is specified using invariants, which need to be satisfied when threads exit critical regions.

```
1   nd := 0;
2   while (nd = 0) do {
3     lk := 0;
4     while (lk ≠ 1) do {                          // acquire lock
5       ⟨lk := [lhead]; if (lk = 1) then [lhead] := 0;⟩
6     }
7     nd := [lhead+1];                              // first node
8     if (nd ≠ 0) then {
9       tmp := [nd+2]; [lhead+1] := tmp        // remove node
10    }
11    ⟨[lhead] := 1⟩;                               // release lock
12  }
13  tmp := [nd]; tmp' := [nd+1]; x := cons(tmp,tmp');
14  C_gcd
```

Figure 1. GCD of Nodes on List

2.3 Combinations of the Two Approaches

The rely and guarantee conditions in Rely-Guarantee reasoning specify state transitions, which are expressive and are suitable to reason about fine-grained concurrent programs. On the other hand, the method views the whole state as a shared resource among threads, which makes it less compositional. CSL has very nice compositionality, but the limited expressiveness of invariants for shared resources makes it unsuitable for fine-grained concurrency.

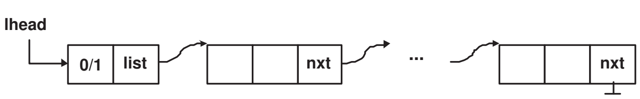
SAGL (Feng et al. 2007) and RGSep (Vafeiadis and Parkinson 2007) have tried to combine merits of both approaches. They split the whole state into thread-private and shared parts. Specifications of threads are in the form of $((p,r),R,G,(q,r'))$, where p and q are pre- and post-conditions specifying the private resources of the thread, while r and r' for the shared part.[1] Their rules for parallel composition are as follows:

$$\frac{C_1 \ \textbf{sat}\ ((p_1,r),R\vee G_2,G_1,(q_1,r_1)) \qquad C_2 \ \textbf{sat}\ ((p_2,r),R\vee G_1,G_2,(q_2,r_2))}{C_1 \parallel C_2 \ \textbf{sat}\ ((p_1 * p_2,r),R,G_1\vee G_2,(q_1 * q_2,r_1\wedge r_2))}$$

The partition of resources enforced by the separating conjunction ensures that each thread cannot touch the private parts of others. The rely and guarantee conditions now only need to specify the part that is indeed shared.

2.4 Problems and Challenges

To see the problems with all these approaches, we first look at a simple program shown in Figs. 1 and 2. The program removes a node from a shared linked list and then computes the greatest common divisor (GCD) of the two numbers stored in the node.



The shared data structure is shown above. The global constant `lhead` points to two memory cells. The first one contains a binary mutex, which enforces mutual exclusive accesses to the linked list pointed to by the pointer stored in the second memory cell.

We use "$\langle C\rangle$" to mean C is executed atomically. "x := [E]" ("[E] := E'") loads values from (stores values into) memory at the location E. "**cons**$(E_1,\ldots,E_n)$" allocates n memory cells with initial values $E_1,\ldots,E_n$. The thread shown in Fig. 1 can be viewed as a consumer of a producer-consumer style program, where the producer (not shown here) generates random numbers and puts them onto the list. Code from line 1 to line 12 acquires the lock, removes a node from the list and then releases the lock. Line 13 copies numbers in the node to newly allocated memory cells. The code C_{gcd} in Line 14 refers to the program in Fig. 2, where two threads collaborate to compute the GCD of numbers pointed to by x.

[1] RGSep uses $p * \boxed{r}$ instead of (p,r).

```
⟨t11 := [x]⟩;                  ||  ⟨t21 := [x+1]⟩;
⟨t12 := [x+1]⟩;                ||  ⟨t22 := [x]⟩;
while (t11 ≠ t12) do{          ||  while (t21 ≠ t22) do{
  if(t11 > t12) then {         ||    if(t21 > t22) then {
    t11 := t11 − t12;          ||      t21 := t21 − t22;
    ⟨[x] := t11⟩;              ||      ⟨[x+1] := t21⟩;
  }                            ||    }
  ⟨t12 := [x+1]⟩;              ||    ⟨t22 := [x]⟩;
}                              ||  }
```

Figure 2. Concurrent GCD

This is a very simple program, but there is no clean and modular way to verify it using existing logics described in the previous sections for the following reasons:

1. C_{gcd} shown in Fig. 2 is a fine-grained concurrent program – two threads share the memory without using locks. The correctness of the code is based on the fact that each thread *preserves* the value at the memory location where the other may update; and that all updates *decreases* the values and preserves the GCD. It is difficult to verify the code using CSL because the invariant of shared resources cannot express preservation and decrease of values without heavy use of auxiliary variables.
2. The functionality of C_{gcd} is self-contained. We want to verify it once and reuse it in different contexts. However, both original Rely-Guarantee reasoning and recent extensions described in Sec. 2.3 require the shared resource be globally known. As a result, when C_{gcd} is verified, the rely and guarantee conditions have to specify the shared list even if it is not accessed by C_{gcd}, negating the very advantage of sequential Separation Logic.
3. The memory block pointed to by x is shared locally by the two threads in C_{gcd}, but not used elsewhere. We should be able to hide it and make it invisible outside when we specify the rely and guarantee conditions for the thread in Fig. 1. This is not supported by existing work on Rely-Guarantee reasoning.
4. Even if we give up the third requirement and are willing to expose the local sharing inside C_{gcd} in the global rely and guarantee conditions, we cannot do so because the memory block pointed to by x is dynamically allocated, whose location is unknown at the beginning.

Among these problems, the first one is with CSL, while the rest are with Rely-Guarantee reasoning, including SAGL and RGSep.

Polymorphic Interpretations of Rely/Guarantee Conditions? It is important to note that using a polymorphic interpretation of rely and guarantee conditions does not automatically solve these problems. For instance, we may want to interpret the validity of the rely condition R over state transitions from σ to σ' in a way such that the following property holds:

If $(\sigma,\sigma')\models R$, $\sigma\perp\sigma''$, *and* $\sigma'\perp\sigma''$, *then* $(\sigma\uplus\sigma'',\sigma'\uplus\sigma'')\models R$.

Here we use $\sigma\perp\sigma''$ to mean σ and σ'' are disjoint, and use $\sigma\uplus\sigma''$ to mean the merge of disjoint states. Their formal definitions are shown in Sec. 5. Although this interpretation takes care of the part of state that is not explicitly specified in R, it does not support local specification and cannot address the second problem mentioned above. Suppose we have verified C_{gcd} with a local specification of R that does not mention the shared list, the interpretation requires that the list be preserved by the environment, which is too strong an assumption and cannot be matched with the actual rely condition for the first 13 lines of code (and the consequence rule cannot be applied, which only allows strengthening of the rely condition).

As a second try, let's consider a very weak interpretation that satisfies the following property:

If $(\sigma,\sigma') \models R$, $\sigma \perp \sigma''$, *and* $\sigma' \perp \sigma'''$, *then* $(\sigma \uplus \sigma'', \sigma' \uplus \sigma''') \models R$.

It says the part of the state unspecified in R might be changed arbitrarily. This interpretation, however, is too weak for the guarantee condition G. So we probably need a different interpretation for G, *e.g.,* the first one. Using different interpretations for R and G makes the logic complicated. A more serious problem with this approach is that it does not allow the hiding of locally shared resources. In the parallel composition rule shown in Sec. 2.1, if we do not specify in R the resource locally shared by C_1 and C_2, the new rely condition $R \vee G_2$ for C_1 is then too weak to be useful. The variation of the rule in Sec. 2.3 has the same problem.

3. Our Approach

As in SAGL and RGSep, we also split program states into thread-private and shared parts. Each thread has exclusive access of its own private resources. Rely/guarantee conditions only specify the shared part. But we try to borrow more ideas from Separation Logic to address the remaining compositionality problems.

We first introduce the separating conjunction over actions, *i.e.,* binary relations of states specifying state transitions. Rely and guarantee conditions are all actions. Similar to the separating conjunction $p * p'$ in Separation Logic, $R * R'$ (or $G * G'$) means the two sub-actions R and R' (or G and G') occur over disjoint parts of states. A formal definition will be given in Sec. 5.

We can now extend the frame rule in Separation Logic to support local rely and guarantee conditions:

$$\frac{R,G \vdash \{(p,r)\}C\{(q,r')\} \quad m \text{ stable with respect to } R' \quad \ldots}{R*R', G*G' \vdash \{(p, r*m)\}C\{(q, r'*m)\}}$$

Following SAGL and RGSep, here we specify private and shared resources separately in pre- and post-conditions (but not in our formal development). We use $R,G \vdash \{(p,r)\}C\{(q,r')\}$ to represent the old judgment C **sat** $((p,r),R,G,(q,r'))$ described in Sec. 2.3. The frame rule says we can verify C with the local specification $((p,r),R,G,(q,r'))$. When C is executed in different contexts with the extra shared resource specified by m, we know C satisfies the "bigger" specification without redoing the proof. The stability check that causes compositionality problems in Rely-Guarantee reasoning, as explained in Sec. 2.1, is no longer an issue here because we can prove $r*m$ is stable with respect to $R*R'$ if r is stable with respect to R and m is stable with respect to R' (we actually need some subtle constraints to prove this, which will be explained in Sec. 5).

The simpler frame rule for private resources, as shown below, is supported in SAGL and RGSep and is sound in our logic too.

$$\frac{R,G \vdash \{(p,r)\}C\{(q,r')\}}{R,G \vdash \{(p*m,r)\}C\{(q*m,r')\}}$$

Since the rely/guarantee conditions specify only the shared resources, they do not need to be changed with the extension of the private predicates.

To allow the hiding of the local sharing of resources by a subset of threads, we introduce a new hiding rule:

$$\frac{R*R', G*G' \vdash \{(p, r*m)\}C\{(q, r'*m')\} \quad \text{[side-conditions omitted]}}{R,G \vdash \{(p*m, r)\}C\{(q*m', r')\}}$$

$$
\begin{array}{llcl}
(Expr) & E & ::= & x \mid X \mid n \mid E+E \mid E-E \mid \ldots \\
(Bexp) & B & ::= & \mathsf{true} \mid \mathsf{false} \mid E=E \mid E \neq E \mid \ldots \\
(Stmts) & C & ::= & x := E \mid x := [E] \mid [E] := E \mid \mathbf{skip} \\
 & & \mid & x := \mathbf{cons}(E,\ldots,E) \mid \mathbf{dispose}(E) \mid C;C \\
 & & \mid & \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ B\ \mathbf{do}\ C \mid C_1 \parallel C_2 \\
 & & \mid & \mathbf{atomic}(B)\{C\}
\end{array}
$$

Figure 3. The Language

$$
\begin{array}{rcl}
(Store) & s & \in PVar \rightharpoonup_{\text{fin}} Int \\
(LvMap) & i & \in LVar \rightharpoonup_{\text{fin}} Int \\
(Heap) & h & \in Nat \rightharpoonup_{\text{fin}} Int \\
(State) & \sigma & \in Store \times LvMap \times Heap \\
(Trans) & \mathcal{R}, \mathcal{G} & \in \mathcal{P}(State \times State)
\end{array}
$$

Figure 4. Program States

This rule says if the resource specified by m and m' is shared locally inside C, and transitions over the resource is specified by R' and G', we can treat it as private and hide R' and G' in the rely/guarantee conditions so that it is invisible from the outside world. Although this rule only converts part of the shared resource into private and does not hide its existence in the pre- and post-conditions, it does hide the resource from other threads because rely/guarantee conditions are the interface between threads.

At first glance, this rule seems to allow a thread to arbitrarily hide any shared resources so that it can cheat its environment. The thread, however, cannot abuse this freedom because the inappropriate hiding will be detected at the point of parallel composition. We will explain this in detail in Sec. 6.

The hiding rule is particularly interesting when C is a parallel composition ($C_1 \parallel C_2$). It allows us to derive a more general rule for parallel composition such that the children threads may share resources that appear to the outside world as private resources of the parent thread. This also solves the last problem described in Sec. 2.4. Sharing of dynamically allocated resources usually follows the pattern of $C_0;(C_1 \parallel C_2)$, like the example in Sec. 2.4. New resources are allocated by C_0 and then shared by C_1 and C_2. Since they are unknown at the beginning of C_0, we cannot specify them in the global R and G. Our new rule allows us to leave them unspecified in the R and G outside of $C_1 \parallel C_2$.

Note that the rules shown in this section are used to illustrate our basic ideas in a semi-formal way. The actual ones in the LRG logic are presented in Sec. 6 and are in different forms. In particular, we do not need the pairs (p,r) as pre- and post-conditions.

4. The Language

The syntax of the language is defined in Fig. 3. We use x and X to represent program variables ($PVar$) and logical variables ($LVar$) respectively. The expressions E and B are pure. The statement $x := [E]$ ($[E] := E'$) loads values from (stores the value E' into) memory at the location E. $x := \mathbf{cons}(E_1,\ldots,E_n)$ allocates n consecutive fresh memory cells and initializes them with $E_1, \ldots, E_n$. The starting address is picked nondeterministically and assigned to x. $\mathbf{dispose}(E)$ frees the memory cell at the location E.

The atomic block $\mathbf{atomic}(B)\{C\}$ executes C atomically if B holds. Otherwise the current process is blocked until B becomes true. As pointed out by Vafeiadis and Parkinson (2007), it can be

$$\frac{\{\ell,\ldots,\ell{+}k{-}1\}\cap dom(h)=\emptyset \quad [\![E_1]\!]_{(s,i)}=n_1 \quad [\![E_k]\!]_{(s,i)}=n_k \quad x\in dom(s)}{(x:=\mathbf{cons}(E_1,\ldots,E_k),(s,i,h)) \leadsto (\mathbf{skip},(s\{x\leadsto \ell\},i,h\uplus\{\ell\leadsto n_1,\ldots,\ell{+}k{-}1\leadsto n_k\}))}$$

$$\frac{[\![E_j]\!]_{(s,i)}\ \mathit{undefined} \quad (1\le j\le k) \quad \text{or} \quad x\notin dom(s)}{(x:=\mathbf{cons}(E_1,\ldots,E_k),(s,i,h)) \leadsto \mathbf{abort}}$$

$$\frac{[\![E]\!]_{(s,i)}=\ell \quad \ell\in dom(h)}{(\mathbf{dispose}(E),(s,i,h)) \leadsto (\mathbf{skip},(s,i,h\setminus\{\ell\}))} \qquad \frac{[\![E]\!]_{(s,i)}\ \mathit{undefined} \quad \text{or} \quad [\![E]\!]_{(s,i)}\notin dom(h)}{(\mathbf{dispose}(E),(s,i,h)) \leadsto \mathbf{abort}}$$

$$\frac{(C_1,\sigma)\leadsto(C_1',\sigma')}{(C_1;C_2,\sigma)\leadsto(C_1';C_2,\sigma')} \qquad \frac{(C_1,\sigma)\leadsto\mathbf{abort}}{(C_1;C_2,\sigma)\leadsto\mathbf{abort}} \qquad \frac{}{(\mathbf{skip};C,\sigma)\leadsto(C,\sigma)}$$

$$\frac{(C_1,\sigma)\leadsto(C_1',\sigma')}{(C_1\,\|\,C_2,\sigma)\leadsto(C_1'\,\|\,C_2,\sigma')} \qquad \frac{(C_2,\sigma)\leadsto(C_2',\sigma')}{(C_1\,\|\,C_2,\sigma)\leadsto(C_1\,\|\,C_2',\sigma')} \qquad \frac{(C_1,\sigma)\leadsto\mathbf{abort} \quad \text{or} \quad (C_2,\sigma)\leadsto\mathbf{abort}}{(C_1\,\|\,C_2,\sigma)\leadsto\mathbf{abort}}$$

$$\frac{}{(\mathbf{skip}\,\|\,\mathbf{skip},\sigma)\leadsto(\mathbf{skip},\sigma)} \qquad \frac{[\![B]\!]_\sigma=\mathsf{tt} \quad (C,\sigma)\leadsto^*(\mathbf{skip},\sigma')}{(\mathbf{atomic}(B)\{C\},\sigma)\leadsto(\mathbf{skip},\sigma')} \qquad \frac{[\![B]\!]_\sigma=\mathsf{tt} \quad (C,\sigma)\leadsto^*\mathbf{abort}}{(\mathbf{atomic}(B)\{C\},\sigma)\leadsto\mathbf{abort}}$$

$$\frac{(\sigma,\sigma')\in\mathcal{R}}{(C,\sigma)\xmapsto{\mathcal{R}}(C,\sigma')} \qquad \frac{(C,\sigma)\leadsto(C',\sigma')}{(C,\sigma)\xmapsto{\mathcal{R}}(C',\sigma')} \qquad \frac{(C,\sigma)\leadsto\mathbf{abort}}{(C,\sigma)\xmapsto{\mathcal{R}}\mathbf{abort}}$$

Figure 5. Operational Semantics

used to model synchronizations at different levels, such as a system-wide lock or atomicity guaranteed by transactional memory. The use of $\mathbf{atomic}(\mathsf{true})\{C\}$ can also be viewed as annotations of atomic machine instructions for fine-grained concurrency (Parkinson et al. 2007). The other statements in Fig. 3 have standard meanings.

Figure 4 presents the model of program states. The store s is a finite partial mapping from program variables to integers; the logical variable mapping i maps logical variables to integers; and the heap h maps memory locations (natural numbers) to integers. The program state σ is a triple of (s,i,h). State transitions $\mathcal{R}$ and $\mathcal{G}$ are binary relations of states.

The semantics of E and B are defined by $[\![E]\!]$ and $[\![B]\!]$ respectively. $[\![E]\!]$ is a partial function of type

$$\mathit{Store}\times\mathit{LvMap}\rightharpoonup\mathit{Int}\,.$$

$[\![B]\!]$ is a partial function of type

$$\mathit{Store}\times\mathit{LvMap}\rightharpoonup\{\mathsf{tt},\mathsf{ff}\}\,.$$

Their definitions are straightforward and are omitted here. We treat program variables as resources, following Parkinson et al. (2006). The semantic functions are undefined if variables in E and B are not assigned values in s and i.

The single step execution of a process is modeled as a binary relation:

$$\_\leadsto\_ \;\in\; \mathcal{P}((\mathit{Stmts}\times\mathit{State})\times((\mathit{Stmts}\times\mathit{State})\cup\{\mathbf{abort}\}))$$

It is defined formally in Fig. 5. Given a statement C and a state σ, we have three cases. First, C can execute one step. We have a new state σ' and a remaining statement C'. In this case, we have $(C,\sigma)\leadsto(C',\sigma')$. If it is not safe to execute the next statement in the state σ, we have $(C,\sigma)\leadsto\mathbf{abort}$. In the third case, the program gets stuck, although σ satisfies the safety requirements. Then $(C,\sigma)\leadsto\_$ is undefined. The third case occurs when C is **skip**, or it begins with an atomic statement $\mathbf{atomic}(B)\{C'\}$ such that B does not hold over σ or C' does not terminate. The **skip** statement plays two roles here: a statement that has no computation effects or a flag to show the end of execution. $\_\leadsto^*\_$ is the transitive-reflexive closure of the single step transition relation. Semantics of the most common statements are elided and are presented in the extended version (Feng 2008).

$$\begin{array}{rlcl}
(\mathit{PVarList}) & O & ::= & \bullet \mid x,O \\
(\mathit{Assertion}) & p,q,r,I & ::= & B \mid \mathsf{emp_h} \mid \mathsf{emp_s} \mid \mathsf{Own}(x) \\
& & & \mid\ E\mapsto E \mid p*q \mid p\mathbin{-\circledast}q \mid \ldots \\
(\mathit{Action}) & a,R,G & ::= & p\ltimes q \mid [p] \mid a*a \mid \exists X.a \mid a\Rightarrow a' \\
& & & \mid\ a\vee a \mid \ldots
\end{array}$$

Figure 6. The Assertion Language

We use $\mathcal{R}$ to represent the possible transitions made by the environment. Then the binary relation $\_\xmapsto{\mathcal{R}}\_$, as defined in Fig. 5, represents one step of state transitions made either by the current process or by its environment characterized by $\mathcal{R}$. Our treatment of the atomic block $\mathbf{atomic}(B)\{C\}$ follows Vafeiadis and Parkinson (2007): execution of the statement C appears to finish in one step and cannot be interrupted by the environment.

5. The Assertion Language

The assertion language is shown in Fig. 6. We use the Separation Logic assertions to specify program states. Following Parkinson et al. (2006), we treat program variables as resources, but do not use fractional permissions, which are orthogonal to our technical development.

Semantics of some assertions are shown in Fig. 7. The boolean expression B holds over a state only if it evaluates to true. It is important to note that, with program variables as resources, the boolean expression $E=E$ does not always hold. It holds if and only if the store contains the variables needed to evaluate E. The assertions $\mathsf{emp_h}$ and $\mathsf{emp_s}$ specify empty heaps and stores respectively. $\mathsf{Own}(x)$ means the ownership of the variable x. $E_1\mapsto E_2$ specifies a singleton heap with E_2 stored at the location E_1. It also requires that the store contain variables used to evaluate E_1 and E_2. The separating conjunction $p*q$ means p and q hold over disjoint part of state. Here we use $f\perp g$ to mean the two finite partial mappings f and g have disjoint domains. The union of two disjoint states σ_1 and σ_2 is defined as $\sigma_1\uplus\sigma_2$. The septraction $p\mathbin{-\circledast}q$, introduced by Vafeiadis and Parkinson (2007), means the state can be extended with a state satisfying p and the extended state satisfies q. The assertions $O\Vdash p$ and emp are syntactic sugars,

$(s,i,h) \models_{\mathrm{SL}} B$ iff $[\![B]\!]_{(s,i)} = \mathsf{tt}$

$(s,i,h) \models_{\mathrm{SL}} \mathsf{emp_s}$ iff $s = \emptyset$

$(s,i,h) \models_{\mathrm{SL}} \mathsf{emp_h}$ iff $h = \emptyset$

$(s,i,h) \models_{\mathrm{SL}} \mathsf{Own}(x)$ iff $dom(s) = \{x\}$

$(s,i,h) \models_{\mathrm{SL}} E_1 \mapsto E_2$ iff there exist ℓ and n such that $[\![E_1]\!]_{(s,i)} = \ell, [\![E_2]\!]_{(s,i)} = n$, $dom(h) = \{\ell\}$ and $h(\ell) = n$

$f \perp g \stackrel{\text{def}}{=} dom(f) \cap dom(g) = \emptyset$

$$(s,i,h) \uplus (s',i',h') \stackrel{\text{def}}{=} \begin{cases} (s \cup s', i, h \cup h') & \text{if } s \perp s',\ h \perp h',\ i = i' \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$\sigma \models_{\mathrm{SL}} p_1 * p_2$ iff there exist σ_1 and σ_2 such that $\sigma_1 \uplus \sigma_2 = \sigma, \sigma_1 \models_{\mathrm{SL}} p_1$ and $\sigma_2 \models_{\mathrm{SL}} p_2$

$\sigma \models_{\mathrm{SL}} p -\!\!\circledast q$ iff there exist σ' and σ'' such that $\sigma'' = \sigma \uplus \sigma', \sigma' \models_{\mathrm{SL}} p$ and $\sigma'' \models_{\mathrm{SL}} q$

$x_1, \ldots, x_n, \bullet \Vdash p \stackrel{\text{def}}{=} (\mathsf{Own}(x_1) * \cdots * \mathsf{Own}(x_n)) \wedge p$

$\mathsf{emp} \stackrel{\text{def}}{=} \mathsf{emp_s} \wedge \mathsf{emp_h}$

Figure 7. Semantics of Selected Separation Logic Assertions

$(\sigma,\sigma') \models p \ltimes q$ iff $\sigma.i = \sigma'.i$, $\sigma \models_{\mathrm{SL}} p$ and $\sigma' \models_{\mathrm{SL}} q$

$(\sigma,\sigma') \models [p]$ iff $\sigma = \sigma'$ and $\sigma \models_{\mathrm{SL}} p$

$(\sigma,\sigma') \models a * a'$ iff there exist σ_1, σ_2, σ_1' and σ_2' such that $\sigma_1 \uplus \sigma_2 = \sigma$, $\sigma_1' \uplus \sigma_2' = \sigma', (\sigma_1,\sigma_1') \models a$, and $(\sigma_2,\sigma_2') \models a'$

$((s,i,h),(s',i,h')) \models \exists X.a$ iff there exist n and i' such that $i' = i\{X \rightsquigarrow n\}$, and $((s,i',h),(s',i',h')) \models a$

$(\sigma,\sigma') \models a \Rightarrow a'$ iff if $(\sigma,\sigma') \models a$, then $(\sigma,\sigma') \models a'$

$\ldots$

$\mathsf{Emp} \stackrel{\text{def}}{=} \mathsf{emp} \ltimes \mathsf{emp}$ $\quad \mathsf{True} \stackrel{\text{def}}{=} \mathsf{true} \ltimes \mathsf{true}$ $\quad \mathsf{Id} \stackrel{\text{def}}{=} [\mathsf{true}]$

$[\![a]\!] \stackrel{\text{def}}{=} \{(\sigma,\sigma') \mid (\sigma,\sigma') \models a\}$

Figure 8. Semantics of Actions

whose definitions are also shown in Fig. 7. O contains a set of program variables, as defined in Fig. 6. We omit other assertions here, which are standard separation logic assertions.

As in Separation Logic, the precision of assertions is defined below. Informally, a predicate p is precise if and only if for any state there is at most one sub-state satisfying p.

Definition 5.1 (Precise Assertions) An assertion p is precise, *i.e.,* $\mathsf{precise}(p)$ holds, if and only if for all s, i, h, s_1, s_2, h_1, h_2, if $s_1 \subseteq s$, $s_2 \subseteq s$, $h_1 \subseteq h$, $h_2 \subseteq h$, $(s_1,i,h_1) \models_{\mathrm{SL}} p$ and $(s_2,i,h_2) \models_{\mathrm{SL}} p$, then $s_1 = s_2$ and $h_1 = h_2$.

Actions. We use actions a to specify state transitions. As shown in Fig. 6, rely/guarantee conditions of threads are actions. Semantics of actions are defined in Fig. 8. The action $p \ltimes q$ means the initial state of the transition satisfies p and the resulting state satisfies q. $[p]$ specifies an identity transition with the states satisfying p. $a * a'$ means the actions a and a' start from disjoint states and

$$\frac{}{[p] \Rightarrow p \ltimes p} \quad \frac{}{[p] \Rightarrow \mathsf{Id}} \quad \frac{}{[\mathsf{emp}] \Leftrightarrow \mathsf{Emp}} \quad \frac{}{a \Rightarrow \mathsf{True}}$$

$$\frac{}{a * \mathsf{Emp} \Leftrightarrow a} \quad \frac{}{(p * p') \ltimes (q * q') \Leftrightarrow (p \ltimes q) * (p' \ltimes q')}$$

$$\frac{}{a * a' \Leftrightarrow a' * a} \quad \frac{a_1 \Rightarrow a_1' \quad a_2 \Rightarrow a_2'}{a_1 * a_2 \Rightarrow a_1' * a_2'} \quad \frac{p \Rightarrow p' \quad q \Rightarrow q'}{p \ltimes q \Rightarrow p' \ltimes q'}$$

Figure 9. Selected Proof Rules for Actions

the resulting states are also disjoint. Emp, True and Id are defined using these primitive actions, which represent empty transitions, arbitrary transitions and arbitrary identity transitions respectively. We use $[\![a]\!]$ to represent the set of transitions satisfying a, and use $(\sigma,\sigma') \models a$ and $(\sigma,\sigma') \in [\![a]\!]$ interchangeably in this paper.

In Fig. 9, we show some selected proof rules for actions, which are sound with respect to the semantics of actions. Many proof rules for Separation Logic assertions can also be ported here for actions. They are omitted due to space limits. Examples of actions are shown below. The following lemma shows the monotonicity of the action $a * \mathsf{Id}$:

Lemma 5.2 If $(\sigma_1,\sigma_2) \models a * \mathsf{Id}$, $\sigma_1' = \sigma_1 \uplus \sigma'$, and $\sigma_2' = \sigma_2 \uplus \sigma'$, then $(\sigma_1',\sigma_2') \models a * \mathsf{Id}$.

Stability. Next we introduce the concept of stability of an assertion p with respect to an action a.

Definition 5.3 (Stability) We say p is stable with respect to the action a, *i.e.,* $\mathsf{Sta}(p,a)$ holds, if and only if for all σ and σ', if $\sigma \models_{\mathrm{SL}} p$ and $(\sigma,\sigma') \models a$, then $\sigma' \models_{\mathrm{SL}} p$.

Informally, $\mathsf{Sta}(p,a)$ means the validity of p is preserved by transitions in $[\![a]\!]$. Examples of $\mathsf{Sta}(p,a)$ are shown below. Following RGSep (Vafeiadis and Parkinson 2007), the following lemma shows the encoding of stability using the septraction $p -\!\!\circledast q$.

Lemma 5.4 The following are true:

- $\mathsf{Sta}(r, p \ltimes q)$ if and only if $((p -\!\!\circledast r) \wedge \mathsf{emp}) * q \Rightarrow r$;
- If $(p -\!\!\circledast r) * q \Rightarrow r$, then $\mathsf{Sta}(r, p \ltimes q)$;
- $\mathsf{Sta}(r, (p \ltimes q) * \mathsf{Id})$ if and only if $(p -\!\!\circledast r) * q \Rightarrow r$.

The separating conjunction $a * a'$ over actions allows us to compose disjoint transitions into one. Naturally, we want the following property about stability to hold:

If $\mathsf{Sta}(p,a)$ *and* $\mathsf{Sta}(p',a')$*, then* $\mathsf{Sta}(p * p', a * a')$.

As explained in Sec. 3, this property is important to support local reasoning. Unfortunately, it is not true in general, as shown in the following example. The example also shows that we cannot get this property even with precise p and p'.

Example 5.5 Let $a \stackrel{\text{def}}{=} ([\ell_1 \mapsto n_1]) \vee ((\ell_2 \mapsto n_2) \ltimes (\ell_2 \mapsto n_2{+}1))$, $p \stackrel{\text{def}}{=} \ell_1 \mapsto n_1$, $a' \stackrel{\text{def}}{=} ([\ell_2 \mapsto n_2]) \vee ((\ell_1 \mapsto n_1) \ltimes (\ell_1 \mapsto n_1{+}1))$, and $p' \stackrel{\text{def}}{=} \ell_2 \mapsto n_2$, and suppose $\ell_1 \neq \ell_2$, we can prove $\mathsf{Sta}(p,a)$ and $\mathsf{Sta}(p',a')$, but $\mathsf{Sta}(p * p', a * a')$ does not hold.

Here is a counterexample. Let the heap h be $\{\ell_1 \rightsquigarrow n_1, \ell_2 \rightsquigarrow n_2\}$, and h' be $\{\ell_1 \rightsquigarrow n_1+1, \ell_2 \rightsquigarrow n_2+1\}$. Then, for any s, s' and i, we have $(s,i,h) \models_{\mathrm{SL}} p * p'$ and $((s,i,h),(s',i,h')) \models a * a'$, but $(s',i,h') \models_{\mathrm{SL}} p * p'$ does not hold. $\square$

$$\frac{}{I \rhd [I]} \qquad \frac{}{I \rhd (I \ltimes I)} \qquad \frac{I \rhd a \quad I \rhd a'}{I \rhd a \vee a'} \qquad \frac{I \rhd a \quad I' \rhd a'}{I * I' \rhd a * a'}$$

Figure 10. Selected Rules for Fence (Assuming $\mathsf{precise}(I)$)

To establish the property, we seem to need some concept of "precise actions", similar to the requirement of precise assertions in Separation Logic. However, precision alone cannot address our problem. The following example shows that even if we have $\mathsf{Sta}(p_1,(r_1 \ltimes r_1'))$ and $\mathsf{Sta}(p_2,(r_2 \ltimes r_2'))$ for precise p_1, r_1, r_1', p_2, r_2 and r_2', we do not necessarily have $\mathsf{Sta}(p_1 * p_2,(r_1 \ltimes r_1') * (r_2 \ltimes r_2'))$.

Example 5.6 Let $p_1 \stackrel{\text{def}}{=} \ell_1 \mapsto n_1$, $r_1 \stackrel{\text{def}}{=} \ell_2 \mapsto n_2$, $r_1' \stackrel{\text{def}}{=} \ell_2 \mapsto n_2{+}1$, $p_2 \stackrel{\text{def}}{=} r_1$, $r_2 \stackrel{\text{def}}{=} p_1$, $r_2' \stackrel{\text{def}}{=} \ell_1 \mapsto n_1{+}1$, and suppose $\ell_1 \neq \ell_2$. We know $\mathsf{Sta}(p_1,(r_1 \ltimes r_1'))$ and $\mathsf{Sta}(p_2,(r_2 \ltimes r_2'))$ are vacuously true, but $\mathsf{Sta}(p_1 * p_2,(r_1 \ltimes r_1') * (r_2 \ltimes r_2'))$ is false. □

The problem is, p and a may specify different resources even if $\mathsf{Sta}(p,a)$ holds. To address this issue, we introduce invariant-fenced actions and use an invariant to identify the specified resource.

Invariant-Fenced Actions. The following definition says an action a is fenced by a *precise* invariant I (represented as $I \rhd a$) if and only if a holds over identity transitions satisfying $[I]$, and I holds over the beginning and end states of transitions satisfying a.

Definition 5.7 (Fence) $I \rhd a$ holds iff $[I] \Rightarrow a$, $a \Rightarrow (I \ltimes I)$ and $\mathsf{precise}(I)$.

It is natural to ask a to hold over identity transitions so that stuttering steps of processes would also satisfy it. The second requirement is important to determine the boundary of transitions a and a' in $a * a'$: the boundary can be uniquely determined if a or a' is fenced by a precise invariant I.

Lemma 5.8 If $(\sigma_1 \uplus \sigma_2, \sigma') \models a * a'$, $\sigma_1 \models_{\text{SL}} I$ and $I \rhd a$, then there exist unique σ_1' and σ_2' such that $\sigma' = \sigma_1' \uplus \sigma_2'$, $(\sigma_1, \sigma_1') \models a$ and $(\sigma_2, \sigma_2') \models a'$.

From Lemma 5.8 we can derive the following frame property of the action $a * \mathsf{Id}$.

Corollary 5.9 If $(\sigma_1 \uplus \sigma_2, \sigma') \models a * \mathsf{Id}$, $\sigma_1 \models_{\text{SL}} I$ and $I \rhd a$, then there exists σ_1' such that $\sigma' = \sigma_1' \uplus \sigma_2$ and $(\sigma_1, \sigma_1') \in [\![a]\!]$.

Figure 10 shows some selected proof rules for the fencing relation. The following lemma shows that the property about stability discussed in the previous section holds given an action fenced by a precise invariant.

Lemma 5.10 If $\mathsf{Sta}(p,a)$, $\mathsf{Sta}(p',a')$, $p \Rightarrow I$ and $I \rhd a$, we have $\mathsf{Sta}(p * p', a * a')$.

Below we give two examples to show invariant fenced actions. In particular, Example 5.12 shows that asking I in $I \rhd a$ to be precise does not prevent the action a from changing the size of the resource.

Example 5.11 Let $I = \ell_1 \mapsto \_ * \ell_2 \mapsto \_$, $a_1 = [\ell_1 \mapsto X * \ell_2 \mapsto Y]$, $a_2 = ((\ell_1 \mapsto X * \ell_2 \mapsto Y) \wedge X > Y) \ltimes (\ell_1 \mapsto X - Y * \ell_2 \mapsto Y)$, and $a_3 = ((\ell_1 \mapsto X * \ell_2 \mapsto Y) \wedge X < Y) \ltimes (\ell_1 \mapsto X * \ell_2 \mapsto Y - X)$. We have $I \rhd a_1$, $I \rhd (a_1 \vee a_2)$, $I \rhd (a_1 \vee a_3)$, and $I \rhd (a_1 \vee a_2 \vee a_3)$, but not $I \rhd a_2$ or $I \rhd a_3$. □

Example 5.12 We define $\mathsf{List}(\ell,n)$ as a linked list pointed to by ℓ with length n:

$$\mathsf{List}(\ell,0) \stackrel{\text{def}}{=} \ell = 0 \wedge \mathsf{emp}$$
$$\mathsf{List}(\ell,n{+}1) \stackrel{\text{def}}{=} (\mathsf{emp_s} \wedge \ell \neq 0 \wedge (\ell \mapsto \_ * \ell{+}1 \mapsto \ell')) * \mathsf{List}(\ell',n)$$

Let $I = \exists n.\ \mathsf{List}(\ell,n)$, and $a = (\mathsf{List}(\ell,m) \wedge m \leq n) \ltimes (\mathsf{List}(\ell,n))$. We can prove that $I \rhd a$ holds. □

6. The LRG Logic

As in SAGL/RGSep, we also split program states into private and shared parts, but the partition is logical and we do not change our model of states defined in Fig. 4. Our logic ensures that each thread has exclusive access to its private resource. The rely/guarantee conditions only specify transitions over shared resources.

If a statement C only accesses the private resource, it can be verified as sequential programs using a set of sequential rules. The judgment for well-formed sequential programs is in the form of $\{p\}\ C\ \{q\}$. The rules are mostly standard Separation Logic rules except that program variables are treated as resources, following Parkinson et al. (2006). They are omitted due to space limits and can be found in the extended version of the paper (Feng 2008). Note that, to prove $\{p\}\ C\ \{q\}$, C cannot contain atomic statements and parallel compositions.

6.1 Rules for Concurrency Verification

If the statement C shares resources with its environment, we need to consider its interaction with the environment and verify it using the set of rules for concurrency, shown in Fig. 11. The judgment for well-formed statements C in a concurrent setting is in the form of $R; G; I \vdash \{p\}\ C\ \{q\}$. R and G are rely/guarantee conditions. They are fenced by the invariant I. p and q are pre- and post-conditions. R, G and I only specify shared resources, but p and q here specify the whole state. Unlike SAGL/RGSep, we do not distinguish private and shared resources syntactically in assertions. Instead, their boundary can be determined by the invariant I.

The ENV rule allows us to convert the judgment $\{p\}\ C\ \{q\}$ into the concurrent form. If C only accesses the private resources and is "well-behaved" sequentially, it is well-behaved in a concurrent setting where there is no resource sharing. Here the rely/guarantee conditions are Emp and the invariant is emp, showing the shared resource is empty. This rule itself is not very useful since it does not allow resource sharing, but a more interesting rule can be derived from this rule and the frame rule shown below.

The ATOMIC rule first requires that the state contain the resource used to evaluate B (as explained in Sec. 5, $B = B$ is no longer a tautology when variables are treated as resources). Since the execution of C cannot be interrupted by the environment, we can treat the whole state as a private resource and verify C using the sequential rules. Outside of the atomic block, p and q need to be stable with respect to $R * \mathsf{Id}$, R for the shared resource and Id for the private (*i.e.,* the environment does not touch the private resource). The transition $p \ltimes q$ consists of sub-transitions over shared and private resources. The one over shared needs to satisfy G, and the private one can be arbitrary (*i.e.,* True). The rule also requires that the shared resource be well-formed with respect to the invariant (*i.e.,* $p \vee q \Rightarrow I * \mathsf{true}$), and that R/G be fenced by I. To have a concise presentation, we use $\mathsf{Sta}(\{r,r'\},R)$ as a short hand for $\mathsf{Sta}(r,R) \wedge \mathsf{Sta}(r',R)$, and $I \rhd \{R,G\}$ for $(I \rhd R) \wedge (I \rhd G)$. Similar representations are used in the remaining part of the paper.

The P-SEQ rule for sequential composition is the same as in standard Rely-Guarantee reasoning and does not need explanation. In the rules P-WHILE and P-IF, we require that the resource needed to

$$\frac{\{p\}\,C\,\{q\}}{\mathsf{Emp};\ \mathsf{Emp};\ \mathsf{emp} \vdash \{p\}\,C\,\{q\}}\ (\textsc{env}) \qquad \frac{R;\ G;\ I \vdash \{p\}\,C_1\,\{q\} \quad R;\ G;\ I \vdash \{q\}\,C_2\,\{r\}}{R;\ G;\ I \vdash \{p\}\,C_1;C_2\,\{r\}}\ (\textsc{p-seq})$$

$$\frac{p \Rightarrow B = B \quad \{p \wedge B\}\,C\,\{q\} \quad \mathsf{Sta}(\{p,q\}, R * \mathsf{Id}) \quad p \ltimes q \Rightarrow G * \mathsf{True} \quad p \vee q \Rightarrow I * \mathsf{true} \quad I \triangleright \{R,G\}}{R;\ G;\ I \vdash \{p\}\,\mathbf{atomic}(B)\{C\}\,\{q\}}\ (\textsc{atomic})$$

$$\frac{p \Rightarrow (B = B) * I \quad R;\ G;\ I \vdash \{p \wedge B\}\,C\,\{p\}}{R;\ G;\ I \vdash \{p\}\,\mathbf{while}\ B\ \mathbf{do}\ C\,\{p \wedge \neg B\}}\ (\textsc{p-while}) \qquad \frac{p \Rightarrow (B = B) * I \quad R;\ G;\ I \vdash \{p \wedge B\}\,C_1\,\{q\} \quad R;\ G;\ I \vdash \{p \wedge \neg B\}\,C_2\,\{q\}}{R;\ G;\ I \vdash \{p\}\,\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\,\{q\}}\ (\textsc{p-if})$$

$$\frac{R \vee G_2;\ G_1;\ I \vdash \{p_1 * r\}\,C_1\,\{q_1 * r_1\} \quad R \vee G_1;\ G_2;\ I \vdash \{p_2 * r\}\,C_2\,\{q_2 * r_2\} \quad r \vee r_1 \vee r_2 \Rightarrow I \quad I \triangleright R}{R;\ G_1 \vee G_2;\ I \vdash \{p_1 * p_2 * r\}\,C_1 \parallel C_2\,\{q_1 * q_2 * (r_1 \wedge r_2)\}}\ (\textsc{par})$$

$$\frac{R;\ G;\ I \vdash \{p\}\,C\,\{q\} \quad \mathsf{Sta}(r, R' * \mathsf{Id}) \quad I' \triangleright \{R',G'\} \quad r \Rightarrow I' * \mathsf{true}}{R * R';\ G * G';\ I * I' \vdash \{p * r\}\,C\,\{q * r\}}\ (\textsc{frame}) \qquad \frac{R * R';\ G * G';\ I * I' \vdash \{p\}\,C\,\{q\} \quad I \triangleright \{R,G\}}{R;\ G;\ I \vdash \{p\}\,C\,\{q\}}\ (\textsc{hide})$$

$$\frac{\begin{array}{c} R;\ G;\ I \vdash \{p\}\,C\,\{q\} \\ X \text{ not free in } R, G, \text{ and } I \end{array}}{R;\ G;\ I \vdash \{\exists X.\, p\}\,C\,\{\exists X.\, q\}}\ (\textsc{p-ex}) \qquad \frac{\begin{array}{c} R;\ G;\ I \vdash \{p\}\,C\,\{q\} \\ R;\ G;\ I \vdash \{p'\}\,C\,\{q'\} \end{array}}{R;\ G;\ I \vdash \{p \wedge p'\}\,C\,\{q \wedge q'\}}\ (\textsc{p-conj}) \qquad \frac{\begin{array}{c} R;\ G;\ I \vdash \{p\}\,C\,\{q\} \\ R;\ G;\ I \vdash \{p'\}\,C\,\{q'\} \end{array}}{R;\ G;\ I \vdash \{p \vee p'\}\,C\,\{q \vee q'\}}\ (\textsc{p-disj})$$

$$\frac{p' \Rightarrow p \quad R' \Rightarrow R \quad G \Rightarrow G' \quad q \Rightarrow q' \quad R;\ G;\ I \vdash \{p\}\,C\,\{q\} \qquad p' \vee q' \Rightarrow I' * \mathsf{true} \quad I' \triangleright \{R',G'\}}{R';\ G';\ I' \vdash \{p'\}\,C\,\{q'\}}\ (\textsc{csq})$$

Figure 11. Inference Rules for Concurrency

evaluate B be available in p but disjoint with the shared resource in I, *i.e.,* it is in the private part. Therefore, the validity of B would not be affected by the environment.

The PAR rule is similar to the one in RGSep shown in Sec. 2.3. The parent thread distribute p_1 and p_2 to the children C_1 and C_2 respectively as their private resources. The resource r is shared by them. We require that r, r_1 and r_2 imply I, *i.e.,* the shared resource is well-formed. Also R needs to be fenced by I.

The FRAME rule allows us to verify C with local specifications, and reuse it in contexts where some extra resource r (*i.e.,* the frame) is used. The frame r contains both private and shared parts. Since C does not access it, the validity of r is preserved at the end as long as r is stable with respect to $R' * \mathsf{Id}$, R' for the shared part and Id for the private. We also require that R' and G' be fenced by the (precise) invariant I', and that the shared part in the frame satisfy I'. Here G' is the thread's guaranteed transition over the extra shared part. Since G' is fenced by I', we know the identity transition made by C over r indeed satisfies G'. This frame rule is more general than the two frame rules in Sec. 3 for shared and private resources. As we will explain later, they can be derived from this rule.

If C knows that the part of the shared resources specified by R', G' and I' is actually not accessed by the outside world, it can leave this part unspecified by applying the HIDE rule. The HIDE rule is similar to its prototype shown in Sec. 3. Note that we do not use two assertions for private and shared resources respectively and use the invariant to determine their boundary instead, therefore changing the invariant from $I * I'$ to I introduces an implicit conversion of resources from shared to private. This conversion is explicit in the prototyping rule in Sec. 3. The advantage of not using two assertions is that we can easily share information in the specifications for private and shared resources. As usual, the hiding rule also requires R and G be fenced by the precise invariant I.

As mentioned in Sec. 3, a thread cannot abuse the freedom provided by the HIDE rule by hiding the resources that are indeed shared. The inappropriate hiding can be detected at the time of the parallel composition. From the PAR rule we can see that the private resource p_1 of C_1 needs to be composed linearly using the separating conjunction with both the private (p_2) and the shared (r) resources used by C_2. If C_1 cheats by converting part of r into p_1 using the HIDE rule, the linearity would be broken and the precondition after parallel composition would be unsatisfiable.

The P-EX rule introduces existential quantification over specifications. The conjunction rule (P-CONJ) is sound in LRG. The P-DISJ rule is a standard disjunction rule. The consequence rule (CSQ) allows adaptations of different part of the specifications.

It is important to note that, like RGSep, we do not have concurrency rules for primitive statements, therefore they either only access the private resource or access the shared part inside the atomic block (where the shared resource has been converted into private).

Derived Rules. In Fig. 12, we show several useful rules that can be derived from the basic set of rules. The ENV-SHARE rule is similar to the ENV rule in Fig. 11, but allows resource sharing with the environment. It is derived from the ENV rule and the FRAME rule.

The rules FR-PRIVATE and FR-SHARE are frame rules for private and shared resources respectively, similar to those shown in Sec. 3. They are derived from the FRAME rule. To get FR-PRIVATE, we simply instantiate R' and G' with Emp and I' with emp in the FRAME rule. The FR-SHARE rule is similar to the FRAME rule, except r contains only shared resource.

The PAR-HIDE rule is a generalization of the PAR rule. The parent thread has private resource $p_1 * p_2 * m$ and shares the resource r with its environments. p_1 and p_2 are distributed to C_1 and C_2 respectively as their private resources. m and r are shared by them. The guarantees about the use of m by the two processes are G'_1 and G'_2 respectively, which are fenced by I'. Since m is private resource of the parent thread, the sharing between children threads does not need to be exposed to the environments. Thus R, G_1 and G_2 only specify transitions over the resource specified by r. They are fenced by I. Here we also require that r, r' and r'' all imply I; and that m, m' and m'' all imply I'. To derive the PAR-HIDE rule, we first apply the PAR rule, and then apply the HIDE rule to convert m to private

$$\frac{\{p\}\,C\,\{q\} \quad \mathsf{Sta}(r, R*\mathsf{Id}) \quad I \triangleright \{R,G\} \quad r \Rightarrow I*\mathsf{true}}{R;\ G;\ I \vdash \{p*r\}\,C\,\{q*r\}}\ (\textsc{env-share})$$

$$\frac{R;\ G;\ I \vdash \{p\}\,C\,\{q\}}{R;\ G;\ I \vdash \{p*r\}\,C\,\{q*r\}}\ (\textsc{fr-private}) \qquad \frac{R;\ G;\ I \vdash \{p\}\,C\,\{q\} \quad \mathsf{Sta}(r, R') \quad I' \triangleright \{R',G'\} \quad r \Rightarrow I'}{R*R';\ G*G';\ I*I' \vdash \{p*r\}\,C\,\{q*r\}}\ (\textsc{fr-share})$$

$$\frac{\begin{array}{c}(R \vee G_2)*G_2';\ G_1*G_1';\ I*I' \vdash \{p_1*m*r\}\,C_1\,\{q_1*m_1'*r_1'\}\\ (R \vee G_1)*G_1';\ G_2*G_2';\ I*I' \vdash \{p_2*m*r\}\,C_2\,\{q_2*m_2'*r_2'\}\\ I \triangleright \{R,G_1,G_2\} \quad I' \triangleright \{G_1',G_2'\} \quad r \vee r_1' \vee r_2' \Rightarrow I \quad m \vee m_1' \vee m_2' \Rightarrow I'\end{array}}{R;\ G_1 \vee G_2;\ I \vdash \{p_1*p_2*m*r\}\,C_1 \parallel C_2\,\{q_1*q_2*(m_1' \wedge m_2')*(r_1' \wedge r_2')\}}\ (\textsc{par-hide})$$

Figure 12. Useful Derived Rules

and to hide G_1' and G_2'. The derivation is shown in the extended version (Feng 2008).

6.2 Semantics and Soundness

The semantics for the judgment $\{p\}\ C\ \{q\}$ is standard, and the soundness of sequential rules is formalized and proved following the standard way established in previous works on sequential Separation Logic (Yang and O'Hearn 2002).

Definition 6.1 $\models \{p\}\,C\,\{q\}$ iff, for any σ such that $\sigma \models_{\mathrm{SL}} p$, $(C,\sigma) \not\rightsquigarrow^*$ **abort**, and, if $(C,\sigma) \rightsquigarrow^* (\mathbf{skip},\sigma')$, then $\sigma' \models_{\mathrm{SL}} q$.

Lemma 6.2 (Seq-Soundness) If $\{p\}\ C\ \{q\}$, then $\models \{p\}\ C\ \{q\}$.

Before we define the semantics for the judgment $R;\ G;\ I \vdash \{p\}\ C\ \{q\}$, we introduce the non-interference property.

Definition 6.3 (Non-Interference) $(C,\sigma,\mathcal{R}) \Longrightarrow^0 \mathcal{G}$ always holds; $(C,\sigma,\mathcal{R}) \Longrightarrow^{n+1} \mathcal{G}$ holds iff $(C,\sigma) \not\rightsquigarrow$ **abort**, and,

(1) for all σ', if $(\sigma,\sigma') \in \mathcal{R}$, then for all $k \le n$, $(C,\sigma',\mathcal{R}) \Longrightarrow^k \mathcal{G}$;

(2) for all σ', if $(C,\sigma) \rightsquigarrow (C',\sigma')$, then $(\sigma,\sigma') \in \mathcal{G}$ and $(C',\sigma',\mathcal{R}) \Longrightarrow^k \mathcal{G}$ holds for all $k \le n$.

So $(C,\sigma,\mathcal{R}) \Longrightarrow^n \mathcal{G}$ means, starting from the state σ, C does not interfere with the environment's transitions in $\mathcal{R}$ up to n steps, and transitions made by C are in $\mathcal{G}$. It also implies the parallel execution of C does not abort within n steps, as the following lemma shows.

Lemma 6.4 If $(C,\sigma,\mathcal{R}) \Longrightarrow^n \mathcal{G}$, there does not exist j such that $j < n$ and $(C,\sigma) \overset{\mathcal{R}}{\longmapsto}{}^j$ **abort**.

The semantics of $R;\ G;\ I \vdash \{p\}\ C\ \{q\}$ is defined below. Theorem 6.6 shows the soundness of the logic.

Definition 6.5 $R;\ G;\ I \models \{p\}\ C\ \{q\}$ iff, for all σ such that $\sigma \models_{\mathrm{SL}} p$, the following are true (where $\mathcal{R} = [\![R*\mathsf{Id}]\!]$ and $\mathcal{G} = [\![G*\mathsf{True}]\!]$):

(1) if $(C,\sigma) \overset{\mathcal{R}}{\longmapsto}{}^* (\mathbf{skip},\sigma')$, then $\sigma' \models_{\mathrm{SL}} q$;

(2) for all n, $(C,\sigma,\mathcal{R}) \Longrightarrow^n \mathcal{G}$.

Theorem 6.6 (Soundness)
If $R;\ G;\ I \vdash \{p\}\ C\ \{q\}$, then $R;\ G;\ I \models \{p\}\ C\ \{q\}$.

To prove the soundness, we first prove the following properties about the syntactic judgment.

$\boxed{O \Vdash (\mathtt{x} = X) \wedge (\mathtt{x} \mapsto M, N)}$

$\langle$t11 := [x]$\rangle$;	$\langle$t21 := [x+1]$\rangle$;
$\langle$t12 := [x+1]$\rangle$;	$\langle$t22 := [x]$\rangle$;
while (t11 ≠ t12) **do**{	**while** (t21 ≠ t22) **do**{
if(t11 > t12) **then** {	**if**(t21 > t22) **then** {
t11 := t11 − t12;	t21 := t21 − t22;
$\langle$[x] := t11$\rangle$;	$\langle$[x+1] := t21$\rangle$;
}	}
$\langle$t12 := [x+1]$\rangle$;	$\langle$t22 := [x]$\rangle$;
}	}

$\boxed{O \Vdash \exists U.\,(\mathtt{x} = X) \wedge (\mathtt{x} \mapsto U, U) \wedge (U = gcd(M,N))}$

where $O = \mathtt{x}, \mathtt{t11}, \mathtt{t12}, \mathtt{t21}, \mathtt{t22}, \bullet$

$R \overset{\text{def}}{=} \mathsf{Emp} \qquad G \overset{\text{def}}{=} \mathsf{Emp} \qquad I \overset{\text{def}}{=} \mathsf{emp}$

Figure 13. Example: Verification of Concurrent GCD

Lemma 6.7 If $R;\ G;\ I \vdash \{p\}\ C\ \{q\}$, then $I \triangleright \{R,G\}$ and $p \vee q \Rightarrow I*\mathsf{true}$.

Proof. By induction over the derivation of $R;\ G;\ I \vdash \{p\}\ C\ \{q\}$. □

As in sequential Separation Logic, the locality property (Yang and O'Hearn 2002; Calcagno et al. 2007a) of primitive statements is essential to prove the soundness. In addition, in the concurrent setting, we need similar properties about the environment's behavior. Lemma 5.2 and Corollary 5.9 show the monotonic property and the frame property of $R*\mathsf{Id}$.

Theorem 6.6 is proved by induction over the derivation of $R;\ G;\ I \vdash \{p\}\ C\ \{q\}$. We show some main lemmas used in the proof in Appendix A. More details can be found in the extended version of the paper (Feng 2008).

7. Examples

In this section we show how the programs in Figs. 1 and 2 can be verified modularly using the LRG logic. Although the example is very simple and may be a bit contrived, it is very representative in that it involves both fine-grained concurrency and lock-based protection of resources, it creates children threads that share dynamically allocated resources, and it requires both local reasoning and information hiding.

7.1 Concurrent GCD

We first show the verification of the concurrent GCD program using local specifications. We show the program and the specifications in Fig. 13. The program is the same as in Fig. 2.

In the example, the parent thread forks two threads. The first one (the one on the left) reads the values [x] and [x+1], but only

$$
\begin{aligned}
p_1 &\stackrel{\text{def}}{=} \mathtt{x} \Vdash \mathtt{x} = X \wedge (\mathtt{x} \mapsto Y, Z) \\
p_2 &\stackrel{\text{def}}{=} \mathtt{x} \Vdash \mathtt{x} = X \wedge (\mathtt{x} \mapsto Y, Z) \wedge Y < Z \\
p_2' &\stackrel{\text{def}}{=} \exists Z'.\, \mathtt{x} \Vdash \mathtt{x} = X \wedge (\mathtt{x} \mapsto Y, Z') \wedge Z' < Z \wedge gcd(Y, Z) = gcd(Y, Z') \\
p_3 &\stackrel{\text{def}}{=} \mathtt{x} \Vdash \mathtt{x} = X \wedge (\mathtt{x} \mapsto Y, Z) \wedge Y > Z \\
p_3' &\stackrel{\text{def}}{=} \exists Y'.\, \mathtt{x} \Vdash \mathtt{x} = X \wedge (\mathtt{x} \mapsto Y', Z) \wedge Y' < Y \wedge gcd(Y, Z) = gcd(Y', Z) \\
R_1 &\stackrel{\text{def}}{=} (p_1 \ltimes p_1) \vee (p_2 \ltimes p_2') \qquad G_1 \stackrel{\text{def}}{=} (p_1 \ltimes p_1) \vee (p_3 \ltimes p_3') \\
R_2 &\stackrel{\text{def}}{=} G_1 \qquad G_2 \stackrel{\text{def}}{=} R_1 \\
I' &\stackrel{\text{def}}{=} \mathtt{x} \Vdash \mathtt{x} \mapsto \_ * \mathtt{x+1} \mapsto \_
\end{aligned}
$$

$$
\begin{aligned}
r_{10} &\stackrel{\text{def}}{=} \exists Z.\, \mathtt{x} \Vdash \mathtt{x} = X \wedge (\mathtt{x} \mapsto M, Z) \wedge gcd(M, Z) = gcd(M, N) \\
p_{10} &\stackrel{\text{def}}{=} (\mathtt{t11}, \mathtt{t12} \Vdash \mathsf{emp_h}) * r_{10} \\
p_{11} &\stackrel{\text{def}}{=} (\mathtt{t11}, \mathtt{t12} \Vdash \mathtt{t11} = M) * r_{10} \\
r_{12} &\stackrel{\text{def}}{=} \exists Z'.\ \mathtt{x} \Vdash \mathtt{x} = X \wedge (\mathtt{x} \mapsto Y, Z') \wedge (Z \geq Z') \\
&\qquad \wedge (Y \geq Z \Rightarrow Z = Z') \wedge gcd(Y, Z') = gcd(M, N) \\
p_{12} &\stackrel{\text{def}}{=} \exists Y, Z.\, (\mathtt{t11}, \mathtt{t12} \Vdash \mathtt{t11} = Y \wedge \mathtt{t12} = Z) * r_{12} \\
p_{13} &\stackrel{\text{def}}{=} p_{12} \\
p_{14} &\stackrel{\text{def}}{=} \exists Y, Z.\, (\mathtt{t11}, \mathtt{t12} \Vdash \mathtt{t11} = Y \wedge \mathtt{t12} = Z \wedge Y > Z) * r_{12} \\
p_{15} &\stackrel{\text{def}}{=} \exists Y, Z.\, (\mathtt{t11}, \mathtt{t12} \Vdash \mathtt{t11} = (Y - Z) \wedge \mathtt{t12} = Z \wedge Y > Z) * r_{12} \\
p_{16} &\stackrel{\text{def}}{=} p_{12} \qquad p_{17} \stackrel{\text{def}}{=} p_{12} \qquad p_{18} \stackrel{\text{def}}{=} p_{12} \\
p_{19} &\stackrel{\text{def}}{=} \exists Y, Z.\, (\mathtt{t11}, \mathtt{t12} \Vdash \mathtt{t11} = Y \wedge \mathtt{t12} = Z \wedge Y = Z) * r_{12}
\end{aligned}
$$

Figure 14. Spec. and Intermediate Assertions for the First Thread

updates `[x]` if `[x]` > `[x + 1]`. The second one (the one on the right) does the reverse. The variable `x` is shared by both threads. `t11`, `t12`, `t21` and `t22` are temporary variables used exclusively in one of the threads. We use $\langle C \rangle$ as the syntactic sugar for **atomic**(true)$\{C\}$. In this fine-grained concurrent program, we only put basic memory loads and stores into atomic blocks.

Figure 13 shows in boxes the pre-condition before forking the two threads and the post-condition after their join. Recall that the assertion $O \Vdash p$ is defined in Fig. 7. The parent thread owns the variables and the memory cells at locations `x` and `x+1` as private resources. Here we use $\mathtt{x} \mapsto M, N$ as a short hand for $\mathtt{x} \mapsto M * \mathtt{x+1} \mapsto N$. At the end, we know the value of `x` is preserved, and values of `[x]` and `[x+1]` are the GCD of their initial values. Recall that capital variables are auxiliary logical variables. The shared resource of the parent thread is empty. Its R and G are simply Emp. The invariant fencing them is emp.

To verify the parent thread, we need to first apply the PAR-HIDE rule shown in Fig. 12. Temporaries (`t11`, `t12`, `t21`, `t22`) are distributed to the children as their private resources. The variable `x` and the memory cells pointed to by `x` are shared by them. The precondition p_{10} for the first thread is specified in Fig. 14, where r_{10} specifies the shared resource. Because of the symmetry between the first and the second threads, we elide specifications for the second thread.

The rely and guarantee conditions of children threads are shown in Fig. 14. R_1 is the first thread's assumption about the behavior of its environment containing the second thread. It says the environment preserves the value of the shared variable `x`, and it either preserves the value stored at `x` and `x+1`, or decrease the value at `x+1` if its original value is bigger than the value at `x`, but the GCD of new values is the same as the GCD of original values. The guarantee G_1 is similar. Because of the symmetry, we use G_2 and R_2 are simply R_1 and G_1. We use I' to fence them. It is easy to see that I' is precise and $I' \triangleright \{R_1, R_2, G_1, G_2\}$ holds.

In Fig. 14, we present all the intermediate assertions as a proof sketch for the first thread. The assertion p_{1k} is the post-condition

$$
\begin{aligned}
I &\stackrel{\text{def}}{=} \mathsf{emp_s} \wedge \exists X.\, (\mathtt{lhead} \mapsto X) * (X = 1 \wedge r \vee X = 0 \wedge \mathsf{emp_h}) \\
r &\stackrel{\text{def}}{=} \exists \ell.\, (\mathtt{lheap{+}1} \mapsto \ell) * List(\ell) \\
List(\ell) &\stackrel{\text{def}}{=} (\ell = 0 \wedge \mathsf{emp_h}) \vee (\ell \neq 0 \wedge \exists \ell'.\, (\ell \mapsto \_, \_, \ell') * List(\ell')) \\
R &\stackrel{\text{def}}{=} I \ltimes I \qquad\qquad G \stackrel{\text{def}}{=} I \ltimes I
\end{aligned}
$$

$\boxed{(O \Vdash \mathsf{emp_h}) * I}$

1 `nd := 0;`

$\boxed{(O \Vdash \mathtt{nd} = 0 \wedge \mathsf{emp_h} \vee (\mathtt{nd} \mapsto \_, \_, \_)) * I}$

2 **while** (`nd` = 0) **do** {

3 `lk := 0;`

$\boxed{(O \Vdash \mathtt{lk} = 0 \wedge \mathsf{emp_h} \vee \mathtt{lk} = 1 \wedge r) * I}$

4 **while** ($\mathtt{lk} \neq 1$) **do** {

5 ⟨`lk := [lhead]`; **if** (`lk` = 1) **then** `[lhead] := 0;`⟩

6 }

$\boxed{(O \Vdash r) * I}$

7 `nd := [lhead + 1];`

8 **if** ($\mathtt{nd} \neq 0$) **then** {

9 `tmp := [nd + 2]; [lhead + 1] := tmp`

10 }

$\boxed{(O \Vdash (\mathtt{nd} = 0 \wedge \mathsf{emp_h} \vee (\mathtt{nd} \mapsto \_, \_, \_)) * r) * I}$

11 ⟨`[lhead] := 1`⟩;

$\boxed{(O \Vdash \mathtt{nd} = 0 \wedge \mathsf{emp_h} \vee (\mathtt{nd} \mapsto \_, \_, \_)) * I}$

12 }

$\boxed{(O \Vdash (\mathtt{nd} \mapsto \_, \_, \_)) * I}$

13 `tmp := [nd]; tmp' := [nd+1]; x :=` **cons**(`tmp,tmp'`);

$\boxed{(O \Vdash \exists M, N.\, (\mathtt{nd} \mapsto M, N, \_) * (\mathtt{x} \mapsto M, N)) * I}$

14 C_{gcd}

$\boxed{(O \Vdash \exists M, N, U.\, (\mathtt{nd} \mapsto M, N, \_) * (\mathtt{x} \mapsto U, U) \wedge gcd(M, N) = U) * I}$

Figure 15. Example: GCD of Nodes on List

following the k-th line. The sub-assertion r_{1j} specifies the shared resource. It is important to note that R_1 and G_1 specify the change and preservation of values in memory, which are crucial to verify the partial correctness. For instance, in p_{11} we know the value of `t11` is consistent with the value at the memory location `x` because R_1 ensures the environment does not update `[x]`. Similarly, we can derive the relationship between the value of `t12` and the value at the memory location `x+1` in p_{12}.

We omit details of the verification of the child thread, which have been shown several times before to illustrate Rely-Guarantee reasoning (Yu and Shao 2004; Feng and Shao 2005; Feng et al. 2007). What is new here is our specification for the parent thread, where the local sharing of memory cells at `x` and `x+1` is hidden from the environment since R, G and I are simply empty.

7.2 Verification of the Thread in Fig. 1

We show the program again in Fig. 15 with specifications and intermediate assertions. The invariant I specifies the well-formedness of the shared resource (recall its structure is illustrated in Sec. 2.4). The rely and guarantee conditions are simply $I \ltimes I$.

The verification of lines 1–13 simply applies the technique for ownership transfer in CSL (O'Hearn 2007). Similar examples have been shown in Feng et al. (2007) and Vafeiadis and Parkinson (2007). Here we show some important intermediate assertions to demonstrate the sketch of the proof and do not explain the details. In each assertion, O specifies the ownership of variables used in the thread and its definition is omitted. The shared resource is always specified by I.

The assertion following Line 13 shows that the allocated memory block at the location x is treated as private resource, so it does not affect our specification of R and G. The pre- and post-conditions for Line 14 (C_{gcd}) are different from the local specifications given in Fig. 13. To reuse our proof for C_{gcd} in the previous section, we can prove it also satisfies the new specification by applying the FRAME rule.

8. Related Work and Conclusions

Rely-Guarantee reasoning has been a well-studied method since it was proposed by Jones (Jones 1983). A comprehensive survey of related works can be found in the book by de Roever et al. (2001). Most of the works, however, have the same compositionality problems explained in the beginning of this paper.

Reynolds *et al.* (Reynolds 2002; Ishtiaq and O'Hearn 2001) proposed Separation Logic for modular verification of sequential programs. O'Hearn has applied the ideas of local reasoning and ownership transfers in Concurrent Separation Logic (CSL) for concurrency verification (O'Hearn 2007). CSL is modular, but the limited expressiveness of the program invariants I makes it difficult to reason about fine-grained concurrency.

This paper extends recent works on SAGL (Feng et al. 2007) and RGSep (Vafeiadis and Parkinson 2007) that have tried to combine merits of both Rely-Guarantee reasoning and Separation Logic. Many technical details are borrowed directly from RGSep, such as the use of the global atomic block and the combination of small-step and big-step operational semantics to model atomicity, but the differences between our work and SAGL/RGSep are also substantial. We define the separating conjunction of rely/guarantee conditions, and introduce a frame rule and hiding rule in the logic. These extensions allow us to support local specifications of rely/guarantee conditions, to hide locally shared resources from global specifications, and to support sharing of dynamically created resources, which are all open problems unsolved in SAGL/RGSep.

One more important improvement over SAGL/RGSep is our assertion language for pre- and post-conditions. SAGL uses two Separation Logic assertions to specify private and shared resources respectively. It is difficult for them to share information. RGSep uses two level logics to address this problem. Shared resources are specified using boxed assertions in a new logic built over Separation Logic assertions. In LRG, all we need is just Separation Logic assertions. We do not need to specify private and shared resources separately. The boundary is interpreted logically by the asserter and is fenced by I. On the other hand, SAGL and RGSep do not need I.

Our model of program states is also different from RGSep. In RGSep the partition of private and shared resources is made physically in the program states. We do not follow this approach for several reasons. First, it is somewhat inconsistent with the philosophy of "ownership is in the eye of the asserter" (O'Hearn et al. 2004; O'Hearn 2007), that is, the change of the boundary between resources is purely logical and there should be no operational effects. Technically, it makes the operational semantics depend on the assertion language because atomic blocks need to be annotated with precise post-conditions to decide the new physical boundary of resources at the exit. In addition, environments that are cooperative in this model might be ill-behaved in the traditional thread model with shared address spaces. Although this would not affect the soundness over closed programs in both models, the soundness of RGSep over programs with open environments does not hold in the traditional thread model.

Vafeiadis (2007) extends RGSep with multiple atomic blocks for multiple regions of shared resources. He also supports local rely and guarantee conditions that specify only individual regions. However, the pre-/post and rely/guarantee conditions for different regions need to be distinguished syntactically using the names of regions, so the assertion language is even more complex than in RGSep. The partition of regions is also done physically in program states. This allows him to avoid the problems shown in Examples 5.5 and 5.6, but has the same limitations as in RGSep explained above. We use the separating conjunction of actions to model subtransitions over disjoint resources. There is no need of multiple atomic blocks and physical regions. Regions in LRG are implicit, whose boundaries are determined logically by the resource invariants. The associativity of separating conjunction allows us to flexibly merge and split regions.

Our work can also be viewed as an extension of CSL with the more expressive rely/guarantee style specifications, but without sacrificing its modularity. Although not proved in this paper, we believe CSL can be shown as a specialized version of LRG in the sense that, given a CSL judgment $\{p\}C\{q\}$, we can prove $I' \ltimes I'$; $I' \ltimes I'$; $I' \vdash \{p * I'\}\ C\ \{q * I'\}$ in LRG. I' is in the form of "(in **atomic**)$\wedge$$\mathsf{emp}$$\vee$ (not in **atomic**)$\wedge$$I$", and I is the invariant about the resource protected by the **atomic** block.

Bornat et al. (2005) extend CSL with permission accounting, where the fractional permissions less than 1 represent read-only permissions. For simplicity, we do not support fractional permissions. Rely/guarantee conditions seem to have similar expressiveness to say some data is read-only. On the other hand, adding fractional permissions to LRG may further simplify specifications of rely/guarantee conditions. It would be interesting future work to study their relationships.

Yang (2007) proposes a relational separation logic to verify equivalence of programs. He also uses assertions over a pair of program states and defines separating conjunction over these assertions, but his assertions are used for very different purposes — instead of specifying state transitions, they are used to relate program states in different programs. Benton (2006) has similar definition of separating conjunction of binary relations, which is used to reason about program equivalence instead of modeling state transitions. Benton uses accessibility maps to determine the boundaries of regions of heap, similar to our use of invariants to fence rely/guarantee conditions. Dynamic Frames are also used (Kassios 2006) for similar purposes.

The major limitation of LRG is the requirement of precise resource invariants, which might be too restrictive to reason about programs that leak shared resources. In particular, there are simple lock-free algorithms that intentionally introduce memory leaks to avoid ABA problems (Herlihy and Shavit 2008). Their correctness depends on the existence of garbage collectors (GC). We may not be able to verify them using LRG. On the other hand, these algorithms can be instrumented using GC-independent techniques, *e.g.,* hazard pointers (Michael 2004). We believe the instrumented algorithms can be verified using LRG, and will test our hypothesis in our future work.

CSL has the similar restriction to precise invariants, but it can be relaxed by using *supported* assertions as invariants and using *intuitionistic* assertions to specify private resources (O'Hearn et al. 2004; Brookes 2007). It is unclear if we can have the same relaxation. The difficulty is caused by the asymmetric extensions of R (to $R{*}\mathsf{Id}$) and G (to $G{*}\mathsf{True}$) in Definition 6.5 and in the ATOMIC rule. Suppose we have threads T_1 and T_2. The Id in T_1's rely condition $R_1{*}\mathsf{Id}$ specifies the inaccessibility of T_1's private resources by the environment. The True in T_2's guarantee $G_2{*}\mathsf{True}$ specifies T_2's exclusive access of its private resources. Therefore, to ensure the non-interference, G_2 and R_1 must have a uniform view of the shared resources, which is enforced by a precise invariant. A supported invariant is not sufficient for this purpose. The asymmetric treatment of R and G results from our attempt to eliminate explicit distinctions between shared and private resources. Since RGSep and SAGL have the explicit distinctions, they do not need a precise view of shared resources.

Another limitation of LRG is that it only supports the verification of safety properties (including partial correctness). Gotsman et al. (2009) extend RGSep to reason about certain liveness properties of non-blocking algorithms. It would be interesting to see if it is possible to extend LRG following similar approaches. Also, we do not discuss the issues about automated verification in this paper. Many works have been done to automate the Rely-Guarantee based verification, *e.g.,* Flanagan et al. (2005) and Calcagno et al. (2007b). We would like to combine these techniques with the new LRG logic in the future.

In summary, we propose the LRG logic in this paper for local Rely-Guarantee reasoning. Introducing separating conjunction of actions allows us to borrow the techniques developed in Separation Logic for local reasoning. Our LRG logic, for the first time, supports a frame rule over rely/guarantee conditions and a hiding rule for hiding the local sharing of resources from the outside world. These rules allow us to write local rely/guarantee conditions and improve the reusability of verified program modules.

Acknowledgments

I would like to thank Matthew Parkinson for the inspiring discussions and suggestions. In particular, Matthew suggested to add the HIDE rule and showed that the PAR-HIDE rule, which was a built-in rule in an earlier version of the paper, could be derived from the HIDE rule and the PAR rule. Thanks to Viktor Vafeiadis, Zhong Shao, and anonymous referees for their suggestions and comments on earlier versions of this paper.

References

Nick Benton. Abstracting allocation : The new new thing. In *Proc. Computer Science Logic (CSL'06)*, volume 4207 of *Lecture Notes in Computer Science*, pages 182–196. Springer, September 2006.

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *Proc. 32nd ACM Symp. on Principles of Prog. Lang. (POPL'05)*, pages 259–270. ACM Press, January 2005.

Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proc. 22nd Annual IEEE Symposium on Logic in Computer Science (LICS'07)*, pages 366–378. IEEE Computer Society, July 2007a.

Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *Proc. 14th Int'l Symposium on Static Analysis (SAS'07)*, volume 4634 of *Lecture Notes in Computer Science*, pages 233–248. Springer, August 2007b.

Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency verification: introduction to compositional and noncompositional methods*. Cambridge University Press, 2001.

Xinyu Feng. Local rely-guarantee reasoning (extended version). Technical Report TTIC-TR-2008-1, Toyota Technological Institute at Chicago, Chicago, IL, U.S.A., October 2008. `http://www.tti-c.org/technical_reports/ttic-tr-2008-1.pdf`.

Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. 2005 ACM Int'l Conf. on Functional Prog. (ICFP'05)*, pages 254–267. ACM Press, September 2005.

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proc. 16th European Symp. on Prog. (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, March 2007.

Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.

Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *Proc. 36th ACM Symp. on Principles of Prog. Lang. (POPL'09)*, page to appear. ACM Press, January 2009.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, March 2008.

Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symp. on Principles of Prog. Lang. (POPL'01)*, pages 14–26. ACM Press, January 2001.

Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

Cliff B. Jones. Wanted: a compositional approach to concurrency. In *Programming Methodology*, pages 5–15. Springer-Verlag, 2003.

Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proc. 14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.

Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15 (6):491–504, 2004.

Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. 31th ACM Symp. on Principles of Prog. Lang. (POPL'04)*, pages 268–280. ACM Press, January 2004.

Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in hoare logics. In *Proc. 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 137–146. IEEE Computer Society, August 2006.

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In *Proc. 34th ACM Symp. on Principles of Prog. Lang. (POPL'07)*, pages 297–302. ACM Press, January 2007.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE Computer Society, July 2002.

Viktor Vafeiadis. *Modular Fine-Grained Concurrency Verification*. PhD thesis, University of Cambridge, July 2007.

Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. 18th Int'l Conf. on Concurrency Theory (CONCUR'07)*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271, September 2007.

Hongseok Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3): 308–334, 2007.

Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *Proc. 5th Int'l Conf. on Foundations of Software Science and Computation Structures (FoSSaCS'02)*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer, April 2002.

Dachuan Yu and Zhong Shao. Verification of safety properties for concurrent assembly code. In *Proc. 2004 ACM Int'l Conf. on Functional Prog. (ICFP'04)*, pages 175–188. ACM Press, September 2004.

A. Soundness Proof for LRG

Theorem 6.6 is proved by induction over the derivation of the judgment $R; G; I \vdash \{p\}\, C\, \{q\}$. The whole proof consists of the soundness proof for each individual rules. Here we show the main lemmas used to prove the soundness of FRAME, HIDE and PAR. More details are shown in the extended version (Feng 2008).

A.1 Soundness of the FRAME rule

Suppose the FRAME rule is applied to derive $R * R'$; $G * G'$; $I * I' \vdash \{p * r\}\, C\, \{q * r\}$. We want to prove $R * R'$; $G * G'$; $I * I' \models \{p * r\}\, C\, \{q * r\}$. By inversion of the FRAME rule we know R; G; $I \vdash \{p\}\, C\, \{q\}$, $\mathsf{Sta}(r, R' * \mathsf{Id})$, $r \Rightarrow I' * \mathsf{true}$, and $I' \triangleright \{R', G'\}$ hold. By the induction hypothesis (of Theorem 6.6), we know R; G; $I \models \{p\}\, C\, \{q\}$. Also, by Lemma 6.7 we know $I \triangleright \{R, G\}$ and $p \Rightarrow I * \mathsf{true}$.

Let $\mathcal{R} = [\![R * \mathsf{Id}]\!]$, $\mathcal{R}' = [\![R * R' * \mathsf{Id}]\!]$, $\mathcal{G} = [\![G * \mathsf{True}]\!]$, and $\mathcal{G}' = [\![G * G' * \mathsf{True}]\!]$. We can prove the following Lemmas A.1 and A.2. Soundness of the FRAME rule is shown in A.3.

Lemma A.1 For all n, C, σ_1, σ_2 and σ', if $\sigma_1 \models_{\mathrm{SL}} I * \mathsf{true}$, $\sigma_2 \models_{\mathrm{SL}} r$, $(C, \sigma_1, \mathcal{R}) \Longrightarrow^n \mathcal{G}$, and $(C, \sigma_1 \uplus \sigma_2) \overset{\mathcal{R}'}{\longmapsto}{}^n (\mathbf{skip}, \sigma')$, then there exist σ'_1 and σ'_2 such that $\sigma' = \sigma'_1 \uplus \sigma'_2$, $\sigma'_2 \models_{\mathrm{SL}} r$, and $(C, \sigma_1) \overset{\mathcal{R}}{\longmapsto}{}^n (\mathbf{skip}, \sigma'_1)$.

Lemma A.2 For all n, σ_1, σ_2 and C, if $(C, \sigma_1, \mathcal{R}) \Longrightarrow^n \mathcal{G}$, $\sigma_1 \models_{\mathrm{SL}} I * \mathsf{true}$, and $\sigma_2 \models_{\mathrm{SL}} I' * \mathsf{true}$, then $(C, \sigma_1 \uplus \sigma_2, \mathcal{R}') \Longrightarrow^n \mathcal{G}'$.

Lemma A.3 (Frame-Sound)
If R; G; $I \models \{p\}\, C\, \{q\}$, then $R * R'$; $G * G'$; $I * I' \models \{p * r\}\, C\, \{q * r\}$.

Proof. By Definition 6.5, we need to prove that, for all σ, if $\sigma \models_{\mathrm{SL}} p * r$, we have

(1) if $(C, \sigma) \overset{\mathcal{R}'}{\longmapsto}{}^* (\mathbf{skip}, \sigma')$, then $\sigma' \models_{\mathrm{SL}} q * r$;

(2) for all n, $(C, \sigma, \mathcal{R}') \Longrightarrow^n \mathcal{G}'$.

By $\sigma \models_{\mathrm{SL}} p * r$, we know there exist σ_1 and σ_2 such that $\sigma = \sigma_1 \uplus \sigma_2$, $\sigma_1 \models_{\mathrm{SL}} p$, and $\sigma_2 \models_{\mathrm{SL}} r$. By R; G; $I \models \{p\}\, C\, \{q\}$, we know:

(a) for all σ''_1, if $(C, \sigma_1) \overset{\mathcal{R}}{\longmapsto}{}^* (\mathbf{skip}, \sigma''_1)$, then $\sigma''_1 \models_{\mathrm{SL}} q$; and

(b) for all n, $(C, \sigma_1, \mathcal{R}) \Longrightarrow^n \mathcal{G}$.

To prove (1), suppose $(C, \sigma) \overset{\mathcal{R}'}{\longmapsto}{}^* (\mathbf{skip}, \sigma')$. From (b) and Lemma A.1 we know there exists σ'_1 and σ'_2 such that $\sigma' = \sigma'_1 \uplus \sigma'_2$, $\sigma'_2 \models_{\mathrm{SL}} r$, and $(C, \sigma_1) \overset{\mathcal{R}'}{\longmapsto}{}^* (\mathbf{skip}, \sigma'_1)$. By (a) we know $\sigma' \models_{\mathrm{SL}} q * r$.

The proof of (2) follows (b) and Lemma A.2. □

A.2 Soundness of the HIDE rule

Suppose the HIDE rule is applied to derive R; G; $I \vdash \{p\}\, C\, \{q\}$. We want to prove R; G; $I \models \{p\}\, C\, \{q\}$. By inversion of the HIDE rule we know $R * R'$; $G * G'$; $I * I' \vdash \{p\}\, C\, \{q\}$ and $I \triangleright \{R, G\}$. By the induction hypothesis (of Theorem 6.6), we know $R * R'$; $G * G'$; $I * I' \models \{p\}\, C\, \{q\}$. Also, by Lemma 6.7 we know $I * I' \triangleright \{R * R', G * G'\}$ and $p \vee q \Rightarrow I * I' * \mathsf{true}$.

Let $\mathcal{R} = [\![R * \mathsf{Id}]\!]$, $\mathcal{R}' = [\![R * R' * \mathsf{Id}]\!]$, $\mathcal{G} = [\![G * \mathsf{True}]\!]$, and $\mathcal{G}' = [\![G * G' * \mathsf{True}]\!]$. We can prove the following Lemmas A.4, A.5 and A.6. Soundness of the HIDE rule is shown in A.7.

Lemma A.4 If $I \triangleright R$, $(I * I') \triangleright (R * R')$, $\sigma \models_{\mathrm{SL}} I * I' * \mathsf{true}$, and $(\sigma, \sigma') \in [\![R * \mathsf{Id}]\!]$, then $(\sigma, \sigma') \in [\![R * R' * \mathsf{Id}]\!]$.

Lemma A.5
For all n, C, σ and σ', if $\sigma \models_{\mathrm{SL}} I * I' * \mathsf{true}$, $(C, \sigma, \mathcal{R}') \Longrightarrow^n \mathcal{G}'$, and $(C, \sigma) \overset{\mathcal{R}}{\longmapsto}{}^n (\mathbf{skip}, \sigma')$, then $(C, \sigma) \overset{\mathcal{R}'}{\longmapsto}{}^n (\mathbf{skip}, \sigma')$.

Lemma A.6
For all n, C and σ, if $\sigma \models_{\mathrm{SL}} I * I' * \mathsf{true}$ and $(C, \sigma, \mathcal{R}') \Longrightarrow^n \mathcal{G}'$, then $(C, \sigma, \mathcal{R}) \Longrightarrow^n \mathcal{G}$.

Lemma A.7 (Hide-Sound)
If $R * R'$; $G * G'$; $I * I' \models \{p\}\, C\, \{q\}$, then R; G; $I \models \{p\}\, C\, \{q\}$.

The proof of Lemma A.7 is similar to the proof of Lemma A.3.

A.3 Soundness of the PAR rule

Suppose the PAR rule is applied to derive

$$R;\ G_1 \vee G_2;\ I \vdash \{p_1 * p_2 * r\}\, C_1 \parallel C_2\, \{q_1 * q_2 * (r_1 \wedge r_2)\}.$$

We want to prove

$$R;\ G_1 \vee G_2;\ I \models \{p_1 * p_2 * r\}\, C_1 \parallel C_2\, \{q_1 * q_2 * (r_1 \wedge r_2)\}.$$

By inversion of the PAR rule we know $R \vee G_2$; G_1; $I \vdash \{p_1 * r\}\, C_1\, \{q_1 * r_1\}$, $R \vee G_1$; G_2; $I \vdash \{p_2 * r\}\, C_2\, \{q_2 * r_2\}$, $I \triangleright R$, and $r \vee r_1 \vee r_2 \Rightarrow I$ hold. By the induction hypothesis (of Theorem 6.6), we know

$$R \vee G_2;\ G_1;\ I \models \{p_1 * r\}\, C_1\, \{q_1 * r_1\}$$

and

$$R \vee G_1;\ G_2;\ I \models \{p_2 * r\}\, C_2\, \{q_2 * r_2\}.$$

Also, by Lemma 6.7 we know $I \triangleright \{R \vee G_2, G_1, R \vee G_1, G_2\}$.

Let $\mathcal{R}_1 = [\![(R \vee G_2) * \mathsf{Id}]\!]$, $\mathcal{R}_2 = [\![(R \vee G_1) * \mathsf{Id}]\!]$, $\mathcal{R} = [\![R * \mathsf{Id}]\!]$, $\mathcal{G}_1 = [\![G_1 * \mathsf{True}]\!]$, $\mathcal{G}_2 = [\![G_2 * \mathsf{True}]\!]$, and $\mathcal{G} = [\![(G_1 \vee G_2) * \mathsf{True}]\!]$. We can prove the following Lemmas A.8 and A.9. Soundness of the PAR rule is shown in A.10.

Lemma A.8 For all n, C_1, C_2, σ_1, σ_2, σ_r and σ', if $\sigma_r \models_{\mathrm{SL}} I$, $(C_1, \sigma_1 \uplus \sigma_r, \mathcal{R}_1) \Longrightarrow^n \mathcal{G}_1$, $(C_2, \sigma_2 \uplus \sigma_r, \mathcal{R}_2) \Longrightarrow^n \mathcal{G}_2$, and $(C_1 \parallel C_2, \sigma_1 \uplus \sigma_2 \uplus \sigma_r) \overset{\mathcal{R}}{\longmapsto}{}^{n+1} (\mathbf{skip}, \sigma')$, there exist σ'_1, σ'_2 and σ'_r such that $\sigma' = \sigma'_1 \uplus \sigma'_2 \uplus \sigma'_r$, $\sigma'_r \models_{\mathrm{SL}} I$, $(C_1, \sigma_1 \uplus \sigma_r) \overset{\mathcal{R}_1}{\longmapsto}{}^n (\mathbf{skip}, \sigma'_1 \uplus \sigma'_r)$ and $(C_2, \sigma_2 \uplus \sigma_r) \overset{\mathcal{R}_2}{\longmapsto}{}^n (\mathbf{skip}, \sigma'_2 \uplus \sigma'_r)$.

Lemma A.9 For all n, C_1, C_2, σ_1, σ_2, σ_r and σ, if $\sigma = \sigma_1 \uplus \sigma_2 \uplus \sigma_r$, $\sigma_r \models_{\mathrm{SL}} I$, $(C_1, \sigma_1 \uplus \sigma_r, \mathcal{R}_1) \Longrightarrow^n \mathcal{G}_1$, and $(C_2, \sigma_2 \uplus \sigma_r, \mathcal{R}_2) \Longrightarrow^n \mathcal{G}_2$, then $(C_1 \parallel C_2, \sigma, \mathcal{R}) \Longrightarrow^n \mathcal{G}$.

Lemma A.10 (Par-Sound) If $R \vee G_2$; G_1; $I \models \{p_1 * r\}\, C_1\, \{q_1 * r_1\}$ and $R \vee G_1$; G_2; $I \models \{p_2 * r\}\, C_2\, \{q_2 * r_2\}$, then R; $G_1 \vee G_2$; $I \models \{p_1 * p_2 * r\}\, C_1 \parallel C_2\, \{q_1 * q_2 * (r_1 \wedge r_2)\}$.

Proof. By Definition 6.5, we need to prove that, for all σ, if $\sigma \models_{\mathrm{SL}} p_1 * p_2 * r$, we have

(1) if $(C_1 \parallel C_2, \sigma) \overset{\mathcal{R}}{\longmapsto}{}^* (\mathbf{skip}, \sigma')$, then $\sigma' \models_{\mathrm{SL}} q_1 * q_2 * (r_1 \wedge r_2)$;

(2) for all n, $(C_1 \parallel C_2, \sigma, \mathcal{R}) \Longrightarrow^n \mathcal{G}$.

By $\sigma \models_{\mathrm{SL}} p_1 * p_2 * r$ we know there exit σ_1, σ_2 and σ_r such that $\sigma = \sigma_1 \uplus \sigma_2 \uplus \sigma_r$, $\sigma_r \models_{\mathrm{SL}} I$, $\sigma_1 \models_{\mathrm{SL}} p_1$, and $\sigma_2 \models_{\mathrm{SL}} p_2$. By $R \vee G_2$; G_1; $I \models \{p_1 * r\}\, C_1\, \{q_1 * r_1\}$ we have

(a1) for all σ'', if $(C_1, \sigma_1 \uplus \sigma_r) \overset{\mathcal{R}_1}{\longmapsto}{}^* (\mathbf{skip}, \sigma'')$, then $\sigma'' \models_{\mathrm{SL}} q_1 * r_1$;

(b1) for all n, $(C_1, \sigma_1 \uplus \sigma_r, \mathcal{R}_1) \Longrightarrow^n \mathcal{G}_1$.

Similarly, we have:

(a2) for all σ'', if $(C_2, \sigma_2 \uplus \sigma_r) \overset{\mathcal{R}_2}{\longmapsto}{}^* (\mathbf{skip}, \sigma'')$, then $\sigma'' \models_{\mathrm{SL}} q_2 * r_2$;

(b2) for all n, $(C_2, \sigma_2 \uplus \sigma_r, \mathcal{R}_2) \Longrightarrow^n \mathcal{G}_2$.

By (b1), (b2) and Lemma A.8 we know there exist σ'_1, σ'_2 and σ'_r such that $\sigma' = \sigma'_1 \uplus \sigma'_2 \uplus \sigma'_r$, $\sigma'_r \models_{\mathrm{SL}} I$, $(C_1, \sigma_1 \uplus \sigma_r) \overset{\mathcal{R}_1}{\longmapsto}{}^n (\mathbf{skip}, \sigma'_1 \uplus \sigma'_r)$ and $(C_2, \sigma_2 \uplus \sigma_r) \overset{\mathcal{R}_2}{\longmapsto}{}^n (\mathbf{skip}, \sigma'_2 \uplus \sigma'_r)$. By (a1) and (a2) we know $\sigma'_1 \uplus \sigma'_r \models_{\mathrm{SL}} q_1 * r_1$ and $\sigma'_2 \uplus \sigma'_r \models_{\mathrm{SL}} q_2 * r_2$. Since $r_1 \vee r_2 \Rightarrow I$ and $\mathsf{precise}(I)$, we have $\sigma'_1 \uplus \sigma'_2 \uplus \sigma'_r \models_{\mathrm{SL}} q_1 * q_2 * (r_1 \wedge r_2)$. Thus (1) is proved.

The proof of (2) follows (b1), (b2) and Lemma A.9. □