Technical Report

UCAM-CL-TR-777
ISSN 1476-2986

Number 777

Concurrent Abstract Predicates

Thomas Dinsdale-Young, Mike Dodds,
Philippa Gardner, Matthew Parkinson,
Viktor Vafeiadis

April 2010

ISSN 1476-2986

Concurrent Abstract Predicates

(extended version)

Thomas Dinsdale-Young Mike Dodds Philippa Gardner
Matthew Parkinson Viktor Vafeiadis

Abstract

Abstraction is key to understanding and reasoning about large computer systems. Abstraction is simple to achieve if the relevant data structures are disjoint, but rather difficult when they are partially shared, as is often the case for concurrent modules. We present a program logic for reasoning abstractly about data structures that provides a fiction of disjointness and permits compositional reasoning. The internal details of a module are completely hidden from the client by *concurrent abstract predicates*. We reason about a module's implementation using separation logic with permissions, and provide abstract specifications for use by client programs using concurrent abstract predicates. We illustrate our abstract reasoning by building two implementations of a lock module on top of hardware instructions, and two implementations of a concurrent set module on top of the lock module.

1 Introduction

When designing physical systems, we use abstraction and locality to hide irrelevant details. For example, when building a house in London, we do not consider the gravitational forces on individual brick molecules, nor what the weather is like in Paris. Similarly, we use abstraction and locality when designing and reasoning about large computer systems. Locality allows us to consider small parts of a system in isolation. Abstraction gives us a structured view of the system, because components can be represented by their abstract properties.

With locality, we can directly provide some degree of abstraction. Using separation logic [16], we can prove that a module operates only on a particular data structure, not accessed by other modules. If the structure is manipulated only by module functions, it can be represented just in terms of its abstract properties, using abstract predicates [21]. For example, we can give a specification for a set using an abstract predicate to assert that "the set is $\{5,6\}$". A client can then reason about the set without reasoning about its internal implementation: given "the set is $\{5,6\}$", the client can infer, after deleting 6, that "the set is $\{5\}$".

This combination of abstract predicates with the low-level locality from separation logic supports coarse-grained reasoning about modules, where each data structure is represented by a single abstract predicate. However, we often need to reason about modules in a fine-grained manner, where many abstract predicates refer to properties of the same data structure. For example, a fine-grained specification for the set module would allow element 6 to be manipulated separately from the rest of the set. Fine-grained reasoning of this sort is advocated by context logic [3].

Fine-grained abstractions often cannot be achieved using traditional abstract predicates. This is because separation in the abstraction (union, in the case of the set module) need not correspond to separation in the implementation [7]. If the set module is implemented using an array, and each element of the set is represented with a disjoint element of the array, then the high-level and low-level separation correspond. However, if the set module is implemented as a singly-linked list, then the implementation must traverse the global list to manipulate individual set elements. Individual set elements are not represented disjointly in the implementation, and fine-grained reasoning is not possible with traditional abstract predicates combined with separation logic.

In this paper, we present a program logic that allows fine-grained abstraction in the presence of sharing, by introducing *concurrent abstract predicates.* These predicates present a fiction of disjointness [7]; that is, they can be used *as if* each abstract predicate represents disjoint resource, whereas in fact resources are shared between predicates. For example, given a set implemented by a linked list we can write abstract predicates asserting "the set contains 5, which I control" and "the set contains 6, which I control". Both predicates assert properties about the same shared structure, and both can be used at the same time by separate threads: for example, elements can be deleted concurrently from a set.

Concurrent abstract predicates capture information about the permitted changes to the shared structure. In the case of the set predicates, each predicate gives the thread full control over a particular element of the set. Only the thread owning the predicate can remove this element. We implement this control using resource permissions [8], with the property that the permissions must ensure that a predicate is *self-stable*: that is, immune from interference from the surrounding environment. Predicates are thus able to specify independent properties about the data, even though the data are shared.

With our program logic, a module implementation can be verified against a high-level specification expressed using concurrent abstract predicates. Clients of the module can then be verified purely in terms of this high-level specification, without reference to the module's implementation. We demonstrate this by presenting two implementations of a lock module satisfying the same abstract lock specification, and using this specification to build two implementations of a concurrent set satisfying the same abstract set specification. At each level, we reason entirely abstractly, avoiding reasoning about the implementation of the preceding level. Hence, concurrent abstract predicates provide the necessary abstraction for compositional reasoning about concurrent systems.

1.1 Paper Structure

In §2 we provide an informal development of our approach using the simple example of an abstract lock specification. We show how our approach can be used to validate two implementations of a lock. In §3 we give an implementation of a set, which makes use of a lock conforming to our abstract specification. We validate the set using our abstract lock specification. In §4 we give a syntax for logical assertions and a proof system for judgements about programs. We define soundness for our logic and sketch our soundness argument. Finally, in §5 we draw conclusions and relate our approach to other work.

In this long version we include appendices give full technical details for our approach. Appendix A defines the operational semantics of our programming language. This semantics omits logical annotations, and deals purely with concrete state. Appendix B gives a full soundness argument for the proof system.

2 Informal Development

We develop our core idea, to define abstract specifications for concurrent modules and prove that concrete module implementations satisfy these specifications. We motivate our work using a lock module, one of the simplest examples of concurrent resource sharing. We define an abstract specification for locks, and give two implementations satisfying the specification.

2.1 Lock Specification

A typical lock module has the functions `lock(x)` and `unlock(x)`. It also has a mechanism for constructing locks, such as `makelock(n)`, which allocates a lock and a contiguous block of memory of size `n`. We give these functions the following specification:

$$\begin{array}{ccc}
\{\mathsf{isLock}(\mathtt{x})\} & \mathtt{lock}(\mathtt{x}) & \{\mathsf{isLock}(\mathtt{x}) * \mathsf{Locked}(\mathtt{x})\} \\
\{\mathsf{Locked}(\mathtt{x})\} & \mathtt{unlock}(\mathtt{x}) & \{\mathsf{emp}\} \\
\{\mathsf{emp}\} & \mathtt{makelock}(\mathtt{n}) & \left\{ \begin{array}{c} \exists x.\, \mathrm{ret} = x \wedge \mathsf{isLock}(x) * \mathsf{Locked}(x) \\ * (x+1) \mapsto \_ * \ldots * (x+\mathtt{n}) \mapsto \_ \end{array} \right\}
\end{array}$$

This abstract specification, which is presented by the module to the client, is independent of the underlying implementation.[1] The assertions $\mathsf{isLock}(x)$ and $\mathsf{Locked}(x)$ are abstract predicates. $\mathsf{isLock}(x)$ asserts that the lock at x can be acquired by the thread with this assertion, while $\mathsf{Locked}(x)$ asserts that the thread holds the lock. We use the separating conjunction, $*$, from separation logic: $p * q$ asserts that the state can be split disjointly into two parts, one satisfying p and the other satisfying q. Below we give a concrete interpretation of these predicates for a simple compare-and-swap lock.

The module presents the following abstract predicate axioms to the client:

$$\mathsf{isLock}(x) \iff \mathsf{isLock}(x) * \mathsf{isLock}(x) \tag{1}$$

$$\mathsf{Locked}(x) * \mathsf{Locked}(x) \iff \mathsf{false} \tag{2}$$

The first axiom allows the client to share freely the knowledge that x is a lock.[2] The second axiom captures the fact that a lock can only be locked once. With the separation logic interpretation of triples (given in §4.5), the client can infer that, if `lock(x)` is called twice in succession, then the program will not terminate as the post-condition is not satisfiable.

[1]This specification resembles those used in the work of Gotsman *et al.* [11] and Hobor *et al.* [15] on dynamically-allocated locks.

[2]We do not record the splittings of $\mathsf{isLock}(x)$, although we could use permissions [2] to explicitly track this information.

2.2 Example: A Compare-and-Swap Lock

Consider a simple compare-and-swap lock implementation.

```
lock(x) {                        unlock(x) {      makelock(n) {
 local b;                         ⟨[x] := 0⟩       local x := alloc(n+1);
 do ⟨b := ¬CAS(&x, 0, 1)⟩        }                 [x] := 1;
 while(b)                                          return x;
}                                                 }
```

Here angle brackets denote atomic statements. The instruction $\texttt{CAS}(a, v_1, v_2)$ is an atomic compare-and-swap operation that reads the value stored at address a, compares it with v_1 and replaces it with v_2 if the two are equal. CAS returns a boolean value indicating whether the swap succeeded.

Interpretation of Abstract Predicates. We relate the lock implementation to our lock specification by giving a concrete interpretation of the abstract predicates. The predicates are interpreted not just as assertions about the internal state of the module, but also as assertions about the internal *interference* of the module: that is, how concurrent threads can modify shared parts of the module state.

To describe this internal interface, we extend separation logic with two assertions, the shared region assertion $\boxed{P}^{r}_{I(\vec{x})}$ and the permission assertion $[\mathrm{A}]^r_\pi$. The shared region assertion $\boxed{P}^{r}_{I(\vec{x})}$ specifies that there is a shared region of memory, identified by label r, and that the entire shared region satisfies P. The shared region is indivisible to ensure that all threads maintain a consistent view of it. This is expressed by the logical equivalence $\boxed{P}^{r}_{I(\vec{x})} * \boxed{Q}^{r}_{I(\vec{x})} \Leftrightarrow \boxed{P \wedge Q}^{r}_{I(\vec{x})}$. The possible changes to the shared region that can occur (the so-called *actions* on the shared region) are declared by the environment $I(\vec{x})$.

The permission assertion $[\textsc{Ac}]^r_\pi$ specifies that the thread has permission π to perform action $\textsc{Ac}$ over shared region r, provided the action is declared in the environment. Following Boyland [2], the permission π can be

- the fractional permission, $\pi \in (0, 1)$, denoting that both the thread and the environment can do the action, or
- the full permission, $\pi = 1$, denoting that the thread can do the action but the environment cannot.[3]

We now have the logical machinery to interpret our lock predicates concretely:

$$\begin{aligned}
\mathsf{isLock}(x) &\equiv \exists r.\, \exists \pi.\, [\textsc{Lock}]^r_\pi * \boxed{(x \mapsto 0 * [\textsc{Unlock}]^r_1) \vee x \mapsto 1}^{r}_{I(r,x)} \\
\mathsf{Locked}(x) &\equiv \exists r.\, [\textsc{Unlock}]^r_1 * \boxed{x \mapsto 1}^{r}_{I(r,x)}
\end{aligned}$$

The abstract predicate $\mathsf{isLock}(x)$ is interpreted by the concrete, implementation-specific assertion on the right-hand side. This specifies that the thread's local region contains the permission $[\textsc{Lock}]^r_\pi$, meaning that the thread can acquire the lock. It also asserts that the shared region satisfies the module's invariant: either the lock is unlocked ($x \mapsto 0$) and

[3]The state model also contains a zero permission, 0, denoting that the thread may not do the action but the environment may.

the region holds the full permission $[\textsc{Unlock}]^r_1$ to unlock the lock; or the lock is locked $(x \mapsto 1)$ and the unlocking permission is gone (the thread that acquired the lock has it).

Meanwhile, the abstract predicate $\mathsf{Locked}(x)$ is interpreted as the permission assertion $[\textsc{Unlock}]^r_1$ in the local state, giving the current thread full permission to unlock the lock in region r, and the shared region assertion, stating that the lock is locked $(x \mapsto 1)$.

The actions permitted on the lock's shared region are declared in $I(r, x)$. Actions describe how either the current thread or the environment may modify the shared region. They have the form $\mathrm{A}\colon P \rightsquigarrow Q$, where assertion P describes the part of the shared region required to do the action and Q describes the part of the region after the action. The actions for the lock module are

$$I(r,x) \stackrel{\text{def}}{=} \left(\begin{array}{rccc} \textsc{Lock}\colon & x \mapsto 0 * [\textsc{Unlock}]^r_1 & \rightsquigarrow & x \mapsto 1, \\ \textsc{Unlock}\colon & x \mapsto 1 & \rightsquigarrow & x \mapsto 0 * [\textsc{Unlock}]^r_1 \end{array} \right)$$

The $\textsc{Lock}$ action requires that the shared region contains the unlocked lock $(x \mapsto 0)$ and full permission $[\textsc{Unlock}]^r_1$ to unlock the lock. The result of the action is to lock the lock $(x \mapsto 1)$ and to move the full unlock permission to the thread's local state ($[\textsc{Unlock}]^r_1$ has gone from the shared region). The movement of $[\textsc{Unlock}]^r_1$ into the locking thread's local region allows the thread to release the lock afterwards. Note that the local region is not explicitly represented in the action; since interference only happens on the shared region, actions do not need to be prescriptive about local state.

The $\textsc{Unlock}$ action requires that the shared region r contains the locked lock $(x \mapsto 1)$. The result of the action is to unlock the lock $(x \mapsto 0)$ and move the $[\textsc{Unlock}]^r_1$ permission into the shared region. The thread must have $[\textsc{Unlock}]^r_1$ in its local state in order to move it to the shared state as a result of the action.

Notice that $\textsc{Unlock}$ is self-referential. The action moves exclusive permission on itself out of local state. Consequently, a thread can only apply $\textsc{Unlock}$ once (intuitively, a thread can only release a lock once without locking it again). In §4.2, we discuss how our semantics supports such self-referential actions.

The abstract predicates must be *self-stable* with respect to the actions: that is, for any action permitted by the module (actions in $I(r, x)$), the predicate must remain true. Self-stability ensures that a client can use these predicates without having to consider the module's internal interference. For example, assume that the predicate $\mathsf{Locked}(x)$ is true. There are two actions the environment can perform that can potentially affect the location x:

- $\textsc{Lock}$, but this action does not apply, as x has value 1 in the shared region of $\mathsf{Locked}(x)$; and

- $\textsc{Unlock}$, but this action also does not apply, as full permission on it is in the local region of $\mathsf{Locked}(x)$.

The implementer of the module must show that the concrete interpretation of the predicates satisfies the axioms presented to the client. In our example, axiom 1, that only a single $\mathsf{Locked}(x)$ can exist, follows from the presence in the thread's local region of full permission on $\textsc{Unlock}$. Axiom 2, that $\mathsf{isLock}(x)$ can be split, follows from the fact that non-exclusive permissions can be arbitrarily subdivided and that $*$ behaves additively on shared region assertions.

```
$\{\mathsf{isLock}(\mathtt{x})\}$
lock(x) {
  $\left\{\exists r.\,\pi.\,[\textsc{Lock}]^r_\pi * \boxed{(\mathtt{x} \mapsto 0 * [\textsc{Unlock}]^r_1) \vee \mathtt{x} \mapsto 1}^{r}_{I(r,\mathtt{x})}\right\}$
  local b;
  do
    $\left\{\exists r.\,\pi.\,[\textsc{Lock}]^r_\pi * \boxed{(\mathtt{x} \mapsto 0 * [\textsc{Unlock}]^r_1) \vee \mathtt{x} \mapsto 1}^{r}_{I(r,\mathtt{x})}\right\}$
    ⟨b := ¬CAS(&x,0,1)⟩;
    $\left\{\begin{array}{l}\exists r.\,\pi.\,\left(\boxed{\mathtt{x} \mapsto 1}^{r}_{I(r,\mathtt{x})} * [\textsc{Lock}]^r_\pi * [\textsc{Unlock}]^r_1 * \mathsf{b} = \mathsf{false}\right) \vee \\ \left(\boxed{(\mathtt{x} \mapsto 0 * [\textsc{Unlock}]^r_1) \vee \mathtt{x} \mapsto 1}^{r}_{I(r,\mathtt{x})} * [\textsc{Lock}]^r_\pi * \mathsf{b} = \mathsf{true}\right)\end{array}\right\}$
  while(b)
  $\left\{\exists r.\,\boxed{\mathtt{x} \mapsto 1}^{r}_{I(r,\mathtt{x})} * [\textsc{Lock}]^r_\pi * [\textsc{Unlock}]^r_1 * \mathsf{b} = \mathsf{false}\right\}$
}
$\{\mathsf{isLock}(\mathtt{x}) * \mathsf{Locked}(\mathtt{x})\}$

$\{\mathsf{Locked}(\mathtt{x})\}$
unlock(x) {
  $\left\{\exists r.\,[\textsc{Unlock}]^r_1 * \boxed{\mathtt{x} \mapsto 1}^{r}_{I(r,\mathtt{x})}\right\}$
  ⟨[x] := 0⟩;
  $\left\{\exists r.\,\boxed{\mathtt{x} \mapsto 0 * [\textsc{Unlock}]^r_1}^{r}_{I(r,\mathtt{x})}\right\}$
  // Stabilise the assertion.
  $\left\{\exists r.\,\boxed{(\mathtt{x} \mapsto 0 * [\textsc{Unlock}]^r_1) \vee \mathtt{x} \mapsto 1}^{r}_{I(r,\mathtt{x})}\right\}$
}
$\{\mathsf{emp}\}$
```

Figure 1: Verifying the compare-and-swap lock implementation: `lock` and `unlock`.

Verifying the Lock Implementation. Given the definitions above, the lock implementation can be verified against its specification; see Fig. 1 and Fig. 2.

For the `unlock` case, the atomic update $\langle [x] := 0\rangle$ is allowed, because it can be viewed as performing the Unlock action, full permission for which is in the thread's local region. The third assertion specifies that the permission $[\textsc{Unlock}]^r_1$ has moved from the local region to the shared region r as stipulated by the unlock action. This assertion is not, however, stable under interference from the environment since another thread could acquire the lock. It does imply the fourth assertion, which is stable under such interference. The semantics of assertions allows us to forget about the shared region, resulting in the post-condition, emp.

For the `lock` case, the key proof step is the atomic compare-and-swap command in the loop. If successful, this command updates the location referred to by x in the shared region from 0 to 1. This update is allowed because of the permission $[\textsc{Lock}]^r_\pi$ in the thread's local region and the action in $I(r, x)$. The post-condition of the `CAS` instruction

```
$\{\mathsf{emp}\}$
 makelock(n) {
   local x := alloc(n + 1);
  $\{\mathtt{x} \mapsto \_ * (\mathtt{x}+1) \mapsto \_ * \ldots * (\mathtt{x}+\mathtt{n}) \mapsto \_\}$
   [x] := 1;
  $\{\mathtt{x} \mapsto 1 * (\mathtt{x}+1) \mapsto \_ * \ldots * (\mathtt{x}+\mathtt{n}) \mapsto \_\}$
   // Create shared lock region.
  $\left\{\exists r. \boxed{\mathtt{x} \mapsto 1}^{r}_{I(r,\mathtt{x})} * [\text{LOCK}]^r_1 * [\text{UNLOCK}]^r_1 * (\mathtt{x}+1) \mapsto \_ * \ldots * (\mathtt{x}+\mathtt{n}) \mapsto \_\right\}$
   return x;
 }
$\{\exists x. \mathrm{ret} = x \wedge \mathsf{isLock}(x) * \mathsf{Locked}(x) * (x+1) \mapsto \_ * \ldots * (x+\mathtt{n}) \mapsto \_\}$
```

Figure 2: Verifying the compare-and-swap lock implementation: `makelock`.

specifies that either location x has value 1 and the unlock permission has moved into the thread's local region as stipulated by the LOCK action, or nothing has happened and the pre-condition is still satisfied. This post-condition is stable and so the Hoare triple is valid.

For the `makelock` case, the key proof step is the creation of a fresh shared region and its associated permissions. Our proof system includes *repartitioning operator*, denoted by $\Rrightarrow$, which enables us to repartition the state between regions and to create regions. In particular, we have that:

$$P \Rrightarrow \exists r. \boxed{P}^{r}_{I(\vec{x})} * \mathsf{all}(I(\vec{x}))$$

which creates the fresh shared region r and full permission for all of the actions defined in $I(\vec{x})$ (denoted by $\mathsf{all}(I(\vec{x}))$). In our example, we have

$$x \mapsto 1 \Rrightarrow \exists r. \boxed{x \mapsto 1}^{r}_{I(r,x)} * [\text{LOCK}]^r_1 * [\text{UNLOCK}]^r_1$$

The final post-condition results from the definitions of $\mathsf{isLock}(x)$ and $\mathsf{Locked}(x)$.

2.3 The Proof System

We give an informal description of the proof system, with the formal details given in §4. Judgements in our proof system have the form $\Delta; \Gamma \vdash \{P\}C\{Q\}$, where Δ contains predicate definitions and axioms, and Γ presents abstract specifications of the functions used by C. The local Hoare triple $\{P\}C\{Q\}$ has the fault-avoiding partial-correctness interpretation advocated by separation logic: if the program C is run from a state satisfying P then it will not fault, but will either terminate in a state satisfying Q or not terminate at all.

The proof rule for atomic commands is

$$\frac{\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\} \qquad \Delta \vdash P \Rrightarrow^{\{p\}\{q\}} Q \qquad \Delta \vdash \mathsf{stable}(P, Q)}{\Delta; \Gamma \vdash \{P\}\ \langle\, C\,\rangle\ \{Q\}}\ (\text{ATOMIC})$$

The bodies of atomic commands do not contain other atomic commands, nor do they contain parallel composition. They can thus be reasoned about without concurrency, using sequential separation logic. The first premise, $\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\}$, is therefore a triple in sequential separation logic, where p, q denote separation logic assertions that do not specify predicates, shared regions or interference.

The second premise, $\Delta \vdash P \Rrightarrow^{\{p\}\{q\}} Q$, says that the interference allowed by P enables the state to be repartitioned to Q, given the change to memory specified by $\{p\}\{q\}$. In our example, the compare-and-swap instruction corresponds to the state change $\{x \mapsto 0\}\{x \mapsto 1\}$. We also require that P and Q are *stable*, so that they cannot be falsified by concurrently executing threads. Pre-condition and post-condition stability is a general requirement that our proof rules have, which for presentation purposes we keep implicit in the rest of the paper.

The repartitioning arrow $P \Rrightarrow Q$ used earlier for constructing a new region is a shorthand for $P \Rrightarrow^{\{\mathsf{emp}\}\{\mathsf{emp}\}} Q$, i.e. a repartitioning where no concrete state changes. We use this repartitioning in the rule of consequence to move resources between regions. The operator $\Rrightarrow$ includes conventional implication, so this rule of consequence subsumes the traditional one.

$$\frac{\Delta \vdash P \Rrightarrow P' \qquad \Delta; \Gamma \vdash \{P'\}\ C\ \{Q'\} \qquad \Delta \vdash Q' \Rrightarrow Q}{\Delta; \Gamma \vdash \{P\}\ C\ \{Q\}}\ (\textsc{Conseq})$$

We now introduce a rule that allows us to combine a verified module with a verified client to obtain a complete verified system. The idea is that clients of the module are verified with respect to the specification of the module, without reference to the internal interference and the concrete predicate definitions.

Our proof system for programs includes abstract specifications for functions. In previous work on verifying fine-grained concurrent algorithms [24], interference had to be specified explicitly for each function. Here we can prove concrete specifications for the functions of a module, and then present an abstract specification to a client that hides the interference internal to the module.

This is achieved by capturing two kinds of information in our predicate definitions: information about state, and information about interference. Our predicates can describe the internal interference of a module. Given this, we can define high-level specifications where abstract predicates correspond to invariant assertions about the state of the module (that is, they are 'self-stable'). As these assertions are invariant, we can hide the definitions of the predicates and treat the specifications as abstract.

The following proof rule expresses the combination of a module with a client, hiding the module's internal predicate definitions. (In §4.5, we show that this rule is a consequence of more fundamental rules in our proof system).

$$\frac{\begin{array}{c}\Delta \vdash \{P_1\}C_1\{Q_1\} \qquad \cdots \qquad \Delta \vdash \{P_n\}C_n\{Q_n\} \\ \Delta \vdash \Delta' \qquad \Delta'; \{P_1\}f_1\{Q_1\}, \ldots, \{P_n\}f_n\{Q_n\} \vdash \{P\}C\{Q\}\end{array}}{\vdash \{P\}\ \text{let}\ f_1 = C_1 \ldots f_n = C_n\ \text{in}\ C\ \{Q\}}$$

This rule defines a module consisting of functions $f_1 \ldots f_n$ and uses it to verify a client specification $\{P\}\ C\ \{Q\}$. The rule can be read as follows:

If
- the implementation C_i of f_i satisfies the specification $\{P_i\}C_i\{Q_i\}$ under predicate assumptions Δ, for each i;
- the axioms exposed to the client in Δ' are satisfied by the predicate assumptions Δ; and
- the specifications $\{P_1\}f_1\{Q_1\}, \ldots, \{P_n\}f_n\{Q_n\}$ and just the predicate assumptions Δ' can be used to prove the client $\{P\}C\{Q\}$;

then the composed system satisfies $\{P\}$ let $f_1 = C_1 \ldots f_n = C_n$ in C $\{Q\}$.

Using this rule, we can define an abstract module specification and use this specification to verify a client program. Any implementation satisfying the specification can be used in the same place. We are only required to show that the module implementation satisfies the specification.

2.4 Example: A Ticketed Lock

We now consider another, more complex lock algorithm: a lock that issues tickets to clients contending for the lock. This algorithm is used in current versions of Linux. It provides fairness guarantees for threads contending for the lock. Despite the fact that the ticketed lock is quite different from the compare-and-swap lock, we will show this module also implements the abstract lock specification described in §2.1.

The lock operations are defined as follows:

```
lock(x) {
 ⟨int i := INCR(x.next);⟩
 while(⟨i ≠ x.owner⟩) skip;
}
```

```
unlock(x) {
 ⟨x.owner++;⟩
}
```

```
makelock(n) {
 local x := alloc(n+2);
 (x+1).owner := 0;
 (x+1).next := 1;
 return (x+1);
}
```

The instruction `INCR` is an atomic operation that increments a stored value and returns the original value. Field names are encoded as negative offsets (.next = 0, .owner = −1). Fields are encoded as *negative* offsets so that the block of memory associated with the lock begins one byte after the address of the lock. This is required by our abstract specification.

To acquire the lock, a client atomically increments x.next and reads it to a variable i. The value of i becomes the client's ticket. The client waits for x.owner to equal its ticket value i. Once this is the case, the client is the thread holding the lock. The lock is released by incrementing x.owner.

The algorithm is correct because (1) each ticket is held by at most one client and (2) only the thread holding the lock increments x.owner.

Interpretation of Abstract Predicates. The actions for the ticketed lock are:

$$T(t,x) \overset{\text{def}}{=} \left(\begin{array}{rl} \textsc{Take}\colon & \exists k.\,([\textsc{Next}(k)]^t_1 * x.\text{next} \mapsto k \quad \leadsto \quad x.\text{next} \mapsto (k+1)), \\ \textsc{Next}(k)\colon & x.\text{owner} \mapsto k \quad \leadsto \quad x.\text{owner} \mapsto (k+1) * [\textsc{Next}(k)]^t_1 \end{array} \right)$$

Intuitively, Take corresponds to taking a ticket value from x.next, and $\textsc{Next}(k)$ corresponds to releasing the lock when x.owner $= k$. The shared region contains permissions on $\textsc{Next}(k)$ for all the values of k not currently used by active threads. Note the $\exists k$ is required to connect the old and new values of the x.next field in the $\textsc{Take}$ action.

The concrete interpretation of the predicates is as follows:

$$\mathsf{isLock}(x) \quad \equiv \quad \exists t.\, \exists \pi.\, \boxed{\begin{array}{l} \exists k, k'.\, x.\text{owner} \mapsto k * x.\text{next} \mapsto k' * \\ k \leq k' * \circledast k'' \geq k'.\ [\textsc{Next}(k'')]^t_1 * \mathsf{true} \end{array}}^{\,t}_{\,T(t,x)} * [\textsc{Take}]^t_\pi$$

$$\mathsf{Locked}(x) \quad \equiv \quad \exists t, k.\, \boxed{x.\text{owner} \mapsto k * \mathsf{true}}^{\,t}_{\,T(t,x)} * [\textsc{Next}(k)]^t_1$$

Here $\circledast$ is the lifting of $*$ to sets; it is the multiplicative analogue of $\forall$.

$\mathsf{isLock}(x)$ requires values x.next and x.owner to be in the shared region, and that a permission on $\textsc{Next}(k)$ is in the shared region for each value greater than the current ticket x.next. It also requires a permission on $\textsc{Take}$ to be in the thread's local region. $\mathsf{Locked}(x)$ requires just that there is an exclusive permission on $\textsc{Next}(k)$ in the local region for the current value, k, of x.owner.

Self-stability of $\mathsf{Locked}(x)$ is ensured by the fact that the predicate holds full permission on the action $\textsc{Next}(k)$, and the action $\textsc{Take}$ cannot affect the x.owner field. Self-stability for $\mathsf{isLock}(x)$ is ensured by the fact that its definition is invariant under the action $\textsc{Take}$.

The axioms follow trivially from the predicate definitions, as in the CAS lock.

Verifying the Lock Implementation. Given the definitions above, the ticketed lock implementation can be verified against the lock specification, as shown in Fig. 3. The proofs follow the intuitive structure sketched above for the actions. That is, `lock(x)` pulls a ticket and a permission out of the shared region, and `unlock(x)` returns it to the shared region. (We omit the proof of `makelock`, which is very similar to the compare-and-swap lock example.)

3 Composing Abstract Specifications

In the previous section we showed that our system can be used to present abstract specifications for concurrent modules. In this section we show how these specifications can be used to verify client programs, which may themselves be modules satisfying abstract specifications. We illustrate this by defining a specification and two implementations for a concurrent set. The implementations assume a lock module satisfying the specification presented in §2.1.

3.1 A Set Module Specification

A typical set module has three functions: `contains(h, v)`, `add(h, v)` and `remove(h, v)`. We give these functions the following abstract specifications:

$$\begin{array}{rcl} \{\mathsf{in}(\mathtt{h},\mathtt{v})\} & \mathtt{contains}(\mathtt{h},\mathtt{v}) & \{\mathsf{in}(\mathtt{h},\mathtt{v}) * \text{ret} = \mathsf{true}\} \\ \{\mathsf{out}(\mathtt{h},\mathtt{v})\} & \mathtt{contains}(\mathtt{h},\mathtt{v}) & \{\mathsf{out}(\mathtt{h},\mathtt{v}) * \text{ret} = \mathsf{false}\} \\ \{\mathsf{own}(\mathtt{h},\mathtt{v})\} & \mathtt{add}(\mathtt{h},\mathtt{v}) & \{\mathsf{in}(\mathtt{h},\mathtt{v})\} \\ \{\mathsf{own}(\mathtt{h},\mathtt{v})\} & \mathtt{remove}(\mathtt{h},\mathtt{v}) & \{\mathsf{out}(\mathtt{h},\mathtt{v})\} \end{array}$$

$\{\mathsf{isLock}(\mathtt{x})\}$

```
lock(x) {
```

$$\left\{ \exists t, \pi.\, [\textsc{Take}]^t_\pi * \boxed{\begin{array}{r}\exists k, k'.\, k \leq k' * \mathtt{x.owner} \mapsto k * \mathtt{x.next} \mapsto k' \\ * \circledast k'' \geq k'.\ [\textsc{Next}(k'')]^t_1 * \mathsf{true}\end{array}}^{\,t}_{T(t,\mathtt{x})} \right\}$$

```
  ⟨int i := INCR(x.next);⟩
```

$$\left\{ \exists t, \pi.\, [\textsc{Take}]^t_\pi * [\textsc{Next}(\mathtt{i})]^t_1 * \boxed{\begin{array}{r}\exists k, k'.\, k \leq \mathtt{i} < k' * \mathtt{x.owner} \mapsto k * \mathtt{x.next} \mapsto k' \\ * \circledast k'' \geq k'.\ [\textsc{Next}(k'')]^t_1 * \mathsf{true}\end{array}}^{\,t}_{T(t,\mathtt{x})} \right\}$$

```
  while(⟨i ≠ x.owner⟩) skip;
```

$$\left\{ \exists t, \pi.\, [\textsc{Take}]^t_\pi * [\textsc{Next}(\mathtt{i})]^t_1 * \boxed{\begin{array}{r}\exists k'.\, \mathtt{i} < k' * \mathtt{x.owner} \mapsto \mathtt{i} * \mathtt{x.next} \mapsto k' \\ * \circledast k'' \geq k'.\ [\textsc{Next}(k'')]^t_1 * \mathsf{true}\end{array}}^{\,t}_{T(t,\mathtt{x})} \right\}$$

```
}
```

$\{\mathsf{isLock}(\mathtt{x}) * \mathsf{Locked}(\mathtt{x})\}$

$\{\mathsf{Locked}(\mathtt{x})\}$

```
unlock(x) {
```

$$\left\{ \exists t, k.\, \boxed{\mathtt{x.owner} \mapsto k * \mathsf{true}}^{\,t}_{T(t,\mathtt{x})} * [\textsc{Next}(k)]^t_1 \right\}$$

```
  ⟨x.owner++;⟩
```

$$\left\{ \exists t.\, \boxed{\exists k.\, \mathtt{x.owner} \mapsto (k+1) * [\textsc{Next}(k)]^t_1 * \mathsf{true}}^{\,t}_{T(t,\mathtt{x})} \right\}$$

```
}
```

$\{\mathsf{emp}\}$

Figure 3: Proofs for the ticketed lock module operations: `lock` and `unlock`.

Here $\mathsf{in}(h, v)$ is an abstract predicate asserting that the set at h contains v. Correspondingly $\mathsf{out}(h, v)$ asserts that the set does not contain v. We define $\mathsf{own}(h, v)$ as a shorthand for the disjunction of these two predicates.

These assertions capture not only knowledge about the set, but also exclusive permission to alter the set by changing whether v belongs to it. Consequently, $\mathsf{out}(h, v)$ is *not* simply the negation of $\mathsf{in}(h, v)$. The exclusivity of permissions is captured by the module's axiom:

$$\mathsf{own}(h, v) \ * \ \mathsf{own}(h, v) \implies \mathsf{false}$$

We can reason disjointly about set predicates, even though they may be implemented by a single shared structure.

$$\mathtt{remove}(\mathtt{h}, \mathtt{v}_1) \parallel \mathtt{remove}(\mathtt{h}, \mathtt{v}_2)$$

For example, the above command should succeed if it has the permissions to change the values $\mathtt{v}_1$ and $\mathtt{v}_2$ (where $\mathtt{v}_1 \neq \mathtt{v}_2$), and it should yield a set without $\mathtt{v}_1$ and $\mathtt{v}_2$. This intuition is captured by the proof outline shown in Fig. 4.

$$\begin{array}{c}
\{\mathsf{own}(\mathtt{h}, \mathtt{v}_1) * \mathsf{own}(\mathtt{h}, \mathtt{v}_2)\} \\
\begin{array}{c|c}
\{\mathsf{own}(\mathtt{h}, \mathtt{v}_1)\} & \{\mathsf{own}(\mathtt{h}, \mathtt{v}_2)\} \\
\mathtt{remove}(\mathtt{h}, \mathtt{v}_1) & \mathtt{remove}(\mathtt{h}, \mathtt{v}_2) \\
\{\mathsf{out}(\mathtt{h}, \mathtt{v}_1)\} & \{\mathsf{out}(\mathtt{h}, \mathtt{v}_2)\}
\end{array} \\
\{\mathsf{out}(\mathtt{h}, \mathtt{v}_1) * \mathsf{out}(\mathtt{h}, \mathtt{v}_2)\}
\end{array}$$

Figure 4: Proof outline for the set module client.

3.2 Example: The Coarse-grained Set

Consider the following coarse-grained concurrent set module, based on the lock module described in §2.1 and a sequential set module.

```
contains(h,v) {               add(h,v) {             remove(h,v) {
  lock(h.lock);                 lock(h.lock);          lock(h.lock);
  r := scontains(h.set,v);      sadd(h.set,v);         sremove(h.set,v);
  unlock(h.lock);               unlock(h.lock);        unlock(h.lock);
  return r;                   }                      }
}
```

Here `scontains`, `sadd` and `sremove` are the operations of the sequential set module. By *sequential*, we mean that at most one of these operations can safely run at once. The concurrent set achieves concurrency by wrapping the sequential operations in a single big lock, which enforces sequential access to the set. As the implementation uses a single lock, concurrency is very limited. Below in §3.3 we discuss a set module where several threads at once can access the shared set structure.

Interpretation of Abstract Predicates. We assume a coarse-grained sequential set predicate $\mathsf{Set}(y, xs)$ that asserts that the set at location y contains values xs. The predicate Set cannot be split, and so must be held by one thread at once. This enforces sequential behaviour. We assume the following specifications for the sequential set operations:

$$\begin{array}{ccc} \{\mathsf{Set}(\mathtt{h}, vs)\} & \mathtt{scontains}(\mathtt{h},\mathtt{v}) & \{\mathsf{Set}(\mathtt{h}, vs) * \mathrm{ret} = (\mathtt{v} \in vs)\} \\ \{\mathsf{Set}(\mathtt{h}, vs)\} & \mathtt{sadd}(\mathtt{h},\mathtt{v}) & \{\mathsf{Set}(\mathtt{h}, \{\mathtt{v}\} \cup vs)\} \\ \{\mathsf{Set}(\mathtt{h}, vs)\} & \mathtt{sremove}(\mathtt{h},\mathtt{v}) & \{\mathsf{Set}(\mathtt{h}, vs \setminus \{\mathtt{v}\})\} \end{array}$$

In the set implementation, the predicate Set is held in the shared state when the lock is not locked. Then when the lock is acquired by a thread, the predicate is pulled into the thread's local state so that it can be modified according to the sequential set specification. When the lock is released, the predicate is returned to the shared state. The actions for the set module are

$$C(s,h) \overset{\text{def}}{=} \left(\begin{array}{l} \textsc{SChange}(v)\colon \begin{pmatrix} \exists vs, ws.\ \mathsf{Set}(h.\mathrm{set}, vs) \\ * [\textsc{SGap}(ws)]^s_1 \wedge \\ vs \setminus \{v\} = ws \setminus \{v\} \end{pmatrix} \rightsquigarrow \mathsf{Locked}(h.\mathrm{lock}), \\ \quad \textsc{SGap}(ws)\colon \mathsf{Locked}(h.\mathrm{lock}) \rightsquigarrow \mathsf{Set}(h.\mathrm{set}, ws) * [\textsc{SGap}(ws)]^s_1 \end{array} \right)$$

The $\textsc{SGap}(ws)$ action allows the thread to return the set containing ws to the shared state. The $\textsc{SChange}(v)$ action allows a thread to acquire the set from the shared state. To do so, the thread must currently hold the lock. It gives up the permission to release the lock in exchange for the set. The thread also acquires the permission $[\textsc{SGap}(ws)]^s_1$, which allows it to re-acquire the lock permission by relinquishing the set, having only changed whether or not v is in the set.

We first define the auxiliary predicates $\mathsf{allgaps}(s)$, $P_{\in}(h,v,s)$ and $P_{\notin}(h,v,s)$:

$$\begin{array}{rcl}
\mathsf{allgaps}(s) & \equiv & \circledast ws.\,[\textsc{SGap}(ws)]_1^s \\
P_{\triangleleft}(h,v,s) & \equiv & \exists vs.\; v \triangleleft vs \wedge \begin{pmatrix} (\mathsf{allgaps}(s) * \mathsf{Set}(h.\mathrm{set}, vs)) \\ \vee\, \mathsf{Locked}(h.\mathrm{lock}) * ([\textsc{SGap}(vs)]_1^s \mathrel{-\!\!\circledast} \mathsf{allgaps}(s)) \end{pmatrix} \\
 & & \text{where } \triangleleft = \in \text{ or } \triangleleft = \notin
\end{array}$$

$\mathsf{allgaps}$ defines the set of all possible SGap permissions. $P_{\in}(h,v,s)$ is used to assert that the shared state contains either the set with contents vs, where $v \in vs$, and all possible SGap permissions; or it contains the Locked predicate and is missing one of the SGap permissions. The missing SGap permission records the contents of the set when it is released. $P_{\notin}(h,v,s)$ defines the case where $v \notin vs$.

The concrete definitions of $\mathsf{in}(h,v)$ and $\mathsf{out}(h,v)$ are as follows:

$$\begin{array}{rcl}
\mathsf{in}(h,v) & \equiv & \exists s.\, \mathsf{isLock}(h.\mathrm{lock}) * [\textsc{SChange}(v)]_1^s * \boxed{P_{\in}(h,v,s)}_{C(s,h)}^{s} \\
\mathsf{out}(h,v) & \equiv & \exists s.\, \mathsf{isLock}(h.\mathrm{lock}) * [\textsc{SChange}(v)]_1^s * \boxed{P_{\notin}(h,v,s)}_{C(s,h)}^{s}
\end{array}$$

The $\mathsf{in}(h,v)$ predicate gives a thread the permissions to acquire the lock, $\mathsf{isLock}(h.\mathrm{lock})$, and to change whether v is in the set, $[\textsc{SChange}(v)]_1^s$. The shared state is described by the predicate $P_{\in}(h,v,s)$. The $\mathsf{out}(h,v)$ predicate is defined analogously to $\mathsf{in}(h,v)$, but with $\notin$ in place of $\in$.

$\mathsf{in}(h,v)$ and $\mathsf{out}(h,v)$ are self-stable. For $\mathsf{in}(h,v)$, the only actions available to another thread are $\textsc{SChange}(w)$, where $w \neq v$, and $\textsc{SGap}(vs)$, where $v \in vs$. The assertion $P_{\in}(h,v,s)$ is invariant under both of these changes: $\textsc{SChange}(w)$ requires the disjunct $\mathsf{allgaps} * \mathsf{Set}(h.\mathrm{set}, vs)$ to hold and leaves the disjunct $\mathsf{Locked}(h.\mathrm{lock}) * ([\textsc{SGap}(vs)]_1 \mathrel{-\!\!\circledast} \mathsf{allgaps}(s))$ holding; $\textsc{SGap}(vs)$ does the reverse. Similar arguments hold for $\mathsf{out}(h,v)$.

Finally, we must ensure that the module's axiom holds. Each $\mathsf{in}(h,v)$ and $\mathsf{out}(h,v)$ predicate includes the exclusive permission $[\textsc{SChange}(v)]_1^s$. The axiom therefore holds as a consequence of the fact that exclusive permissions cannot be combined.

Verifying the Set Implementation. Given the definitions above, we can verify the implementations of the set module. Fig. 5 shows a proof of `add(h, v)` when the value is not in the set. The case where the value is in the set, and the proofs of `remove` and `contains` follow a similar structure.

The most interesting steps of this proof are those before and after the operation `sadd(h.set, v)`, when the permissions $[\textsc{SChange}(v)]_1^s$ and $[\textsc{SGap}(vs)]_1^s$ are used to repartition between shared and local state. These steps are purely logical repartitioning of the state, moving state between shared and local regions without mutating the underlying heap. Any number of such abstract rewritings can take place in a proof, assuming they are permitted by the permissions held by the thread.

3.3 Example: The Fine-grained Set

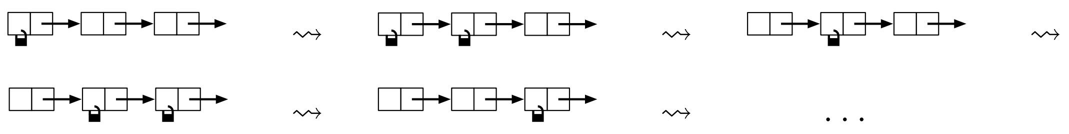
Our previous implementation of a concurrent set used a single global lock, and consequently only a single thread at once could access the set. We now consider a set implementation that uses a sorted list with one lock per node in the list. Many threads at

```
$\{\mathsf{out}(\mathtt{h},\mathtt{v})\}$
add(h,v)
  $\left\{\exists s.\,\mathsf{isLock}(\mathtt{h.lock}) * [\textsc{SChange}(\mathtt{v})]^s_1 * \boxed{P_{\notin}(\mathtt{h},\mathtt{v},s)}^{s}_{C(s,\mathtt{h})}\right\}$
  lock(h.lock);
  $\left\{\exists s.\,\mathsf{isLock}(\mathtt{h.lock}) * \mathsf{Locked}(\mathtt{h.lock}) * [\textsc{SChange}(\mathtt{v})]^s_1 * \boxed{P_{\notin}(\mathtt{h},\mathtt{v},s)}^{s}_{C(s,\mathtt{h})}\right\}$
  // use SChange to extract Set predicate and SGap permission.
  $\left\{\begin{array}{l}\exists s, vs.\,\mathsf{isLock}(\mathtt{h.lock}) * [\textsc{SGap}(vs \cup \{\mathtt{v}\})]^s_1 * [\textsc{SChange}(\mathtt{v})]^s_1 * \mathsf{Set}(\mathtt{h.set}, vs) \\ \qquad * \boxed{\mathsf{Locked}(\mathtt{h.lock}) * ([\textsc{SGap}(vs \cup \{\mathtt{v}\})]^s_1 \mathrel{-\!\circledast} \mathsf{allgaps}(s))}^{s}_{C(s,\mathtt{h})}\end{array}\right\}$
  sadd(h.set,v);
  $\left\{\begin{array}{l}\exists s, vs.\,\mathsf{isLock}(\mathtt{h.lock}) * [\textsc{SGap}(vs \cup \{\mathtt{v}\})]^s_1 * [\textsc{SChange}(\mathtt{v})]^s_1 * \mathsf{Set}(\mathtt{h.set}, vs \cup \{\mathtt{v}\}) \\ \qquad * \boxed{\mathsf{Locked}(\mathtt{h.lock}) * ([\textsc{SGap}(vs \cup \{\mathtt{v}\})]^s_1 \mathrel{-\!\circledast} \mathsf{allgaps}(s))}^{s}_{C(s,\mathtt{h})}\end{array}\right\}$
  // use SGap permission to put back Set and SGap permission.
  $\left\{\exists s.\,\mathsf{isLock}(\mathtt{h.lock}) * \mathsf{Locked}(\mathtt{h.lock}) * [\textsc{SChange}(\mathtt{v})]^s_1 * \boxed{P_{\in}(\mathtt{h},\mathtt{v},s)}^{s}_{C(s,\mathtt{h})}\right\}$
  unlock(h.lock);
  $\left\{\exists s.\,\mathsf{isLock}(\mathtt{h.lock}) * [\textsc{SChange}(\mathtt{v})]^s_1 * \boxed{P_{\in}(\mathtt{h},\mathtt{v},s)}^{s}_{C(s,\mathtt{h})}\right\}$
}
$\{\mathsf{in}(\mathtt{h},\mathtt{v})\}$
```

Figure 5: Proof of the `add(h, v)` specification for the coarse-grained set module.

once can access the list; the algorithm ensures that the threads do not violate each others safety properties. The source code for this algorithm (adapted from [13, §9.5]) is given in Fig. 6.

The three module functions use the function `locate(h, v)` that traverses the sorted list from the head `h` up to the position for a node holding value `v`, whether or not such a node is present. It begins by locking the initial node (the head node) of the list. It then moves down the list by hand-over-hand locking. The algorithm first locks the node following its currently held node, and then releases the previously-held lock. The following diagram illustrates this pattern of locking:



No thread can access a node locked by another thread, or traverse past a locked node. Consequently, a thread cannot overtake any other threads accessing the list. Nodes can be added and removed from locked segments of the list. If a thread locks a node, then a new node can be inserted directly after it, as long as it preserves the sorted nature of the list. Also, if a thread has locked two nodes in sequence, then the second can be removed.

Since nodes of the list are deleted when an element is removed from the set, the lock module must provide a mechanism for disposing of locked blocks, `disposelock(x,n)`. For brevity, we omitted details of this in our exposition. In order for disposal to be sound, no other thread must be able to access the lock, and so we add an extra argument to account for the splitting of locks. This gives a lock predicate $\mathsf{isLock}(x, i)$, where $i \in (0, 1]$. Lock

```
add(h, v) {
  local p, c, z;
  (p, c) := locate(h, v);
  if (c.val ≠ v) {
    z := makelock(2);
    unlock(z);
    z.value := v;
    z.next := c;
    p.next := z;
  }
  unlock(p);
}
```

```
remove(h,v) {
  local p, c, z;
  (p, c) := locate(h, v);
  if (c.val == v) {
    lock(c);
    z := c.next;
    p.next := z;
    disposelock(c, 2);
  }
  unlock(p);
}
```

```
contains(h, v) {
  local p, c, b;
  (p, c) := locate(h, v);
  b := (c.val == v);
  unlock(p);
  return b;
}
```

```
locate(h, v) {
  local p, c;
  p := h;
  lock(p);
  c := p.next;
  while (c.val < v) {
    lock(c);
    unlock(p);
    p := c;
    c := p.next;
  }
  return(p, c);
}
```

Figure 6: Lock-coupling list algorithm.

splitting is achieved by the following axiom:

$$\mathsf{isLock}(x, i+j) \quad \iff \quad \mathsf{isLock}(x,i) * \mathsf{isLock}(x,j)$$

Creating a lock gives $i = 1$, locking only requires $i \in (0,1]$ and disposal requires $i = 1$. Lock disposal has the following specification:

$$\left\{ \begin{array}{l} \mathsf{isLock}(\mathtt{x}, 1) * \mathsf{Locked}(\mathtt{x}) * \\ (\mathtt{x}+1) \mapsto \_ * \ldots * (\mathtt{x}+\mathtt{n}) \mapsto \_ \end{array} \right\} \quad \mathtt{disposelock(x,n)} \quad \{\mathsf{emp}\}$$

For both lock implementations we have given, the `disposelock` function simply deallocates the appropriate memory block.

Interpretation of Abstract Predicates. We first define some basic predicates for describing the structure of a list. List nodes consist of a lock, a value field and a link field. Because owning a node gives the permission to lock its successor, our node predicate includes the isLock predicate of the *next* node.

$$\mathsf{link}(a,b) \equiv (a.\mathrm{next} \mapsto b) * \mathsf{isLock}(b,1) \qquad \mathsf{val}(a,v) \equiv (a.\mathrm{value} \mapsto v)$$
$$\mathsf{node}(a,b,v) \equiv \mathsf{val}(a,v) * \mathsf{link}(a,b)$$

The actions for the fine-grained set module are defined by the interference environment $F(s)$, defined by the following assertion:

$$\textsc{LChange}(v):\quad \exists n,t.\ [\textsc{LGap}(n,t,v)]^s_1 * \mathsf{link}(n,t) \quad\rightsquigarrow\quad \mathsf{Locked}(n)$$

$$\textsc{LGap}(n,t,v):\begin{cases} \mathsf{Locked}(n) \ \rightsquigarrow\ \mathsf{link}(n,t) * [\textsc{LGap}(n,t,v)]^s_1 \\[2ex] \exists v_1,v_2.\begin{pmatrix}\mathsf{Locked}(n)\\ * \mathsf{val}(n,v_1)\\ * \mathsf{val}(t,v_2)\end{pmatrix} \rightsquigarrow \begin{pmatrix}\exists y.\ [\textsc{LGap}(n,t,v)]^s_1 * \mathsf{node}(y,t,v)\\ * \mathsf{link}(n,y) * \mathsf{val}(n,v_1)\\ * \mathsf{val}(t,v_2) * \ \wedge v_1<v<v_2\end{pmatrix} \\[2ex] \begin{pmatrix}\mathsf{Locked}(n) *\\ \mathsf{Locked}(t) * \mathsf{val}(t,v)\end{pmatrix} \rightsquigarrow \begin{pmatrix}[\textsc{LGap}(n,t,v)]^s_1 *\\ [\textsc{LGap}(t,y,v)]^s_1 * \mathsf{link}(n,y)\end{pmatrix}\end{cases}$$

The LGap and LChange actions are analogous to the similarly-named actions defined for the coarse-grained set, in that they allow a thread to remove part of the shared state by holding a lock and to return that part with a limited modification. Rather than removing the entire set as with the coarse-grained set, only a single link of the list is removed with each lock. Both LGap and LChange are parameterised by v, the value that the thread may add or remove from the set.

The LChange(v) action allows a thread, having locked some node n, to take the link of that node (that is, the next pointer and the knowledge of the following node's lock) from the shared state, together with a permission that will allow the thread to return that link, possibly having added or removed a node with value v; in exchange, the thread gives up the $\mathsf{Locked}(n)$ predicate.

The three LGap(n, t, v) actions deal with replacing the link of node n:

- The first simply allows the same link to be returned as was originally removed. In returning the link, the thread also gives up the LGap permission, but regains the $\mathsf{Locked}(n)$ predicate.
- The second allows the link from n to t to be replaced with a link from n to a new node at some address y, having value v, that in turn links to the node at t. The action requires v to fall between the values of the nodes at n and t, which is essential to maintaining the sorted order of the list. As before, the thread gives up the LGap permission and regains the $\mathsf{Locked}(n)$ predicate.
- The third allows the link from n to t to be replaced with one from n to y, where there was formerly a link from t to y. To do this, the thread must also have locked the node at t and obtained its corresponding LGap permission and $\mathsf{link}(t, y)$ from the shared state. Node t must also have the value v; the value component of t is removed from the shared state, and its lock and next components are not returned. The LGap permissions for both nodes are returned to the shared state, and both $\mathsf{Locked}(n)$ and $\mathsf{Locked}(t)$ are regained by the thread.

The predicates $L_{\in}(h, v, s)$ and $L_{\notin}(h, v, s)$ correspond to lists containing and not containing v. Their definitions are given in Fig. 7, along with auxiliary predicates. The slsg ('sorted

$$
\begin{aligned}
\mathsf{lsg}(x,y,S,[]) &\equiv x = y \land S = \emptyset \\
\mathsf{lsg}(x,y,S \uplus \{(x,z)\}, v :: vs) & \\
&\equiv \mathsf{Locked}(x) * \mathsf{val}(x,v) * \mathsf{lsg}(z,y,S,vs) \\
\mathsf{lsg}(x,y,S,v :: vs) &\equiv x \neq y \land \mathsf{node}(x,v,z) * \mathsf{lsg}(z,y,S,vs) \\
\mathsf{slsg}(x,y,S,vs) &\equiv \mathsf{lsg}(x,y,S,vs) \land \mathsf{sorted}(vs) \land \exists vs'.\, vs = -\infty :: vs'@[\infty] \\
\mathsf{gaps}(S,s) &\equiv \circledast(x,y) \notin S.\, \circledast v.\, [\text{LG{\scriptsize AP}}(x,y,v)]^s_1 * \\
&\quad\ \circledast(x,y). \in S.\, \exists w.\, \circledast v \neq w.\, [\text{LG{\scriptsize AP}}(x,y,v)]^s_1 \land \\
&\quad\ \forall x,y,w,z.\, (x,y) \in S \land (w,z) \in S \Rightarrow (x = w \Leftrightarrow y = z) \\
\mathsf{mygaps}(v,s) &\equiv \circledast x,y.\, [\text{LG{\scriptsize AP}}(x,y,v)]^s_1 * \mathsf{true} \\
L_{\triangleleft}(h,v,s) &\equiv \exists vs, S.\ v \triangleleft vs \land \mathsf{slsg}(h,\mathsf{nil},S,vs) * (\mathsf{gaps}(S,s) \land \mathsf{mygaps}(v,s)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \triangleleft = \in \text{ or } \triangleleft = \notin
\end{aligned}
$$

Figure 7: Predicates for lock-coupling list.

list with gaps') predicate represents a list with locked segments. The gaps predicate tracks the unused Gap permissions in the shared state. In both cases, a carrier set S records the locked sections in the list.

The concrete interpretation of the predicates for this implementation of the set module are defined as follows:

$$
\begin{aligned}
\mathsf{in}(h,v) &\equiv \exists s.\, \exists \pi.\, \mathsf{isLock}(h,\pi) * [\text{LC{\scriptsize HANGE}}(v)]^s_1 * \boxed{L_{\in}(h,v,s)}^s_{F(s)} \\
\mathsf{out}(h,v) &\equiv \exists s.\, \exists \pi.\, \mathsf{isLock}(h,\pi) * [\text{LC{\scriptsize HANGE}}(v)]^s_1 * \boxed{L_{\notin}(h,v,s)}^s_{F(s)}
\end{aligned}
$$

The predicate axiom holds as a consequence of the fact that exclusive permissions cannot be combined.

Verifying the set implementation Fig. 8 gives a proof for `locate(h,v)`. The initial code begins the traversal at the head of the list, locking the head node.

The main loop searches through the list checking if the current element, `c`, is less than the element `v` being searched for. At the start of the loop, the thread has locked `p`, signified by having the LGap($\mathtt{p}, \mathtt{c}, \mathtt{v}$) permission in its local state. The first step of the loop locks the new current element `c`. This is allowed as the $\mathsf{link}(\mathtt{p}, \mathtt{c})$ predicate includes $\mathsf{isLock}(\mathtt{c}, 1)$.

As the thread holds the permission to LChange(`v`), the corresponding action is used to swap the newly-acquired $\mathsf{Locked}(\mathtt{c})$ predicate for an LGap permission and the link from c to its successor. The thread has now locked two nodes, so the first should be unlocked. The first LGap($\mathtt{p}, \mathtt{c}, \mathtt{v}$) action is used to regain the $\mathsf{Locked}(\mathtt{p})$ predicate so that `p` can be safely unlocked.

As with the coarse-grained set, locking and unlocking are two-step processes. First the lock is acquired in local state, then a permission is used to change the shared state to reflect this locked status. Similarly, for the unlock, first the permission is used to extract the knowledge that the node is locked, and then it is locally unlocked.

```
locate(h, v) {
  $\{\mathsf{in}(\mathtt{h},\mathtt{v})\}$
  p := h;
  lock(p);
  c := p.next;
  while (c.val < v) {
    $\left\{\exists s.\ \boxed{\begin{array}{r}(\exists v'.\,\mathsf{val}(\mathtt{c},v') * v' < \mathtt{v}) \land \exists vs, S.\ \mathtt{v} \in vs \land \{(\mathtt{p},\mathtt{c})\} \subseteq S \land \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs)\\ * (\mathsf{gaps}(S,s) \land ([\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)))\end{array}}^{s}_{F(s)} \\ * \mathsf{link}(\mathtt{p},\mathtt{c}) * [\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * \exists \pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1\right\}$
    lock(c);
    $\left\{\exists s.\ \boxed{\begin{array}{r}(\exists v'.\,\mathsf{val}(\mathtt{c},v') * v < \mathtt{v}) \land \exists vs, S.\ \mathtt{v} \in vs \land \{(\mathtt{p},\mathtt{c})\} \subseteq S \land \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs)\\ * (\mathsf{gaps}(S,s) \land ([\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)))\end{array}}^{s}_{F(s)} \\ * \mathsf{link}(\mathtt{p},\mathtt{c}) * [\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * \exists \pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1 * \mathsf{Locked}(\mathtt{c})\right\}$
    // By $\textsc{LChange}$ using $\mathsf{Locked}(\mathtt{c})$
    $\left\{\exists s, z.\ \boxed{\begin{array}{l}(\exists v'.\,\mathsf{val}(\mathtt{c},v') * v' < \mathtt{v})\\ \land \exists vs, S.\ \mathtt{v} \in vs \land \{(\mathtt{p},\mathtt{c}),(\mathtt{c},z)\} \subseteq S \land \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs) *\\ (\mathsf{gaps}(S,s) \land (([\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * [\textsc{LGap}(\mathtt{c},z,\mathtt{v})]^s_1) \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)))\end{array}}^{s}_{F(s)} \\ * \mathsf{link}(\mathtt{p},\mathtt{c}) * [\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * [\textsc{LGap}(\mathtt{c},z,\mathtt{v})]^s_1 * \exists \pi.\,\mathsf{isLock}(\mathtt{h},\pi) \\ * [\textsc{LChange}(\mathtt{v})]^s_1 * \mathsf{link}(\mathtt{c},z)\right\}$
    // By $\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})$
    $\left\{\exists s, z.\ \boxed{\begin{array}{r}(\exists v'.\,\mathsf{val}(\mathtt{c},v') * v' < \mathtt{v}) \land \exists vs, S.\ \mathtt{v} \in vs \land \{(\mathtt{c},z)\} \subseteq S \land \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs)\\ * (\mathsf{gaps}(S,s) \land ([\textsc{LGap}(\mathtt{c},z,\mathtt{v})]^s_1 \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)))\end{array}}^{s}_{F(s)} \\ * \mathsf{Locked}(\mathtt{p}) * [\textsc{LGap}(\mathtt{c},z,\mathtt{v})]^s_1 * \exists \pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1 * \mathsf{link}(\mathtt{c},z)\right\}$
    unlock(p);
    p := c;
    c := p.next;
    $\left\{\exists s.\ \boxed{\begin{array}{r}(\exists v'.\,\mathsf{val}(\mathtt{p},v') * v' < \mathtt{v}) \land \exists vs, S.\ \mathtt{v} \in vs \land \{(\mathtt{p},\mathtt{c})\} \subseteq S \land \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs)\\ * (\mathsf{gaps}(S,s) \land ([\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)))\end{array}}^{s}_{F(s)} \\ * [\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * \exists \pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1 * \mathsf{link}(\mathtt{p},\mathtt{c})\right\}$
  }
  $\left\{\exists s.\ \boxed{\begin{array}{r}(\exists v'.\,\mathsf{val}(\mathtt{p},v') * v' < \mathtt{v}) \land (\exists v''.\,\mathsf{val}(\mathtt{c},v'') * v'' \geq \mathtt{v}) \land \exists vs, S.\ \mathtt{v} \in vs \land \{(\mathtt{p},\mathtt{c})\} \subseteq S\\ \land \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs) * (\mathsf{gaps}(S,s) \land ([\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)))\end{array}}^{s}_{F(s)} \\ * [\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * \exists \pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1 * \mathsf{link}(\mathtt{p},\mathtt{c})\right\}$
  return(p, c);
}
```

Figure 8: Proof for $\mathtt{locate}(\mathtt{h},\mathtt{v})$.

```
$\{\mathsf{in}(\mathtt{h},\mathtt{v})\}$
(p, c) := locate(v);   // Use spec. from Fig. 8
$\left\{\exists s.\ \boxed{\begin{array}{r}(\exists v'.\,\mathsf{val}(\mathtt{p},v') * v' < \mathtt{v}) \wedge (\exists v''.\,\mathsf{val}(\mathtt{c},v'') * v'' \geq \mathtt{v}) \wedge \exists vs, S.\ \mathtt{v} \in vs \wedge \{(\mathtt{p},\mathtt{c})\} \subseteq S \\ \wedge\, \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs) * (\mathsf{gaps}(S,s) \wedge ([\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)))\end{array}}^{s}_{F(s)} \\ * [\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * \exists\pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1 * \mathsf{link}(\mathtt{p},\mathtt{c})\right\}$
if (c.val == v) {
  lock(c);
  $\left\{\exists s.\ \boxed{\begin{array}{r}(\mathsf{val}(\mathtt{c},\mathtt{v}) * \mathsf{true}) \wedge \exists vs, S.\ \ \in vs \wedge \{(\mathtt{p},\mathtt{c})\} \subseteq S \wedge \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs) \\ * (\mathsf{gaps}(S,s) \wedge ([\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)))\end{array}}^{s}_{F(s)} \\ * [\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * \exists\pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1 * \mathsf{link}(\mathtt{p},\mathtt{c}) * \mathsf{Locked}(\mathtt{c})\right\}$
  // By LChange using Locked(c)
  $\left\{\exists s, z.\ \boxed{\begin{array}{r}(\mathsf{val}(\mathtt{c},\mathtt{v}) * \mathsf{true}) \wedge \exists vs, S.\ \mathtt{v} \in vs \wedge \{(\mathtt{p},\mathtt{c}),(\mathtt{c},z)\} \subseteq S \wedge \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs) \\ * \left(\mathsf{gaps}(S,s) \wedge \left(\left(\begin{array}{c}[\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 \\ * [\textsc{LGap}(\mathtt{c},z,\mathtt{v})]^s_1\end{array}\right) \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)\right)\right)\end{array}}^{s}_{F(s)} \\ * [\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * [\textsc{LGap}(\mathtt{c},z,\mathtt{v})]^s_1 * \mathsf{link}(\mathtt{p},\mathtt{c}) * \mathsf{link}(\mathtt{c},z) \\ * \exists\pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1\right\}$
  z := c.next;
  p.next := z;
  $\left\{\exists s, z.\ \boxed{\begin{array}{r}(\mathsf{val}(\mathtt{c},\mathtt{v}) * \mathsf{true}) \wedge \exists vs, S.\ \mathtt{v} \in vs \wedge \{(\mathtt{p},\mathtt{c}),(\mathtt{c},z)\} \subseteq S \wedge \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs) \\ * \left(\mathsf{gaps}(S,s) \wedge \left(\left(\begin{array}{c}[\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 \\ * [\textsc{LGap}(\mathtt{c},z,\mathtt{v})]^s_1\end{array}\right) \mathrel{-\!\!\circledast} \mathsf{mygaps}(\mathtt{v},s)\right)\right)\end{array}}^{s}_{F(s)} \\ * [\textsc{LGap}(\mathtt{p},\mathtt{c},\mathtt{v})]^s_1 * [\textsc{LGap}(\mathtt{c},z,\mathtt{v})]^s_1 * \mathsf{link}(\mathtt{p},z) * \mathsf{isLock}(\mathtt{c},1) * (\mathtt{c.next} \mapsto z) \\ * \exists\pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1\right\}$
  // By LGap(p, c, v)
  $\left\{\exists s, z.\ \boxed{\exists vs, S.\ \mathtt{v} \notin vs \wedge \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs) * (\mathsf{gaps}(S,s) \wedge \mathsf{mygaps}(\mathtt{v},s))}^{s}_{F(s)} \\ * \mathsf{Locked}(\mathtt{p}) * \mathsf{Locked}(\mathtt{c}) * \mathsf{val}(\mathtt{c},\mathtt{v}) * \mathsf{isLock}(\mathtt{c},1) * (\mathtt{c.next} \mapsto z) \\ * \exists\pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1\right\}$
  disposelock(c, 2);
}
$\left\{\exists s, z.\ \boxed{\exists vs, S.\ \mathtt{v} \notin vs \wedge \mathsf{slsg}(\mathtt{h},\mathsf{nil},S,vs) * (\mathsf{gaps}(S,s) \wedge \mathsf{mygaps}(\mathtt{v},s))}^{s}_{F(s)} \\ * \mathsf{Locked}(\mathtt{p}) * \exists\pi.\,\mathsf{isLock}(\mathtt{h},\pi) * [\textsc{LChange}(\mathtt{v})]^s_1\right\}$
unlock(p);
$\{\mathsf{out}(\mathtt{h},\mathtt{v})\}$
```

Figure 9: Proof-sketch for remove(h, v).

A proof for `remove(h,x)` is given in Fig. 9. This proof uses the specification for `locate` to acquire a lock on the node immediately prior to the target value. If the value is in the list, the thread uses the $\mathsf{Locked(c)}$ predicate and LChange permission to get the LGap permission on the current node. It then uses the third LGap action to move the node out of the shared state, where it can be safely disposed. In the (empty) else branch, the first LGap action is used to return the link to the shared state without modifying anything.

4 Semantics and Soundness

We present our assertion language and program logic, and sketch a proof of soundness of our logic. The details of the proof are given in the appendix.

4.1 Assertion Syntax

Recall from §2.3 that our proof judgements have the form $\Delta, \Gamma \vdash \{P\}\, C\, \{Q\}$. Here, P and Q are *assertions* from the set Assn. They are constructed from the standard connectives and quantifiers of separation logic, together with permission assertions, regions of the form $\boxed{P}^{r}_{I}$ where I denotes an *interference assertion* from the set IAssn[4], and abstract predicates. We also define a set of *basic assertions*, BAssn, which are the standard separation-logic assertions, a set of *predicate assertions* Δ from the set PAssn, and a set of *function assertions* Γ from the set FAssn. These assertion sets are formally defined by:

$$\begin{aligned}
(\mathsf{Assn})\quad P, Q &::= \mathsf{emp} \mid E_1 \mapsto E_2 \mid P * Q \mid P \mathrel{-\!\circledast} Q \mid \mathsf{false} \mid P \Rightarrow Q \mid \exists x.\, P \mid \circledast x.\, P \mid \\
&\qquad \mathsf{all}(I, r) \mid [\gamma(E_1, \ldots, E_n)]^{r}_{\pi} \mid \boxed{P}^{r}_{I} \mid \alpha(E_1, \ldots, E_n)
\end{aligned}$$

$$(\mathsf{BAssn})\quad p, q ::= \mathsf{emp} \mid E_1 \mapsto E_2 \mid p * q \mid p \mathrel{-\!\circledast} q \mid \mathsf{false} \mid p \Rightarrow q \mid \exists x.\, p \mid \circledast x.\, p$$

$$(\mathsf{IAssn})\qquad I ::= \gamma(\vec{x})\colon \exists \vec{y}.\ (P \rightsquigarrow Q) \mid I_1, I_2$$

$$(\mathsf{PAssn})\qquad \Delta ::= \varnothing \mid \Delta, \forall \vec{x}.\, P \implies Q \mid \Delta, \forall \vec{x}.\, \alpha(\vec{x}) \equiv P \quad \text{s.t. } \alpha \notin \Delta$$

$$(\mathsf{FAssn})\qquad \Gamma ::= \varnothing \mid \Gamma, \{P\} f \{Q\} \quad \text{s.t. } f \notin \Gamma$$

Here γ ranges over the set of action names, AName; α ranges over the set of abstract predicate names, PName; x and y range over the set of logical variables, Var; and f ranges over the set of function names, FName. We assume an appropriate syntax for expressions, $E, r, \pi \in \mathsf{Expr}$, including basic arithmetic operators.

In Assn and BAssn, $p \mathrel{-\!\circledast} q$ is the existential version of the magic wand from separation logic, and $\circledast x.$ is the multiplicative analogue of $\forall x$. In the interface assertion $\gamma(\vec{x})\colon \exists \vec{y}.\ (P \rightsquigarrow Q)$, the action $\gamma(\vec{x})$ describes the transformation of the shared region satisfying P to a shared region satisfying Q. Actions may be parametrised: for example, in the ticketed-lock implementation, the Next(k) action releases the lock only when x.owner is k. The existential quantifier allows (unparametrised) variables to be shared between P and Q: for example, the Take action takes a ticket value k from x.next; it does

[4]In our proofs, we often refer to interference assertions by parameterised names, e.g. $I(r, x)$ in the proof of the CAS lock (§2.2). These parameterised names make the layout of the proof clearer, and are used purely as a short-hand for the full definitions. In all cases, they can be substituted for their definitions in-place. For clarity, we omit these names from the syntax.

not matter what ticket value it is, but it does matter that the resulting value of x.next is $k + 1$. Interface assertions are not unique: for example, see the action LGAP from the fine-grained set implementation. The abstract predicate assertions Δ consist of the axioms for predicates, $\forall \vec{x}.\, P \implies Q$, and concrete definitions of predicates, $\forall \vec{x}.\, \alpha(\vec{x}) \equiv P$. The function assertions Γ are Hoare triples specifying function behaviour.

4.2 Assertion Model

Let $(\mathsf{Heap}, \uplus, \emptyset)$ be any separation algebra [4] representing *machine states* (or *heaps*). Typically, we take Heap to be the standard heap model with with arbitrary element denoted h: that is, the set of finite partial functions from locations to values, where $\uplus$ is the union of partial functions with disjoint domains.

Our assertions include permissions which specify the possible interactions with shared regions. Hence, we define LState, the set of *logical states*, which pair heaps with permission assignments (elements of Perm, defined below):

$$l \in \mathsf{LState} \overset{\text{def}}{=} \mathsf{Heap} \times \mathsf{Perm}$$

We write l_{H} and l_{P} to stand for the heap and permission components respectively.

Assertions make an explicit (logical) division between shared state, which can be accessed by all threads, and thread-local state, which is private to a thread and cannot be subject to interference. Shared state is divided into *regions.* Intuitively, each region can be seen as the internal state of a single shared structure, i.e. a single lock, set, etc. Each region is identified by a *region name*, r, from the set RName. A region is also associated with an interference assertion, from the set IAssn, that determines how threads may modify the shared region. A *shared state* in SState is therefore a finite partial function mapping region names to logical states and interference assertions:

$$s \in \mathsf{SState} \overset{\text{def}}{=} \mathsf{RName} \overset{\text{fin}}{\rightharpoonup} (\mathsf{LState} \times \mathsf{IAssn})$$

A *world* in World is a pair of a local (logical) state and a shared state, subject to a well-formedness condition:

$$w \in \mathsf{World} \overset{\text{def}}{=} \{(l, s) \in \mathsf{LState} \times \mathsf{SState} \mid \mathsf{wf}(l, s)\}$$

Informally, the well-formedness condition ensures that all parts of the state are disjoint and that each permission corresponds to an appropriate region; we defer the formal definition of well-formedness. We write w_{L} and w_{S} to stand for the local and shared components respectively.

Permission assertions in our logic are satisfied by *permission assignments* in our model. Permission assignments in Perm assign *permission values* to actions, recording whether the particular action is allowed to the thread or environment. Recall from § 2.2 that actions can be self-referential. For example, the action UNLOCK moves the permission assertion $[\textsc{Unlock}]^r_1$ from local to shared state. Our semantics breaks the loop by distinguishing between the syntactic representation of an action and its semantics. Actions are represented syntactically by *tokens*, consisting of the region name, the action name and a finite sequence of value parameters:

$$t,\ (r, \gamma, \vec{v})\ \in \mathsf{Token} \overset{\text{def}}{=} \mathsf{RName} \times \mathsf{AName} \times \mathsf{Val}^*$$

For example, we have the token $(r, \textsc{Unlock}, ())$.

A permission assignment is a function associating each token with a *permission value* from the interval $[0,1]$ determining whether the associated action can occur:

$$pr \in \mathsf{Perm} \stackrel{\text{def}}{=} \mathsf{Token} \to [0,1]$$

Intuitively, 1 represents full, exclusive permission, 0 represents no permission, and the intermediate values represent partial, non-exclusive permission.[5] We write $\mathbf{0}_{\mathsf{Perm}}$ for the empty permission assignment mapping all tokens to 0, and $[t \mapsto \pi]$ for the permission mapping the token t to π and all other tokens to 0: for example, the permission assignment $[(r, \textsc{Unlock}, ()) \mapsto 1]$.

We now define composition of worlds. First, composition $\oplus$ on $[0,1]$ is defined as addition, with the proviso that the operator is undefined if the two permissions add up to more than 1. Composition on Perm is the obvious lifting: $pr \oplus pr' \stackrel{\text{def}}{=} \lambda t.\ pr(t) \oplus pr'(t)$. Composition on logical states is defined by lifting composition for heaps and permission assignments: $l \oplus l' \stackrel{\text{def}}{=} (l_{\mathrm{H}} \uplus l'_{\mathrm{H}},\ l_{\mathrm{P}} \oplus l'_{\mathrm{P}})$. Composition on worlds is defined by composing local states and requiring that shared states be identical:

$$w \oplus w' \stackrel{\text{def}}{=} \begin{cases} (w_{\mathrm{L}} \oplus w'_{\mathrm{L}}, w_{\mathrm{S}}) & \text{if } w_{\mathrm{S}} = w'_{\mathrm{S}} \\ \bot & \text{otherwise.} \end{cases}$$

We are nearly in a position to define well-formedness of worlds. First, we define the *action domain* of an interference assertion, $\mathrm{adom}(I)$, to be the set of action names and their appropriate parameters:

$$\begin{aligned} \mathrm{adom}(\gamma(x_1, \ldots, x_n) : \exists \vec{y}.\, (P \rightsquigarrow Q)) &\stackrel{\text{def}}{=} \{(\gamma, (v_1, \ldots, v_n)) \mid v_i \in \mathsf{Val}\} \\ \mathrm{adom}(I_1, I_2) &\stackrel{\text{def}}{=} \mathrm{adom}(I_1) \cup \mathrm{adom}(I_2) \end{aligned}$$

We also need the operator $\lfloor (l,s) \rfloor$ which collapses a pair of a logical state l and shared state s to a single logical state:

$$\lfloor (l,s) \rfloor \stackrel{\text{def}}{=} l \oplus \left(\textstyle\bigoplus_{r \in \mathrm{dom}(s)} s(r) \right)$$

where $\bigoplus$ is the natural lifting of $\oplus$ to sets. Finally, the well-formedness of worlds, $\mathsf{wf}(l,s)$, is given by:

$$\begin{aligned} \mathsf{wf}(l,s) \stackrel{\text{def}}{\iff}\ & \lfloor (l,s) \rfloor \text{ is defined } \wedge \\ & \forall r, \gamma, \vec{v}.\ \lfloor (l,s) \rfloor_{\mathrm{P}}(r, \gamma, \vec{v}) > 0 \implies \exists l', I.\, s(r) = (l', I) \wedge (\gamma, \vec{v}) \in \mathrm{adom}(I) \end{aligned}$$

4.3 Assertion Semantics

Fig. 10 presents the semantics of assertions, $[\![P]\!]_{\delta,i}$. We first define a weaker semantics $(\!|P|\!)_{\delta,i}$ that does not enforce well-formedness, then define $[\![P]\!]_{\delta,i}$ by restricting it to the set of

[5]This is the fractional permission model of Boyland [2]. With minimal changes we could add a *deny* permission prohibiting both the environment and thread from performing the action (see Dodds *et al.* [8]). We can achieve much the same effect in the Boyland-style system by placing a full permission in the shared state.

$$
\begin{array}{rcl}
(\!|-|\!)_{\_,\_} & : & \mathsf{Assn} \times \mathsf{PEnv} \times \mathsf{Interp} \to \mathcal{P}(\mathsf{LState} \times \mathsf{SState}) \\
(\!|E_1 \mapsto E_2|\!)_{\delta,i} & \overset{\text{def}}{=} & \left\{ (l,s) \,\middle|\, \begin{array}{r} \mathrm{dom}(l_{\mathrm{H}}) = \{[\![E_1]\!]_i\} \wedge l_{\mathrm{H}}([\![E_1]\!]_i) = [\![E_2]\!]_i \\ \wedge\, l_{\mathrm{P}} = \mathbf{0}_{\mathsf{Perm}} \wedge s \in \mathsf{SState} \end{array} \right\} \\
(\!|\mathsf{emp}|\!)_{\delta,i} & \overset{\text{def}}{=} & \{((\emptyset, \mathbf{0}_{\mathsf{Perm}}), s) \mid s \in \mathsf{SState}\} \\
(\!|P_1 * P_2|\!)_{\delta,i} & \overset{\text{def}}{=} & \{w_1 \oplus w_2 \mid w_1 \in (\!|P_1|\!)_{\delta,i} \wedge w_2 \in (\!|P_2|\!)_{\delta,i}\} \\
(\!|P_1 -\!\!\circledast P_2|\!)_{\delta,i} & \overset{\text{def}}{=} & \{w \mid \exists w_1, w_2.\, w_2 = w \oplus w_1 \wedge w_1 \in (\!|P_1|\!)_{\delta,i} \wedge w_2 \in (\!|P_2|\!)_{\delta,i}\} \\
(\!|\circledast x.\, P|\!)_{\delta,i} & \overset{\text{def}}{=} & \bigcup_W \left\{ \bigoplus_v W(v) \,\middle|\, \forall v.\, W(v) \in (\!|P|\!)_{\delta,i[x\mapsto v]} \right\} \\
(\!|\mathsf{all}(I,r)|\!) & \overset{\text{def}}{=} & \left\{ (\emptyset,\ \bigoplus_{(\gamma,\vec{v}) \in \mathrm{adom}(I)} [(r,\gamma,\vec{v}) \mapsto 1]) \right\} \\
(\!|[\gamma(E_1, \ldots E_n)]^r_\pi|\!)_{\delta,i} & \overset{\text{def}}{=} & \left\{ ((\emptyset, [([\![r]\!]_i, \gamma, [\![E_1]\!]_i, \ldots, [\![E_n]\!]_i) \mapsto [\![\pi]\!]_i]), s) \,\middle|\, \begin{array}{r} s \in \mathsf{SState} \wedge \\ [\![\pi]\!]_i \in (0,1] \end{array} \right\} \\
(\!|\boxed{P}^{\,r}_{I}|\!)_{\delta,i} & \overset{\text{def}}{=} & \{((\emptyset, \mathbf{0}_{\mathsf{Perm}}), s) \mid \exists l.\ (l,s) \in (\!|P|\!)_{\delta,i} \wedge s([\![r]\!]_i) = (l, [\![I]\!]_i)\} \\
(\!|\alpha(E_1, \ldots, E_n)|\!)_{\delta,i} & \overset{\text{def}}{=} & \delta(\alpha, [\![E_1]\!]_i, \ldots, [\![E_n]\!]_i)
\end{array}
$$

$$
\begin{array}{rcl}
[\![-]\!]_{\_,\_} & : & \mathsf{Assn} \times \mathsf{PEnv} \times \mathsf{Interp} \to \mathcal{P}(\mathsf{World}) \\
[\![P]\!]_{\delta,i} & \overset{\text{def}}{=} & \{(l,s) \in (\!|P|\!)_{\delta,i} \mid \mathsf{wf}((l,s))\}
\end{array}
$$

Figure 10: Semantics of assertions. The cases for conjunction, implication, existential, etc. are standard, simply distributing over the local and shared state.

well-formed worlds. The semantics of assertions depends on the semantics of expressions, $[\![-]\!]_\_ : \mathsf{Expr} \times \mathsf{Interp} \to \mathsf{Val}$. We have not formally defined this, and just assume an appropriate semantics. The semantics of interface assertions can also depend on the semantics of free variables. We define $[\![-]\!]_\_ : \mathsf{IAssn} \times \mathsf{Interp} \to \mathsf{IAssn}$ which simply replaces the free variables with their values.

The semantics is parameterised by a *predicate environment*, δ, mapping abstract predicates to their semantic definitions, and an *interpretation*, i, mapping logical variables to values:

$$\delta \in \mathsf{PEnv} \overset{\text{def}}{=} \mathsf{PName} \times \mathsf{Val}^* \to \mathcal{P}(\mathsf{World}) \qquad\qquad i \in \mathsf{Interp} \overset{\text{def}}{=} \mathsf{Var} \to \mathsf{Val}$$

We assume that $\mathsf{RName} \cup (0,1] \subset \mathsf{Val}$, so that variables may range over region names and (non-zero) permission values.

The semantics of the usual separation-logic connectives are standard. Notice that the semantics of the cell assertion $\mapsto$ and the empty assertion emp uses an arbitrary shared state s. The permission assertion $[\gamma(E_1, \ldots)]^r_\pi$ states that the token $([\![r]\!]_i, \gamma, [\![E_1]\!]_i \ldots)$ is associated with permission value $[\![\pi]\!]_i$. We restrict the permissions to those in $(0,1]$, since the 0 case is given by emp. A shared-state assertion $\boxed{P}^{\,r}_{I}$ asserts that P holds for region $[\![r]\!]_i$ in the shared state, and that the region's interference is given by the interference assertion, $[\![I]\!]_i$. For example, in the compare-and-swap lock implementation (§2.2), $\boxed{(x \mapsto 0 * [\textsc{Unlock}]^r_1) \vee x \mapsto 1}^{\,r}_{I(r,x)}$ asserts that the shared state for a lock is either unlocked with the full $\textsc{Unlock}$ permission, or locked. The interference assertion $I(r,x)$ defines the meaning of actions $\textsc{Lock}$ and $\textsc{Unlock}$, parameterised by region r and lock location x. Predicate assertions, $\alpha(E_1, ..., E_n)$, are mapped to sets of satisfying local and shared-state pairs by a predicate environment δ.

A shared state assertion $\boxed{P}_I^r$ defines the contents of an entire region. To maintain the same view of a region between threads, splitting a shared-state assertion must preserve the entire contents of the region. Consequently, separating conjunction behaves as conventional (non-separating) conjunction between shared-state assertions over the same region. That is:

$$\boxed{P}_I^r * \boxed{Q}_I^r \iff \boxed{P \wedge Q}_I^r$$

We permit nesting of shared-state assertions. However, our semantics for for shared state does not include nesting: the shared state is just a finite partial function $\mathsf{RName} \overset{\text{fin}}{\rightharpoonup} \mathsf{LState} \times \mathsf{IAssn}$. To resolve this, nested assertions are equivalent in our semantics to flattened assertions with all nested regions moved to the top level. Nested assertions can always be rewritten in an equivalent unnested form as follows:

$$\boxed{\boxed{P}_I^r * Q}_{I'}^{r'} \iff \boxed{P}_I^r * \boxed{Q}_{I'}^{r'}$$

(this even holds when when $r = r'$)

In this paper, we use nesting to ensure that shared and unshared elements can be referenced by a single abstract predicate. If an assertion is written using an abstract predicate, in some cases it is impossible to recover abstraction after unnesting the formula. For example, our compare-and-swap implementation of the lock module defines $\mathsf{Locked}(x)$ as:

$$\mathsf{Locked}(x) \equiv \exists r.\ [\textsc{Unlock}]_1^r * \boxed{x \mapsto 1}_{I(r,x)}^r$$

Suppose we now put this predicate into a region s. The following implications hold:

$$\begin{aligned} \boxed{\mathsf{Locked}(x)}_K^s &\iff \boxed{\exists r.\ [\textsc{Unlock}]_1^r * \boxed{x \mapsto 1}_{I(r,x)}^r}_K^s && \textit{(definition of } \mathsf{Locked}\textit{)} \\ &\iff \boxed{\exists r.\ [\textsc{Unlock}]_1^r}_K^s * \boxed{x \mapsto 1}_{I(r,x)}^r && \textit{(un-nesting)} \end{aligned}$$

In the final formula, we can no longer define a single abstract predicate capturing the information in the original Locked predicate. The shared state s is in the way.

In the future, nesting may also be useful for defining mutually recursive modules. We could define a pair of abstract predicates to refer to each other's shared state, and allow the semantics to resolve the recursion. This may give a modular way to reason disjointly about closely-related data-structures. Work on this idea is ongoing.

4.4 Environment Semantics

An interference assertion defines the actions that are permitted on a shared region. For example, in the compare-and-swap lock implementation (§2.2), the assertion $I(r, x)$ provides the action $\textsc{Lock} : x \mapsto 0 * [\textsc{Unlock}]_1^r \rightsquigarrow x \mapsto 1$. We interprete an interference assertion semantically using an *interference environment*, which is a map from tokens to sets of shared-state pairs:

$$[\![-]\!]_{\_} : \mathsf{IAssn} \times \mathsf{PEnv} \to \mathsf{Token} \to \mathcal{P}(\mathsf{SState} \times \mathsf{SState})$$

given by

$$[\![I_1, I_2]\!]_\delta(r, \gamma, \vec{v}) \overset{\text{def}}{=} [\![I_1]\!]_\delta(r, \gamma, \vec{v}) \cup [\![I_2]\!]_\delta(r, \gamma, \vec{v})$$

$$[\![\gamma(\vec{x})\colon \exists \vec{y}.\, (P \rightsquigarrow Q)]\!]_\delta(r, \gamma', \vec{v}) \overset{\text{def}}{=} \left\{ (s, s') \,\middle|\, \begin{array}{r} \gamma' = \gamma \wedge (\forall r' \neq r.\, s(r') = s'(r')) \\ \wedge\, \exists l, l', l_0, I.\, s(r) = (l \oplus l_0, I) \wedge \\ s'(r) = (l' \oplus l_0, I) \wedge \\ \exists \vec{v}'.\, (l, s) \in (\!|P|\!)_{\delta, [\vec{x} \mapsto \vec{v}, \vec{y} \mapsto \vec{v}']} \wedge \\ (l', s') \in (\!|Q|\!)_{\delta, [\vec{x} \mapsto \vec{v}, \vec{y} \mapsto \vec{v}']} \end{array} \right\}$$

The interference environment interprets thebinterface assertion $\gamma(\vec{x})\colon \exists \vec{y}.\, (P \rightsquigarrow Q)$ by mapping token $(r, \gamma, \vec{v})$ to a binary relation between shared states, such that the regions are identical except in the r case where it relates states satisfying $\boxed{P}^{r}_{I}$ to states satisfying $\boxed{Q}^{r}_{I}$. For example, for the action LOCK on shared region r, the interface environment relates the unlocked lock region with the locked region.

With our semantics of interference assertions, we are now in a position to define the rely, R_δ, which describes the updates the environment can perform, and the guarantee, G_δ, which describes the updates that the thread can perform. First, we give the guarantee. A thread can update its local state as it pleases, but any change to a shared region, r, must correspond to an action, $\gamma(\vec{v})$, for which the thread has sufficient permission, $(w_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \vec{v}) > 0$. For example, in the compare-and-swap lock proof (§2.2), the thread must hold permission 1 on UNLOCK before unlocking. Without this restriction, other threads could potentially unlock the lock.

It is important that each update preserves the total amount of permission in the world, that is $\lfloor w \rfloor_{\mathrm{P}} = \lfloor w' \rfloor_{\mathrm{P}}$ in the definition of G_δ below, so that threads cannot acquire permissions out of thin air. This does *not* hold for heaps, as we permit memory allocation. Moreover, as we saw in the compare-and-swap lock implementation, the thread can create a new shared region by giving away some of its local state and gaining full permission on the newly created region. This is described by G^{c} below. Conversely, a thread can destroy any region that it fully owns and grab ownership of the state it protects, as described by $(G^{\mathrm{c}})^{-1}$ below.

The guarantee G_δ is defined by:

$$G^{\mathrm{c}} \overset{\text{def}}{=} \left\{ (w, w') \,\middle|\, \begin{array}{r} \exists r, I, \ell_1, \ell_2.\;\; r \notin \mathrm{dom}(w_{\mathrm{S}}) \wedge w'_{\mathrm{S}} = w_{\mathrm{S}}[r \mapsto (\ell_1, I)] \wedge \\ w_{\mathrm{L}} = \ell_1 \oplus \ell_2 \wedge w'_{\mathrm{L}} = \ell_2 \oplus (\!|\mathsf{all}(I, r)|\!) \end{array} \right\}$$

$$G_\delta \overset{\text{def}}{=} \left\{ (w, w') \,\middle|\, \begin{array}{l} ((\exists r, \gamma, \vec{v}.\, (w_{\mathrm{S}}, w'_{\mathrm{S}}) \in [\![(w_{\mathrm{S}}(r))_2]\!]_\delta(r, \gamma, \vec{v}) \wedge \\ (w_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \vec{v}) > 0) \vee w_{\mathrm{S}} = w'_{\mathrm{S}}) \wedge \lfloor w \rfloor_{\mathrm{P}} = \lfloor w' \rfloor_{\mathrm{P}} \end{array} \right\} \cup G^{\mathrm{c}} \cup (G^{\mathrm{c}})^{-1}$$

Some permitted updates do not modify the heap, but simply repartition it between shared regions. This is captured by $\overline{G}_\delta \overset{\text{def}}{=} G_\delta \cap \{(w, w') \mid \lfloor w \rfloor_H = \lfloor w' \rfloor_H\}$. In practice, we allow an unlimited number of repartitionings in a single step, only one of which actually modifies the heap. This is captured by $\widehat{G}_\delta$, defined as:

$$\widehat{G}_\delta \overset{\text{def}}{=} (\overline{G}_\delta)^*; G_\delta; (\overline{G}_\delta)^*$$

Using the guarantee, we define the notion of *repartitioning* with respect to an update from p to q, written $P \Longrightarrow^{\{p\}\{q\}}_{\delta} Q$. The guarantee states how a thread can change the shared regions, and can create and destroy shared regions if the thread has the full permissions. The repartitioning states how the local and shared heap is updated, so that it is compatible with the update from p to q as well as being compatible with the guarantee.

Definition 4.1 (Repartitioning). $P \Longrightarrow_{\delta}^{\{p\}\{q\}} Q$ holds if and only if, for every variable interpretation i and world w_1 in $[\![P]\!]_{\delta,i}$, there exists a heap h_1 in $[\![p]\!]_i$ and a residual heap h' such that

- $h_1 \oplus h' = \lfloor w_1 \rfloor_H$; and
- for every heap h_2 in $[\![q]\!]_i$, there exists a world w_2 in $[\![Q]\!]_{\delta,i}$ such that
  - $h_2 \oplus h' = \lfloor w_2 \rfloor_H$; and
  - the update is allowed by the guarantee: that is, $(w_1, w_2) \in \widehat{G}_\delta$.

Note that, if $p = q = \mathsf{emp}$, then the repartitioning preserves the concrete state and only allows the world to be repartitioned. We write $P \Longrightarrow_{\delta} Q$ as a shorthand for $P \Longrightarrow_{\delta}^{\{\mathsf{emp}\}\{\mathsf{emp}\}} Q$.

The rely R_δ describes the possible world updates that the environment can do. Intuitively, it models the interference from other threads. At any point, it can update the shared state by performing one of the actions in any one of the shared regions r, provided that the environment potentially has permission to perform that action. For this to be possible, the world must contain less than the total permission ($\lfloor w \rfloor_{\mathrm{P}}(r, \gamma, \vec{v}) < 1$). This models the fact that some other thread's local state could contain permission $\pi > 0$ on the action. In addition, the environment can create a new region (using R^{c}) or can destroy an existing region (using $(R^{\mathrm{c}})^{-1}$) provided that no permission for that region exists elsewhere in the world.

The rely R_δ is defined by:

$$R^{\mathrm{c}} \overset{\mathrm{def}}{=} \left\{ (w, w') \;\middle|\; \begin{array}{r} \exists r, \ell, I.\, r \notin \mathrm{dom}(w_{\mathrm{S}}) \wedge w'_{\mathrm{L}} = w_{\mathrm{L}} \wedge w'_{\mathrm{S}} = w_{\mathrm{S}}[r \mapsto (\ell, I)] \wedge \\ \lfloor w' \rfloor \text{ defined} \wedge (\forall \gamma, \vec{v}.\, \lfloor w' \rfloor_{\mathrm{P}}(r, \gamma, \vec{v}) = 0) \end{array} \right\}$$

$$R_\delta \overset{\mathrm{def}}{=} \left\{ (w, w') \;\middle|\; \begin{array}{r} \exists r, \gamma, \vec{v}.\, (w_{\mathrm{S}}, w'_{\mathrm{S}}) \in [\![(w_{\mathrm{S}}(r))_2]\!]_\delta(r, \gamma, \vec{v}) \wedge \\ w_{\mathrm{L}} = w'_{\mathrm{L}} \wedge \lfloor w \rfloor_{\mathrm{P}}(r, \gamma, \vec{v}) < 1 \end{array} \right\} \cup R^{\mathrm{c}} \cup (R^{\mathrm{c}})^{-1}$$

The rely R_δ enables us to define the stability of assertions. An assertion is *stable* if and only if it cannot be falsified by the interference from other threads that it permits.

Definition 4.2 (Stability). $\mathsf{stable}_\delta(P)$ holds iff for all w, w' and i, if $w \in [\![P]\!]_{\delta,i}$ and $(w, w') \in R_\delta$, then $w' \in [\![P]\!]_{\delta,i}$.

Similarly, a predicate environment is stable if and only if all the predicates it defines are stable.

Definition 4.3 (Predicate Environment Stability). $\mathsf{pstable}(\delta)$ holds iff for all $X \in \mathrm{ran}(\delta)$, for all w and w', if $w \in X$ and $(w, w') \in R_\delta$, then $w' \in X$.

A syntactic predicate environment, Δ, is defined in the semantics as a set of stable predicate environments:

$$\begin{aligned} [\![\varnothing]\!] &\overset{\mathrm{def}}{=} \{\delta \mid \mathsf{pstable}(\delta)\} \\ [\![\Delta,\, \forall \vec{x}.\, \alpha(\vec{x}) \equiv P]\!] &\overset{\mathrm{def}}{=} [\![\Delta]\!] \cap \{\delta \mid \mathsf{pstable}(\delta) \wedge \forall \vec{v}.\ \delta(\alpha, \vec{v}) = [\![P]\!]_{\delta,[\vec{x} \mapsto \vec{v}]}\} \\ [\![\Delta,\, \forall \vec{x}.\, P \Rightarrow Q]\!] &\overset{\mathrm{def}}{=} [\![\Delta]\!] \cap \{\delta \mid \mathsf{pstable}(\delta) \wedge \forall \vec{v}.\ [\![P]\!]_{\delta,[\vec{x} \mapsto \vec{v}]} \subseteq [\![Q]\!]_{\delta,[\vec{x} \mapsto \vec{v}]}\} \end{aligned}$$

4.5 Programming Language and Proof System

We define a proof system for deriving local Hoare triples for a simple concurrent imperative programming language of commands:

$$\begin{array}{ll}(\mathsf{Cmd}) & C ::= \mathbf{skip} \mid c \mid f \mid \langle C \rangle \mid C_1; C_2 \mid C_1 + C_2 \mid C^* \mid C_1 \| C_2 \mid \\ & \quad\quad \text{let } f_1 = C_1 \ldots f_n = C_n \text{ in } C\end{array}$$

We require that atomic statements $\langle C \rangle$ are not nested and that function names $f_1 \ldots f_n$ for a let are distinct. Here c stands for basic commands, modelled semantically as subsets of $\mathcal{P}(\mathsf{Heap} \times \mathsf{Heap})$ satisfying the locality conditions of [4].

Judgements about such programs have the form $\Delta; \Gamma \vdash \{P\}\, C\, \{Q\}$, which asserts that, beginning in a state satisfying P, interpreted under predicate definitions satisfying Δ, the program C using procedures specified by Γ will not fault and, if it terminates, the final state will satisfy Q.

The proof rules for our Hoare-style program logic are shown in Fig. 11. These rules are modified from RGSep [24] and deny-guarantee [8]. All of the rules in our program logic carry a implicit assumption that the pre- and post-conditions of their judgements are stable.

The judgement $\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\}$ appearing in Atomic and Prim is a judgement in standard sequential separation logic. The other minor judgements are defined semantically to quantify over all $\delta \in [\![\Delta]\!]$: $\Delta \vdash P \Longrightarrow^{\{p\}\{q\}} Q$ means $\forall \delta \in [\![\Delta]\!] \,.\, P \Longrightarrow_{\delta}^{\{p\}\{q\}} Q$ (and similarly without a superscript); $\Delta \vdash \mathsf{stable}(P)$ means $\forall \delta \in [\![\Delta]\!] \,.\, \mathsf{stable}_{\delta}(P)$; and $\Delta \vdash \Delta'$ means $[\![\Delta]\!] \subseteq [\![\Delta']\!]$.

To reason about predicate assumptions, we introduce two rules, Pred-I and Pred-E. The Pred-I rule allows the assumptions about the predicate definitions to be weakened. If a triple is provable with assumptions Δ', then it must be provable under stronger assumptions Δ. The Pred-E rule allows the introduction of predicate definitions. For this to be sound, the predicate name α must not be used anywhere in the existing definitions and assertions. We require that recursively-defined predicate definitions are satisfiable; otherwise the premise of a proof rule could be vacuously true. We ensure this by requiring that all occurrences of the predicate in its definition are positive.

The Par rule is the key rule for disjoint concurrency. In this rule, we exploit our fiction of disjointness to prove safety for concurrent programs. Our set-up allows us to define a simple parallel rule while capturing fine-grained interactions. The Atomic and Conseq rule were discussed in §2.3. That section also discussed a rule for modules, which can be derived as follows:

$$\dfrac{\dfrac{\Delta \vdash \{P_1\}C_1\{Q_1\} \qquad \cdots \qquad \dfrac{\Delta \vdash \Delta' \quad \Delta'; \{P_1\}f_1\{Q_1\}, \ldots \vdash \{P\}C\{Q\}}{\Delta; \{P_1\}f_1\{Q_1\}, \ldots \vdash \{P\}C\{Q\}}\ \text{Pred-I}}{\Delta \vdash \{P\} \text{ let } f_1 = C_1 \ldots f_n = C_n \text{ in } C\ \{Q\}}\ \text{Let}}{\vdash \{P\} \text{ let } f_1 = C_1 \ldots f_n = C_n \text{ in } C\ \{Q\}}\ \text{Pred-E}$$

Deriving this rule in this way reveals a side-condition: the predicate names defined in Δ must not appear in P or Q. Intuitively, the module's predicates must not appear outside the scope of the module definition.

$$\frac{}{\Delta; \Gamma \vdash \{P\}\ \mathbf{skip}\ \{P\}}\ (\textsc{Skip}) \qquad \frac{\Delta; \Gamma \vdash \{P\}\ C_1\ \{P'\} \quad \Delta; \Gamma \vdash \{P\}\ C_2\ \{P'\}}{\Delta; \Gamma \vdash \{P\}\ C_1 + C_2\ \{P'\}}\ (\textsc{Choice})$$

$$\frac{\Delta; \Gamma \vdash \{P\}\ C_1\ \{P''\} \quad \Delta; \Gamma \vdash \{P''\}\ C_2\ \{P'\}}{\Delta; \Gamma \vdash \{P\}\ C_1;\ C_2\ \{P'\}}\ (\textsc{Seq}) \qquad \frac{\Delta; \Gamma \vdash \{P\}\ C\ \{P\}}{\Delta; \Gamma \vdash \{P\}\ C^*\ \{P\}}\ (\textsc{Loop})$$

$$\frac{\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\} \quad \Delta \vdash P \Rrightarrow^{\{p\}\{q\}} Q}{\Delta; \Gamma \vdash \{P\}\ \langle C \rangle\ \{Q\}}\ (\textsc{Atomic}) \qquad \frac{\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\}}{\Delta; \Gamma \vdash \{p\}\ C\ \{q\}}\ (\textsc{Prim})$$

$$\frac{\Delta; \Gamma \vdash \{P_1\}\ C_1\ \{Q_1\} \quad \Delta; \Gamma \vdash \{P_2\}\ C_2\ \{Q_2\}}{\Delta; \Gamma \vdash \{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\ (\textsc{Par}) \qquad \frac{\{P\}\ f\ \{Q\} \in \Gamma}{\Delta; \Gamma \vdash \{P\}\ f\ \{Q\}}\ (\textsc{Call})$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \{P\}\ C\ \{Q\} \\ \Delta \vdash \mathsf{stable}(R)\end{array}}{\Delta; \Gamma \vdash \{P * R\}\ C\ \{Q * R\}}\ (\textsc{Frame}) \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash \{P'\}\ C\ \{Q'\} \\ \Delta \vdash P \Rrightarrow P' \quad \Delta \vdash Q' \Rrightarrow Q\end{array}}{\Delta; \Gamma \vdash \{P\}\ C\ \{Q\}}\ (\textsc{Conseq})$$

$$\frac{\Delta; \Gamma \vdash \{P_1\}\ C\ \{Q\} \quad \Delta; \Gamma \vdash \{P_2\}\ C\ \{Q\}}{\Delta; \Gamma \vdash \{P_1 \vee P_2\}\ C\ \{Q\}}\ (\textsc{Disj}) \qquad \frac{\begin{array}{c}x \notin \mathsf{fv}(\Delta, \Gamma, P, C, Q) \\ \Delta; \Gamma \vdash \{P\}\ C\ \{Q\}\end{array}}{\Delta; \Gamma \vdash \{\exists x.\, P\}\ C\ \{Q\}}\ (\textsc{Ex})$$

$$\frac{\Delta \vdash \Delta' \quad \Delta'; \Gamma \vdash \{P\}C\{Q\}}{\Delta; \Gamma \vdash \{P\}C\{Q\}}\ (\textsc{Pred-I}) \qquad \frac{\begin{array}{c}\Delta \vdash \mathsf{stable}(R) \quad \alpha \notin \Delta, \Gamma, P, Q \\ \Delta, (\forall \vec{x}.\ \alpha(\vec{x}) \equiv R); \Gamma \vdash \{P\}C\{Q\}\end{array}}{\Delta; \Gamma \vdash \{P\}C\{Q\}}\ (\textsc{Pred-E})$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \{P_1\}\ C_1\ \{Q_1\} \quad \ldots \quad \Delta; \Gamma \vdash \{P_n\}\ C_n\ \{Q_n\} \\ \Delta; \{P_1\}\ f_1\ \{Q_1\}, \ldots, \{P_n\}\ f_n\ \{Q_n\}, \Gamma \vdash \{P\}\ C\ \{Q\}\end{array}}{\Delta; \Gamma \vdash \{P\}\ \text{let}\ f_1 = C_1 \ldots f_n = C_n\ \text{in}\ C\ \{Q\}}\ (\textsc{Let})$$

Figure 11: Proof rules. All rules assume that the pre- and post-conditions of their judgements are stable.

4.6 Judgement Semantics and Soundness

We define the meaning of judgements in our proof system with respect to a transition relation $C, h \xrightarrow{\eta} C', h'$ defining the operational semantics of our language. The transition is parameterised with a *function environment*, η, mapping function names to their definitions. We also define a faulting relation $C, h \xrightarrow{\eta} \mathit{fault}$.

$$\begin{array}{rcll} \eta & \in \mathsf{FEnv} & \stackrel{\text{def}}{=} & \mathsf{FName} \to \mathsf{Cmd} \\ \xrightarrow{-} & \in \mathsf{OpTrans} & \stackrel{\text{def}}{=} & \mathcal{P}(\mathsf{FEnv} \times \mathsf{Cmd} \times \mathsf{Heap} \times \mathsf{Cmd} \times \mathsf{Heap}) \\ \xrightarrow{-} \mathit{fault} & \in \mathsf{OpFault} & \stackrel{\text{def}}{=} & \mathcal{P}(\mathsf{FEnv} \times \mathsf{Cmd} \times \mathsf{Heap}) \end{array}$$

To define the meaning of judgements, we first define the notion of a logical configuration $(C, w, \eta, \delta, i, Q)$ being safe for at least n steps. Intuitively, safety for n steps means that a derivation of length n consisting of steps from the thread and environment does not fault, respects the guarantee at each step, and terminates only in a state satisfying the post-condition.

The soundness proof for our logic is based on reasoning about configuration safety. For example, to show the soundness of the parallel rule, we must show that if safety holds for two configurations for programs C_1 and C_2, then the compositions of the two configurations is safe for program $C_1 \| C_2$.

Definition 4.4 (Configuration safety). C, w, η, δ, i, Q safe$_0$ always holds; and $C, w, \eta, \delta, i,$ safe$_{n+1}$ iff the following four conditions hold:

1. $\forall w'$, if $(w, w') \in (R_\delta)^*$ then $C, w', \eta, \delta, i, Q$ safe$_n$;
2. $\neg((C, \lfloor w \rfloor_H) \xrightarrow{\eta} \mathit{fault})$;
3. $\forall C', h'$, if $(C, \lfloor w \rfloor_H) \xrightarrow{\eta} (C', h')$, then there $\exists w'$ such that $(w, w') \in \widehat{G}_\delta$, $h' = \lfloor w' \rfloor_H$ and $C', w', \eta, \delta, i, Q$ safe$_n$; and
4. if C=**skip**, then $\exists w'$ such that $\lfloor w \rfloor_H = \lfloor w' \rfloor_H$, $(w, w') \in \widehat{G}_\delta$, and $w' \in [\![Q]\!]_{\delta,i}$.

This definition says that a configuration is safe provided that: (1) changing the world in a way that respects the rely is still safe; (2) the program cannot fault; (3) if the program can make a step, then there should be an equivalent step in the logical world that is allowed by the guarantee; and (4) if the configuration has terminated, then the post-condition should hold. The use of $\widehat{G}_\delta$ in the third and fourth conjuncts allows the world to be repartitioned.

Definition 4.5 (Judgement Semantics). $\Delta; \Gamma \models \{P\}\, C\, \{Q\}$ holds iff

$$\forall i, n.\ \forall \delta \in [\![\Delta]\!]\,.\ \forall \eta \in [\![\Gamma]\!]_{n,\delta,i}\,.\ \forall w \in [\![P]\!]_{\delta,i}\,.\ C, w, \eta, \delta, i, Q \text{ safe}_{n+1}\,,$$

where $[\![\Gamma]\!]_{n,\delta,i} \overset{\text{def}}{=} \{\eta \mid \forall \{P\} f \{Q\} \in \Gamma.\ \forall w \in [\![P]\!]_{\delta,i}\,.\ \eta(f), w, \eta, \delta, i, Q \text{ safe}_n\}$.

Theorem 4.6 (Soundness). *If* $\Delta; \Gamma \vdash \{P\}\, C\, \{Q\}$, *then* $\Delta; \Gamma \models \{P\}\, C\, \{Q\}$.

Proof is by by structural induction on derivations. See Appendix B for full details. The most interesting case is the Par rule, which embodies the compositionality of our logic. The proof requires the following lemma.

Lemma 4.7 (Abstract state locality). *If* $(C, \lfloor w_1 \oplus w_2 \rfloor_H) \xrightarrow{\eta} (C', h)$ *and* $C, w_1, \eta, \delta, i, Q$ *safe*$_n$, *then* $\exists w_1', w_2'$ *such that* $(C, \lfloor w_1 \rfloor_H) \xrightarrow{\eta} (C', \lfloor w_1' \rfloor_H)$, $h = \lfloor w_1' \oplus w_2' \rfloor_H$, $(w_1, w_1') \in \widehat{G}_\delta$, *and* $(w_2, w_2') \in (R_\delta)^*$.

Proof. We require that basic commands obey a concrete locality assumption. We must prove that the rely and guarantee obey similar locality lemmas. The lemma then follows from the definition of configuration safety. The full proof is given in Appendix B, Lemma B.11. □

This lemma shows that program only affects the resources required for it to run safely: that is, programs are safely contained within their abstract footprints. The soundness of Par follows immediately.

5 Conclusions and Related Work

Our program logic allows fine-grained abstraction in a concurrent setting. It brings together three streams of research: (1) abstract predicates [21] for abstracting the internal details of a module or class; (2) deny-guarantee [8] for reasoning about concurrent programs; and (3) context logic [3, 7] for fine-grained reasoning at the module level.

Our work on concurrent abstract predicates has been strongly influenced by O'Hearn's concurrent separation logic (CSL) [19]. CSL takes statically allocated locks as a primitive. With CSL, we can reason about programs with locks as if they are disjoint from each other, even though they interfere on a shared state. CSL therefore provides a key example of the fiction of disjointness. The CSL approach has been extended with new proof rules and assertions to deal with dynamically-allocated locks [11, 15] and re-entrant locks [12]. Parkinson *et al.* [20] have shown how the CSL approach can be used to reason about concurrent algorithms that do not use locks. However, representing the concurrent interactions in an invariant can require intractable auxiliary state.

Jacobs and Piessens [17] have developed an approach to reasoning abstractly that is based on CSL for dynamically allocated locks [11]. Their logic uses auxiliary state to express the temporal nature of interference. To deal modularly with auxiliary state they add a special implication that allows the auxiliary state to be changed in any way that satisfies the invariant. This implication is similar to our repartitioning operator $\Rrightarrow$. Unlike our operator, theirs can be used in a module specification, allowing a client's auxiliary state to be updated during the module's execution. Our operator could be extended with this technique, which may simplify the use of the lock specification in the set algorithms.

An alternative to using invariants is to abstract interference over the shared state by relations modelling the interaction of different threads: the rely-guarantee method [18]. There have been two recent logics that combine RG with separation logic: RGSep [24] and SAGL [10]. Both allow more elegant proofs of concurrent algorithms than the invariant-based approaches, but they have the drawback that interference on the shared state cannot be abstracted. Feng's Local Rely-Guarantee [9] improves the locality of RGSep and SAGL by scoping interference with a precise footprint, but this approach still has no support for abstraction. Many of our ideas on abstraction originated in Dinsdale-Young, Gardner and Wheelhouse's work on using RGSep to analyse a concurrent B-tree algorithm [6, 22].

We have combined RGSep with resource permissions, as first introduced for deny-guarantee reasoning [8]. Deny-guarantee is a reformulation of rely-guarantee allowing reasoning about dynamically scoped concurrency. Deny-guarantee reasoning is related to the 'state guarantees' of Bierhoff et al. [1] in linear logic, which are also splittable permissions describing how a state can be updated.

We have also used ideas from context logic [3], a high-level logic for fine-grained local reasoning about program modules. Recent work in context logic has shown how implementations of modules can be verified, by relating local reasoning about module specifications with the low-level reasoning about implementations [7]. As presented here, these implementations break away from the fiction of disjointness presented by the module specifications.

Proofs in our proof system are slightly more complex in practice than RGSep and SAGL, as can be seen by comparing the lock-coupling list proof with the RGSep one [24].

This may be the penalty that we pay for having greater modularity although, as we acquire more experience with doing proofs using concurrent abstract predicates, we expect to be able to reduce this complexity.

An alternative approach to abstracting concurrent algorithms is to use linearisability [14]. Linearisability provides a fiction of atomicity allowing "reason[ing] about properties of concurrent objects given just their (sequential) specifications" [14]. With linearisability, we are forced to step outside the program logic at module boundaries and fall back on operational reasoning. In contrast, with concurrent abstract predicates we are able to write modular proofs within a single logical formalism.

Acknowledgements Thanks to Richard Bornat, Alexey Gotsman, Peter O'Hearn, Suresh Jagannathan, Mohammad Raza, Noam Rinetzky, Mark Wheelhouse, John Wickerson and the anonymous referees of our conference paper [5] for useful feedback. This work was supported by an EPSRC DTA (Dinsdale-Young), an EPSRC Programme Grant EP/H008373/1 (Dinsdale-Young and Gardner), an EPSRC grant EP/F019394/1 (Dodds and Parkinson), an RAEng/EPSRC research fellowship (Parkinson) and a Microsoft Research Cambridge/RAEng senior research fellowship (Gardner).

References

[1] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320, 2007.

[2] J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.

[3] C. Calcagno, P. Gardner, and U. Zarfaty. Local reasoning about data update. *Festschrift* Computation, Meaning and Logic: Articles dedicated to Gordon Plotkin, 172, 2007.

[4] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Symp. on Logic in Comp. Sci. (LICS'07)*, pages 366–378, 2007.

[5] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[6] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Local reasoning about a concurrent B*-list algorithm. Talk and unpublished report, see `http://www.doc.ic.ac.uk/~pg/`, 2009.

[7] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Locality refinement. Technical Report DTR10-8, Imperial College London, 2010.

[8] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

[9] X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

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

[11] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.

[12] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's Reentrant Locks. In *APLAS*, pages 171–187, 2008.

[13] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Elsevier, 2008.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[15] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[16] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, Jan. 2001.

$$\frac{(C,h) \xrightarrow{\eta[f_1 \mapsto C_1 \ldots f_n \mapsto C_n]} (C',h')}{(\text{let } f_1 = C_1 \ldots f_n = C_n \text{ in } C, h) \xrightarrow{\eta} (\text{let } f_1 = C_1 \ldots f_n = C_n \text{ in } C', h')}$$

$$\frac{}{(\text{let } \ldots \text{ in } \mathbf{skip}, h) \xrightarrow{\eta} (\mathbf{skip}, h)} \qquad \frac{f \in \text{dom}(\eta)}{(f,h) \xrightarrow{\eta} (\eta(f), h)} \qquad \frac{(h,h') \in c \quad h' \neq \texttt{fault}}{(c,h) \xrightarrow{\eta} (\mathbf{skip}, h')}$$

$$\frac{(C,h) \xrightarrow{\eta} (C_1,h')}{(C;C',h) \xrightarrow{\eta} (C_1;C',h')} \qquad \frac{}{(\mathbf{skip};C,h) \xrightarrow{\eta} (C,h)} \qquad \frac{}{(C^*,h) \xrightarrow{\eta} (\mathbf{skip} + (C;C^*), h)}$$

$$\frac{}{(C_1 + C_2, h) \xrightarrow{\eta} (C_1, h)} \qquad \frac{}{(C_1 + C_2, h) \xrightarrow{\eta} (C_2, h)} \qquad \frac{(C,h) \xrightarrow{\eta}^{*} (\mathbf{skip}, h')}{(\langle C \rangle, h) \xrightarrow{\eta} (\mathbf{skip}, h')}$$

$$\frac{(C_1,h) \xrightarrow{\eta} (C_1',h')}{(C_1 \| C_2, h) \xrightarrow{\eta} (C_1' \| C_2, h')} \qquad \frac{(C_2,h) \xrightarrow{\eta} (C_2',h')}{(C_1 \| C_2, h) \xrightarrow{\eta} (C_1 \| C_2', h')} \qquad \frac{}{(\mathbf{skip} \| \mathbf{skip}, h) \xrightarrow{\eta} (\mathbf{skip}, h)}$$

$$\frac{(C_1,h) \xrightarrow{\eta} \mathit{fault}}{(C_1;C_2,h) \xrightarrow{\eta} \mathit{fault}} \qquad \frac{(C_1,h) \xrightarrow{\eta} \mathit{fault}}{(C_1 \| C_2,h) \xrightarrow{\eta} \mathit{fault}} \qquad \frac{(C_2,h) \xrightarrow{\eta} \mathit{fault}}{(C_1 \| C_2,h) \xrightarrow{\eta} \mathit{fault}}$$

$$\frac{f \notin \text{dom}(\eta)}{(f,h) \xrightarrow{\eta} \mathit{fault}} \qquad \frac{(h, \texttt{fault}) \in c}{(c,h) \xrightarrow{\eta} \mathit{fault}} \qquad \frac{(C,h) \xrightarrow{\eta}^{*} \mathit{fault}}{(\langle C \rangle, h) \xrightarrow{\eta} \mathit{fault}}$$

Figure 12: Operational semantics.

[17] B. Jacobs and F. Piessens. Modular full functional specification and verification of lock-free data structures. Technical Report CW 551, Katholieke Universiteit Leuven, Department of Computer Science, June 2009.

[18] C. B. Jones. Annoted bibliography on rely/guarantee conditions. `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf`, 2007.

[19] P. W. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.

[20] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, pages 297–302, Jan. 2007.

[21] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

[22] P. Pinto. Reasoning about B^{Link} trees. Advanced masters ISO project, Imperial College London, 2010. Supervised by Dinsdale-Young, Gardner and Wheelhouse.

[23] V. Vafeiadis. *Modular Fine-Grained Concurrency Verification.* PhD thesis, University of Cambridge, July 2007.

[24] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

A Language Semantics

The operational semantics of our programming language is given in Fig. 12. This semantics is largely taken from [23]. However, this prior semantics required two operational semantics – a logical semantics with proof annotations to separate local and shread state and a machine semantics erasing such annotations. Using Vafeiadis's new approach to soundness of RGSep, Definition 4.4, we can avoid this requirement for two semantics.

We assume a simple Dijkstra-style language with constructs for sequential composition, nondeterministic choice and looping. Programs in the given in paper can be encoded into this language for proof purposes. Primitive updates in this language are modelled by relations $c \subseteq \mathsf{States} \times (\mathsf{States} \cup \{\mathtt{fault}\})$. We require that each of the primitive updates of our language is local.

Assumption 1 (Primitive locality)**.** *Each primitive update, c, satisfies the following conditions:*

1. *If $(h \oplus h', \mathtt{fault}) \in c$, then $(h, \mathtt{fault}) \in c$, and*
2. *if $(h \oplus h', h_2) \in c$ and $(h, \mathtt{fault}) \notin c$ then there exists an h'_2 such that $(h, h'_2) \in c$ and $h'_2 \oplus h' = h_2$.*

B Soundness of proof system

In order to prove the soundness of our proof system, we must show that each proof rule represents a valid inference. Typically, we show that the conclusion of each proof rule is semantically valid (under the assumption that the premisses are semantically valid) by induction on the number of safe steps we wish to take from an appropriate arbitrary configuration.

The first case we consider is the Skip rule, which is justified by the following lemma. The lemma is also used to deal with other rules in cases where the program reduces in one step to **skip**, such as $\langle C \rangle$ and **skip**||**skip**.

Lemma B.1 (Skip safety)**.** *If $\mathsf{stable}_\delta(Q)$ and $w \in [\![Q]\!]_{\delta,i}$ then* **skip**, w, η, δ, i, Q *safe*$_n$.

Proof. Prove by induction on n. The 0 case holds by definition. Consider now the inductive case. By the definition of stability, any w' such that $(w, w') \in (R_\delta)^*$ must satisfy $w' \in [\![Q]\!]_{\delta,i}$. Consequently the first clause holds by the inductive assumption. Clauses 2 and 3 hold trivially, as no reductions from **skip** exist in the semantics. Clause 4 holds trivially by picking w' as w. □

Locality is essential for the soundness of the frame rule and disjoint concurrency. We required it of our primitive update commands, and the following two lemmas show that it holds also for the operational semantics of programs, both when executed for a single step or for multiple steps. The locality properties capture the intuition that a program operates within a footprint, independent of any additional state. The two lemmas deal only with the heap state, and not with the abstract logical state.

Lemma B.2 (Concrete locality)**.** *Locality holds for the operational semantics over single steps and multiple steps.*

- *If $(C, h \oplus h') \xrightarrow{\eta} (C_2, h_2)$ and $\neg((C, h) \xrightarrow{\eta} \mathit{fault})$, then $\exists h'_2$ such that $(C, h) \xrightarrow{\eta} (C_2, h'_2)$ and $h'_2 \oplus h' = h_2$.*

- *If* $(C, h \oplus h') \xrightarrow{\eta}{}^{*} (C_2, h_2)$ *and* $\neg((C, h) \xrightarrow{\eta}{}^{*} \mathit{fault})$*, then* $\exists h_2'$ *such that* $(C, h) \xrightarrow{\eta}{}^{*} (C_2, h_2')$ *and* $h_2' \oplus h' = h_2$*.*

Proof. Proved by induction over the structure of a derivation or derivation sequence.

Consider the first case, consisting of a single derivation $(C, h \oplus h') \xrightarrow{\eta} (C_2, h_2)$. For most of the rules, locality holds simply by the first clause of the inductive assumption. For the two rules dealing with primitive updates, locality is ensured by Assumption 1. For the rule dealing with atomic statements, locality is ensured by the second clause of the inductive assumption.

Now consider the second case, where we have a multi-step derivation $(C, h \oplus h') \xrightarrow{\eta}{}^{*} (C_2, h_2)$. Any such derivation consists of k single-step derivations. We can thus prove the result by appeal to the two inductive assumptions. □

Lemma B.3 (Concreate fault locality). *Fault locality holds for the operational semantics over single steps and multiple steps.*

- *If* $(C, h \oplus h') \xrightarrow{\eta} \mathit{fault}$*, then* $(C, h) \xrightarrow{\eta} \mathit{fault}$*.*
- *If* $(C, h \oplus h') \xrightarrow{\eta}{}^{*} \mathit{fault}$*, then* $(C, h) \xrightarrow{\eta}{}^{*} \mathit{fault}$*.*

Proof. The result is proved by induction over the structure of a faulting derivation or derivation sequence.

The first clause holds when we have a single-step faulting derivation, $(C, h \oplus h') \xrightarrow{\eta} \mathit{fault}$. For most of the rules, locality holds immediately by the first clause of the inductive assumption. For the rule dealing with primitive update, locality is ensured by Assumption 1, as in the previous lemma. For the rule dealing with atomic statements, locality holds by the second clause of the inductive assumption.

To prove the second clause, we observe that the shortest faulting derivation sequence $(C, h \oplus h') \xrightarrow{\eta}{}^{*} \mathit{fault}$ must consist of a length-k derivation $(C, h \oplus h') \xrightarrow{\eta}{}^{k} (C', h'')$ and a single derivation $(C', h'') \xrightarrow{\eta} \mathit{fault}$. We can thus show the result by applying Lemma B.2 and the first inductive assumption. □

The following guarantee locality lemmas capture the notion that, if the guarantee permits some action, then the action that does the same thing but leaves some additional disjoint state unchanged is also permitted by the guarantee. For justifying the parallel composition and frame rules, this additional state corresponds to the state of another thread and the frame, respectively.

Lemma B.4 (Guarantee locality I). *If* $(w_1, w_1') \in \widehat{G}_\delta$ *and* $w_1 \oplus w_2$ *and* $w_1' \oplus w_2'$ *are defined, where* $w_2' = ((w_2)_\mathrm{L}, (w_1')_\mathrm{S})$*, then* $(w_1 \oplus w_2, w_1' \oplus w_2') \in \widehat{G}_\delta$*.*

Proof. We assume that $(w_1, w_1') \in G_\delta$; for multiple guarantee steps, the result follows from this case by induction. By the definition of state composition, $(w_1)_\mathrm{S} = (w_1 \oplus w_2)_\mathrm{S}$ and $(w_1')_\mathrm{S} = (w_1' \oplus w_2')_\mathrm{S}$. Also, by permission composition, if $((w_1)_\mathrm{L})_\mathrm{P}(r, \gamma, \vec{v}) \in (0, 1]$ then $((w_1 \oplus w_2)_\mathrm{L})_\mathrm{P}(r, \gamma, \vec{v}) \in (0, 1]$. For region creation, it follows that the new region cannot be in the domain of $(w_1)_\mathrm{S}$. For the region removal, $((w_1)_\mathrm{L})_\mathrm{P}$ must contain all the permissions on the region being removed, hence so will $((w_1 \oplus w_2)_\mathrm{L})_\mathrm{P}$. The rest follows immediately by the definition of guarantee. □

The following two lemmas are direct corollaries of Lemma B.4.

Lemma B.5 (Guarantee locality II)**.** *If $(w_1, w'_1) \in \widehat{G}_\delta$ and $(w_2, w'_2) \in (R_\delta)^*$ and $w_1 \oplus w_2$ defined and $w'_1 \oplus w'_2$ defined, then $(w_1 \oplus w_2, w'_1 \oplus w'_2) \in \widehat{G}_\delta$.*

Lemma B.6 (Guarantee locality III)**.** *If $(w_1, w'_1) \in \widehat{G}_\delta$ and $w_1 \oplus w_2$ is defined and $\lfloor w_1 \rfloor_{\mathrm{H}} = \lfloor w'_1 \rfloor_{\mathrm{H}}$, then $(w_1 \oplus w_2, w'_1 \oplus w'_2) \in \widehat{G}_\delta$, where $w'_2 = ((w_2)_{\mathrm{L}}, (w'_1)_{\mathrm{S}})$.*

The following rely locality lemma allows us to decompose a transition that is permitted by the rely on a composite state into transitions permitted by the rely on the component states.

Lemma B.7 (Rely locality)**.** *If $w = w_1 \oplus w_2$ and $(w, w') \in (R_\delta)^*$ then $\exists w'_1, w'_2$ such that $(w_1, w'_1) \in (R_\delta)^*$ and $(w_2, w'_2) \in (R_\delta)^*$ and $w' = w'_1 \oplus w'_2$.*

Proof. We assume that $(w, w') \in R_\delta$; for multiple rely steps, the result follows from this case by induction. Take $w'_1 = ((w_1)_{\mathrm{L}}, (w')_{\mathrm{S}})$ and $w'_2 = ((w_2)_{\mathrm{L}}, (w')_{\mathrm{S}})$. By the definition of state composition, $(w_1)_{\mathrm{S}} = (w_2)_{\mathrm{S}} = w_{\mathrm{S}}$. A rely step does not affect the local state, so $w' = w'_1 \oplus w'_2$. Furthermore, w_1 and w_2 have no more permissions than w and so any change to the shared state allowed by the rely on w is also allowed by the rely on w_1 and w_2. Hence, $(w_1, w'_1) \in (R_\delta)^*$ and $(w_2, w'_2) \in (R_\delta)^*$. $\square$

The following two lemmas deal with the consistency of rely with guarantee: an action permitted for one thread by the guarantee corresponds to interference for another thread that is consistent with the rely.

Lemma B.8 (Containment of rely in guarantee I)**.** *If $w_1 \oplus w_2$ is defined and $(w_1, w'_1) \in G_\delta$ and $\lfloor w'_1 \rfloor_{\mathrm{H}} \oplus ((w_2)_{\mathrm{L}})_{\mathrm{H}}$ is defined then $w'_2 = ((w_2)_{\mathrm{L}}, (w'_1)_{\mathrm{S}})$ is well-formed and $(w_2, w'_2) \in R_\delta$ and $w'_1 \oplus w'_2$ is defined.*

Proof. By the definition of state composition $(w_1)_{\mathrm{S}} = (w_1 \oplus w_2)_{\mathrm{S}}$. By the definition of permission composition, if $((w_1)_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \vec{v}) \in (0, 1]$ and $w_1 \oplus w_2$ is defined, then $((w_2)_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \vec{v}) \in [0, 1)$. We define w'_2 as $w'_2 = ((w_2)_{\mathrm{L}}, (w'_1)_{\mathrm{S}})$. For region creation this is trivial. For region removal, we know $((w_1)_{\mathrm{L}})_{\mathrm{P}}$ must contain all the permissions for the removed region, therefore $((w_2)_{\mathrm{L}})_{\mathrm{P}}$ cannot contain any permissions on the region, hence it can be removed. The rest holds immediately by the definition of rely. $\square$

Remark B.9. If $w_1 \oplus w_2$ is defined and $\lfloor w_1 \rfloor_{\mathrm{H}} = \lfloor w'_1 \rfloor_{\mathrm{H}}$, then $\lfloor w'_1 \rfloor_{\mathrm{H}} \oplus ((w_2)_{\mathrm{L}})_{\mathrm{H}}$ is defined.

Lemma B.10 (Containment of rely in guarantee II)**.** *If $w_1 \oplus w_2$ is defined and $(w_1, w'_1) \in \widehat{G}_\delta$ and $\lfloor w'_1 \rfloor_{\mathrm{H}} \oplus ((w_2)_{\mathrm{L}})_{\mathrm{H}}$ is defined then $w'_2 = ((w_2)_{\mathrm{L}}, (w'_1)_{\mathrm{S}})$ is well-formed and $(w_2, w'_2) \in (R_\delta)^*$ and $w'_1 \oplus w'_2$ is defined.*

Proof. This follows from Lemma B.8 by induction on the number of $\overline{G}_\delta$ steps, making use of Remark B.9. $\square$

Note that the above would not hold for $(G_\delta)^*$ instead of $\widehat{G}_\delta$. To understand why this is so, suppose that there are two actions for some shared region, one of which involves putting, say, the heap cell $5 \mapsto 3$ into the region and making some other update and the other involves taking the cell $5 \mapsto 3$ back out of the shared region. $(G_\delta)^*$ would allow both actions to happen in a single step without the thread performing the action to have ever had the cell $5 \mapsto 3$ – it is created and destroyed in a virtual sense. This is an issue if another thread already holds $5 \mapsto 3$, since it would assume that, since the intermediate state is inconsistent, the actions could not be performed. $\widehat{G}_\delta$ is immune to this since it only permits a single concrete update, and thereby disallows such virtual state.

The following abstract state locality lemma captures the intuition that the operational semantics is local at the level of logical state.

Lemma B.11 (Abstract state locality). *If $(C, \lfloor w_1 \oplus w_2 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (C', h)$ and $C, w_1, \eta, \delta, i, Q$ safe$_{n+1}$, then $\exists w_1', w_2'$ such that $(C, \lfloor w_1 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (C', \lfloor w_1' \rfloor_{\mathrm{H}})$ and $h = \lfloor w_1' \oplus w_2' \rfloor_{\mathrm{H}}$ and $(w_1, w_1') \in \widehat{G}_\delta$ and $(w_2, w_2') \in (R_\delta)^*$ and $C', w_1', \eta, \delta, i, Q$ safe$_n$.*

Proof. Let $h' = ((w_2)_{\mathrm{L}})_{\mathrm{H}}$. By the definition of composition $\lfloor w_1 \oplus w_2 \rfloor_{\mathrm{H}} = \lfloor w_1 \rfloor_{\mathrm{H}} \oplus h'$.

By appeal to Lemma B.2 (Concrete single-step locality) there exists an h'' such that $C, \lfloor w_1 \rfloor_{\mathrm{H}} \xrightarrow{\eta} C', h''$ and $h'' \oplus h' = h$. By the definition of safety there must exist a w_1' such that $\lfloor w_1' \rfloor_{\mathrm{H}} = h''$ and $(w_1, w_1') \in \widehat{G}_\delta$ and $C', w_1', \eta, \delta, i, Q$ safe$_n$.

We now define w_2' as $((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}})$ – that is, the local state from w_2 and shared state from w_1'. Since $h = h'' \oplus h' = \lfloor w_1' \rfloor_{\mathrm{H}} \oplus ((w_2')_{\mathrm{L}})_{\mathrm{H}}$ and no permissions in common with $((w_2')_{\mathrm{L}})_{\mathrm{H}}$ are created or destroyed by any guarantee step, it must be that $w_1' \oplus w_2'$ is defined and that $h = \lfloor w_1 \oplus w_2 \rfloor_{\mathrm{H}}$. It remains to prove that $(w_2, w_2') \in (R_\delta)^*$. This follows immediately by appeal to Lemma B.10. □

We now have the building blocks necessary to justify the parallel composition rule. The following lemma establishes the safety conditions necessary for the soundess of the Par rule.

Lemma B.12 (Parallel safety decomposition). *If $w = w_1 \oplus w_2$, and $C_1, w_1, \eta, \delta, i, Q_1$ safe$_n$ and $C_2, w_2, \eta, \delta, i, Q_2$ safe$_n$, and $\mathsf{stable}_\delta(Q_1)$ and $\mathsf{stable}_\delta(Q_2)$ then $C_1 \| C_2, w, \eta, \delta, i, Q_1 * Q_2$ safe$_n$.*

Proof. By induction. The zero case holds trivially. In the inductive case we must prove $C_1 \| C_2, w, \eta, \delta, i, Q_1 * Q_2$ safe$_{n+1}$. We break down the definition of safety as follows (the fourth clause is satisfied trivially as the program is not **skip**):

1. $(w, w') \in (R_\delta)^* \implies C_1 \| C_2, w', \eta, \delta, i, Q_1 * Q_2$ safe$_n$
2. $\neg((C_1 \| C_2, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} \mathit{fault})$
3. $(C_1 \| C_2, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (C', h') \implies \exists w'.\, \lfloor w' \rfloor_{\mathrm{H}} = h' \wedge (w, w') \in \widehat{G}_\delta$ $\wedge\, C', w', \eta, \delta, i, Q_1 * Q_2$ safe$_n$

For the first clause, we assume that $(w, w') \in R_\delta$. By Lemma B.7 (Rely locality) $\exists w_1', w_2'$ such that $(w_1, w_1') \in R_\delta$ and $(w_2, w_2') \in R_\delta$ and $w' = w_1' \oplus w_2'$. Hence, with the assumptions we get $C_1, w_1', \eta, \delta, i, Q_1$ safe$_n$ and $C_2, w_2', \eta, \delta, i, Q_2$ safe$_n$. Hence, by the inductive assumption this proves the clause.

For the second clause, we observe that if the thread faults in a parallel composition, then either C_1 or C_2 must have faulted. By Lemma B.3 (Concrete fault locality) the threads must also fault in $\lfloor w_1 \rfloor_{\mathrm{H}}$ or $\lfloor w_2 \rfloor_{\mathrm{H}}$ respectively. But by the safety assumptions for C_1 and C_2 this cannot occur.

For the third clause, there are two cases: either $C_1 \| C_2 = \mathbf{skip} \| \mathbf{skip}$, or one of C_1 or C_2 is reduced. We first consider the **skip** case. In this case, $C' = \mathbf{skip}$ and $h' = \lfloor w \rfloor_{\mathrm{H}}$ by the semantics. If $n = 0$ then the clause holds trivially by letting $w' = w$.

Assume that $n > 0$. By the safety of C_1, there is a w_1' with $\lfloor w_1' \rfloor_{\mathrm{H}} = \lfloor w_1 \rfloor_{\mathrm{H}}$ and $(w_1, w_1') \in \widehat{G}_\delta$ and $w_1' \in [\![Q_1]\!]_{\delta,i}$. By Lemma B.10 and Remark B.9, $(w_2, w_2') \in (R_\delta)^*$ and $w_1' \oplus w_2'$ is defined, for $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{S}})$. By the safety of C_2, it follows that $\mathbf{skip}, w_2', \eta, \delta, i, Q_2$ safe$_n$. Since $n > 0$, this implies that there is a w_2'' with $\lfloor w_2' \rfloor_{\mathrm{H}} = \lfloor w_2'' \rfloor_{\mathrm{H}}$ and $(w_2', w_2'') \in \widehat{G}_\delta$ and $w_2'' \in [\![Q_2]\!]_{\delta,i}$. By Lemma B.10 and Remark B.9, $(w_1', w_1'') \in (R_\delta)^*$ and $w_1'' \oplus w_2''$ is defined, for $w_1'' = ((w_1')_{\mathrm{L}}, (w_2'')_{\mathrm{S}})$. Let $w' = w_1'' \oplus w_2''$. Now, $\lfloor w_1'' \oplus w_2'' \rfloor_{\mathrm{H}} = ((w_1'')_{\mathrm{L}})_{\mathrm{H}} \oplus \lfloor w_2'' \rfloor_{\mathrm{H}} = ((w_1')_{\mathrm{L}})_{\mathrm{H}} \oplus \lfloor w_2' \rfloor_{\mathrm{H}} = \lfloor w_1' \oplus w_2' \rfloor_{\mathrm{H}} = \lfloor w_1' \rfloor_{\mathrm{H}} \oplus ((w_2')_{\mathrm{L}})_{\mathrm{H}} = \lfloor w_1 \rfloor_{\mathrm{H}} \oplus ((w_2)_{\mathrm{L}})_{\mathrm{H}} = \lfloor w_1 \oplus w_2 \rfloor_{\mathrm{H}}$, as required. By Lemma B.6, $(w_1 \oplus w_2, w_1' \oplus w_2') \in \widehat{G}_\delta$ and $(w_1' \oplus w_2', w_1'' \oplus w_2'') \in \widehat{G}_\delta$. Since the concrete heap is unchanged in both of these steps, $(w, w') \in \widehat{G}_\delta$, as required. Since $\mathsf{stable}_\delta(Q_1)$ and $\mathsf{stable}_\delta(Q_2)$, it follows that $\mathsf{stable}_\delta(Q_1 * Q_2)$. By stability of Q_1, $w_1'' \in [\![Q_1]\!]_{\delta,i}$. Thus, since we know that $w_2'' \in [\![Q_2]\!]_{\delta,i}$, $w' \in [\![Q_1 * Q_2]\!]_{\delta,i}$. By appeal to Lemma B.1 (Skip safety), we have the final requirement, that $C', w', \eta, \delta, i, Q_1 * Q_2$ safe$_n$.

Now consider the case where C_1 or C_2 is reduced. We only consider C_1; the C_2 case is identical. It must hold that $(C_1, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (C_1', h')$ and $C' = C_1' \| C_2$. By appeal to Lemma B.11 (Abstract state locality) there are w_1', w_2' such that $(C_1, \lfloor w_1 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (C_1', \lfloor w_1' \rfloor_{\mathrm{H}})$ and $h' = \lfloor w_1' \oplus w_2' \rfloor_{\mathrm{H}}$ and $(w_1, w_1') \in \widehat{G}_\delta$ and $(w_2, w_2') \in (R_\delta)^*$ and $C_1', w_1', \eta, \delta, i, Q_1$ safe$_n$. Let $w' = w_1' \oplus w_2'$. We have that $\lfloor w' \rfloor_{\mathrm{H}} = h'$ as required. By Lemma B.5 (Guarantee locality II), we also have $(w, w') \in \widehat{G}_\delta$, as required. By the safety of C_2, we have $C_2, w_2', \eta, \delta, i, Q_2$ safe$_n$. By the inductive assumption, we the final requirement, that $C', w', \eta, \delta, i, Q_1 * Q_2$ safe$_n$. □

The following two lemmas justify the Frame rule.

Lemma B.13. *If $C, w_1, \eta, \delta, i, Q$ safe$_n$ and $w = w_1 \oplus w_2$ and $w_2 \in [\![F]\!]_{\delta,i}$ and $\mathsf{stable}_\delta(F)$, then $C, w, \eta, \delta, i, Q * F$ safe$_n$.*

Proof. By induction on n. The zero case is trivial. In the inductive case we must prove $C, w, \eta, \delta, i, Q * F$ safe$_{n+1}$, which we break down as follows

1. $(w, w') \in (R_\delta)^* \implies C, w', \eta, \delta, i, Q * F$ safe$_n$
2. $\neg((C, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} \mathit{fault})$
3. $(C, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (C', h') \implies \exists w'.\, \lfloor w' \rfloor_{\mathrm{H}} = h' \wedge (w, w') \in \widehat{G}_\delta$
   $\wedge\, C', w', \eta, \delta, i, Q * F$ safe$_n$
4. $C = \mathbf{skip} \implies \exists w'.\, (w, w') \in \widehat{G}_\delta \wedge \lfloor w \rfloor_{\mathrm{H}} = \lfloor w' \rfloor_{\mathrm{H}} \wedge w' \in [\![Q * F]\!]_{\delta,i}$

For the first clause, we can assume $(w, w') \in (R_\delta)^*$ which with Lemma B.7 (Rely locality) gives us w_1', w_2' such that $(w_1, w_1') \in (R_\delta)^*$ and $(w_2, w_2') \in (R_\delta)^*$ and $w' = w_1' \oplus w_2'$. By the safety assumption, we have $C, w_1', \eta, \delta, i, Q$ safe$_n$. As F is stable, we know $w_2' \in [\![F]\!]_{\delta,i}$. By the inductive assumption, we have $C, w', \eta, \delta, i, Q * F$ safe$_n$, which completes the case.

The second clause holds by the contrapositive of Lemma B.3 (Concrete fault locality).

For the third clause, we assume a transition $C, \lfloor w \rfloor_{\mathrm{H}} \xrightarrow{\eta} C', h'$. By appeal to Lemma B.11 (Abstract state locality) there exist w_1', w_2' such that $(C, \lfloor w_1 \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (C', \lfloor w_1' \rfloor_{\mathrm{H}})$ and $h' = \lfloor w_1' \oplus w_2' \rfloor_{\mathrm{H}}$ and $(w_1, w_1') \in \widehat{G}_\delta$ and $(w_2, w_2') \in (R_\delta)^*$ and $C', w_1', \eta, \delta, i, Q$ safe$_n$. Let $w' = w_1' \oplus w_2'$. We have that $\lfloor w' \rfloor_{\mathrm{H}} = h'$, as required. By Lemma B.5, we have $(w, w') \in \widehat{G}_\delta$, as required. By the stability of F we have that $w_2' \in \llbracket F \rrbracket_{\delta,i}$. Hence, by the inductive assumption, we have the final requirement, that $C', w', \eta, \delta, i, Q * F$ safe$_n$.

For the fourth clause assume that $C = \mathbf{skip}$. By the safety assumption, we know there exists w_1' such that $(w_1, w_1') \in \widehat{G}_\delta$ and $\lfloor w_1 \rfloor_{\mathrm{H}} = \lfloor w_1' \rfloor_{\mathrm{H}}$ and $w_1' \in \llbracket Q \rrbracket_{\delta,i}$. By Lemma B.6 (Guarantee locality III), for $w_2' = ((w_2)_{\mathrm{L}}, (w_1')_{\mathrm{H}})$ we have $(w_1 \oplus w_2, w_1' \oplus w_2') \in \widehat{G}_\delta$. Let $w' = w_1' \oplus w_2'$. We have that $(w, w') \in \widehat{G}$, as required. Also, $\lfloor w \rfloor_{\mathrm{H}} = \lfloor w_1 \rfloor_{\mathrm{H}} \oplus ((w_2)_{\mathrm{L}})_{\mathrm{H}} = \lfloor w_1' \rfloor_{\mathrm{H}} \oplus ((w_2')_{\mathrm{L}})_{\mathrm{H}} = \lfloor w' \rfloor_{\mathrm{H}}$, as required. By Lemma B.10 (Containment of rely in guarantee II), we have $(w_2, w_2') \in (R_\delta)^*$. By the stability of F, it follows that $w_2 \in \llbracket F \rrbracket_{\delta,i}$. Thus, since also $w_1' \in \llbracket Q \rrbracket_{\delta,i}$, we have the final requirement, that $w' \in \llbracket Q * F \rrbracket_{\delta,i}$. □

Lemma B.14 (Frame safety)**.** *If $\delta, \eta \models_n \{P\}\, C\, \{Q\}$ and $\mathsf{stable}_\delta(F)$, then $\delta, \eta \models_n \{P * F\}\, C\, \{Q * F\}$.*

Proof. We pick w, i such that $w \in \llbracket P * F \rrbracket_{\delta,i}$. Note that by the definition of $*$, $\exists w_1, w_2$ such that $w_1 \in \llbracket P \rrbracket_{\delta,i}$, $w_2 \in \llbracket F \rrbracket_{\delta,i}$, and $w = w_1 \oplus w_2$. Thus the result follows directly from Lemma B.13. □

For our modified rule of consequence, we have to establish that repartitioning the state in accordance with the guarantee is permissible both before and after the execution of a command. The following two lemmas capture this.

Lemma B.15 (Pre-partitioning safety)**.** *If $P \Longrightarrow_\delta P'$ and $\delta, \eta \models_n \{P'\}C\{Q\}$ and $\mathsf{stable}_\delta(P)$, then $\delta, \eta \models_n \{P\}C\{Q\}$.*

Proof. Induction on n. The zero case is trivial. For the inductive case, we must establish $\delta, \eta \models_{n+1} \{P\}C\{Q\}$.

Fix i and $w \in \llbracket P \rrbracket_{\delta,i}$. By our first assumption, there exists a w' with $w' \in \llbracket P' \rrbracket_{\delta,i}$ and $\lfloor w \rfloor_{\mathrm{H}} = \lfloor w' \rfloor_{\mathrm{H}}$ and $(w, w') \in \widehat{G}_\delta$. By our second assumption, $C, w', \eta, \delta, i, Q,$ safe$_{n+1}$.

We wish to show C, w, η, δ, i, Q safe$_{n+1}$, which we break down to

1. $(w, w'') \in (R_\delta)^* \implies C, w', \eta, \delta, i, Q$ safe$_n$
2. $\neg((C, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} \textit{fault})$
3. $(C, \lfloor w \rfloor_{\mathrm{H}}) \xrightarrow{\eta} (C', h') \implies \exists w''.\, \lfloor w'' \rfloor_{\mathrm{H}} = h' \wedge (w, w'') \in \widehat{G}_\delta$
   $\wedge\, C', w'', \eta, \delta, i, Q$ safe$_n$
4. $C = \mathbf{skip} \implies \exists w''.\, (w, w'') \in (G_\delta)^* \wedge \lfloor w \rfloor_{\mathrm{H}} = \lfloor w'' \rfloor_{\mathrm{H}} \wedge w'', i \models_\delta Q$

The first clause holds by the inductive assumption since P is stable. The remaining clauses hold by the fact that w' and w do not differ in concrete state, and the fact that C is safe of w'. Since the choice of i and w was arbitrary, $\delta, \eta \models_{n+1} \{P\}C\{Q\}$, as required. □

Lemma B.16 (Post-partitioning safety)**.** *If $Q' \Longrightarrow_\delta Q$ and $\delta, \eta \models_n \{P\}C\{Q'\}$, then $\delta, \eta \models_n \{P\}C\{Q\}$.*

Proof. Induction on n. The zero case is trivial. For the inductive case, we must establish $\delta, \eta \models_{n+1} \{P\}C\{Q\}$.

Fix i and $w \in [\![P]\!]_{\delta,i}$. By assumption, $C, w, \eta, \delta, i, Q'$ safe$_{n+1}$. We must show that C, w, η, δ, i, Q safe$_{n+1}$. This follows trivially for the first three clauses of the definition of safety. For the fourth, we can assume that

$$\exists w'. (w, w') \in \widehat{G}_\delta \wedge \lfloor w \rfloor_{\mathrm{H}} = \lfloor w' \rfloor_{\mathrm{H}} \wedge w', i \models_\delta Q'$$

and must show

$$\exists w'. (w, w') \in \widehat{G}_\delta \wedge \lfloor w \rfloor_{\mathrm{H}} = \lfloor w' \rfloor_{\mathrm{H}} \wedge w', i \models_\delta Q$$

which follows by the definition of $\Longrightarrow$ and $\widehat{G}_\delta$. □

The next lemma establishes the soundness of the Atomic rule.

Lemma B.17 (Atomic safety)**.** *If* $\vdash_{\mathsf{SL}} \{p\}C\{q\}$ *and* $P \overset{\{p\}\{q\}}{\Longrightarrow}_\delta Q$*, and* $\mathsf{stable}_\delta(P)$ *and* $\mathsf{stable}_\delta(Q)$*, then* $\delta; \eta \models_n \{P\} \langle\, C \,\rangle \{Q\}$

Proof. By induction on n. The zero case is trivial. For the inductive case we assume a w and i such that $w \in [\![P]\!]_{\delta,i}$. Note that, by the definition of $\Longrightarrow$ there are h_1, h'' such that $\lfloor w \rfloor_{\mathrm{H}} = h_1 \oplus h''$ and $h_1 \in [\![p]\!]_i$ and for all $h_2 \in [\![q]\!]_i$ there exists $w' \in [\![Q]\!]_{\delta,i}$ such that $h_2 \oplus h'' = \lfloor w' \rfloor_{\mathrm{H}}$ and $(w, w') \in \widehat{G}_\delta$.

We now need to prove that $\langle C \rangle, w, \eta, \delta, i, Q$ safe$_{n+1}$, which we break down as follows (the fourth clause is trivial as the command is not **skip**)

1. $(w, w') \in (R_\delta)^* \implies \langle C \rangle, w', \eta, \delta, i, Q$ safe$_n$
2. $\neg(\langle C \rangle, \lfloor w \rfloor_{\mathrm{H}} \xrightarrow{\eta} \mathit{fault})$
3. $\langle C \rangle, \lfloor w \rfloor_{\mathrm{H}} \xrightarrow{\eta} C', h' \implies \exists w'. \lfloor w' \rfloor_{\mathrm{H}} = h' \wedge (w, w') \in \widehat{G}_\delta$
   $\wedge\, C', w', \eta, \delta, i, Q$ safe$_n$

The first clause holds by the stability of P and the inductive assumption.

Suppose for a contradiction that $\langle C \rangle, \lfloor w \rfloor_{\mathrm{H}} \xrightarrow{\eta} \mathit{fault}$. By the semantics, this means that $C, \lfloor w \rfloor_{\mathrm{H}} \xrightarrow{\eta} \mathit{fault}$. By Lemma B.3 (Concrete fault locality) this means that $C, h_1 \xrightarrow{\eta} \mathit{fault}$, which cannot be by separation logic soundness.

For clause 3, we assume there is some C', h' with $\langle C \rangle, \lfloor w \rfloor_{\mathrm{H}} \xrightarrow{\eta} C', h'$. By the operational semantics, $C' = \mathbf{skip}$ and $C, \lfloor w \rfloor_{\mathrm{H}} \xrightarrow{\eta}{}^{*} \mathbf{skip}, h'$. By the soundness of separation logic, we have $\neg(C, h_1 \xrightarrow{\eta}{}^{*} \mathit{fault})$. By Lemma B.2 (Concrete locality), there is an h_2 with $C, h_1 \xrightarrow{\eta}{}^{*} \mathbf{skip}, h_2$ and $h' = h_2 \oplus h''$. Again, by the soundness of separation logic, we have $h_2 \in [\![q]\!]_i$. Therefore, there exists w' with $h_2 \oplus h'' = \lfloor w' \rfloor_{\mathrm{H}}$ and $(w, w') \in \widehat{G}$ and $w' \in [\![Q]\!]_{\delta,i}$. For this w', the first two conditions hold by the above, and the third by Lemma B.1 (Skip safety). □

Finally, the following lemma establishes the soundness of the Pred-E rule. The result is quite simple, since the predicate being eliminated has no bearing on the semantical validity of the assertion.

Lemma B.18 (Predicate elimination)**.** *If* $\Delta, \forall \overline{x}.\ \alpha(\overline{x}) \equiv R; \Gamma \models \{P\}C\{Q\}$ *and* $\Delta \vdash \mathsf{stable}(R)$ *and* $\alpha \notin \Gamma, \Delta, P, Q$*, then* $\Delta; \Gamma \models \{P\}C\{Q\}$.

Proof. We first claim that:

$$\begin{aligned}\delta' \in [\![\Delta]\!] \iff & \exists \delta.\, \delta \in [\![\Delta, \forall \overline{x}.\ \alpha(\overline{x}) \equiv R]\!] \land \\ & \forall \vec{v}.\, \delta'[(\alpha, \vec{v}) \mapsto \bot] = \delta[(\alpha, \vec{v}) \mapsto \bot]\end{aligned}$$

This can be easily proved by appeal to the predicate definition semantics. Note that stability is required otherwise the set of δs is empty.

We then make use of this result to prove that for any n, i, and $\delta' \in [\![\Delta]\!]$, there exists a $\delta.\, \delta \in [\![\Delta \land \forall \overline{x}.\ \alpha(\overline{x}) \Leftrightarrow R]\!]$ such that

$$[\![\Gamma]\!]_{n,\delta',i} \quad = \quad [\![\Gamma]\!]_{n,\delta,i}$$

and for any assertion P, it holds that

$$w \in [\![P]\!]_{\delta',i} \quad \iff \quad w \in [\![P]\!]_{\delta,i}$$

Both results can be proved by simple appeal to the semantics. These results are sufficient to prove our main result, as other values are unaffected by the selection of δ. □

We now bring together these lemmas in the following theorem, which establishes the soundess of our proof system.

Theorem B.19 (Soundness)**.** *If* $\Delta; \Gamma \vdash \{P\}C\{Q\}$*, then* $\Delta; \Gamma \models \{P\}C\{Q\}$*.*

Proof. Proved by induction over proof rules. We consider in detail the Skip, Par, Frame, Atomic, Conseq and Pred-E rules. Other rules follow trivially by the inductive assumption.

$$\frac{}{\Delta; \Gamma \models \{P\}\ \mathbf{skip}\ \{P\}}\ (\textsc{Skip})$$

Holds by Lemma B.1 (Skip safety).

$$\frac{\Delta; \Gamma \models \{P_1\}\ C_1\ \{Q_1\} \quad \Delta; \Gamma \models \{P_2\}\ C_2\ \{Q_2\}}{\Delta; \Gamma \models \{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\ (\textsc{Par})$$

Holds by Lemma B.12 (Parallel safety decomposition).

$$\frac{\Delta; \Gamma \models \{P\}\ C\ \{Q\} \quad \mathsf{stable}(F)}{\Delta; \Gamma \models \{P * F\}\ C\ \{Q * F\}}\ (\textsc{Frame})$$

Holds by Lemma B.14 (Frame safety).

$$\frac{\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\} \quad \Delta \vdash P \Longrightarrow_{\delta}^{\{p\}\{q\}} Q}{\Delta; \Gamma \models \{P\}\ \langle C \rangle\ \{Q\}}\ (\textsc{Atomic})$$

Holds by Lemma B.17 (Atomic safety).

$$\frac{\Delta \vdash P \Rrightarrow P' \qquad \Delta; \Gamma \models \{P'\}\ C\ \{Q'\} \qquad \Delta \vdash Q' \Rrightarrow Q}{\Delta; \Gamma \models \{P\}\ C\ \{Q\}}\ (\textsc{Conseq})$$

For simplicity we treat the P and Q implications separately - no loss of generality results. The P case follows from Lemma B.15 (Pre-partitioning safety), while the Q case follows from Lemma B.16 (Post-partitioning safety).

$$\frac{\begin{array}{c}\mathsf{stable}(R) \qquad \alpha \notin \Gamma, \Delta, P, Q \\ \Delta, \forall \overline{x}.\ \alpha(\overline{x}) \equiv R; \Gamma \models \{P\}C\{Q\}\end{array}}{\Delta; \Gamma \models \{P\}C\{Q\}}\ (\textsc{Pred-E})$$

Holds by appeal to Lemma B.18 (Predicate elimination).

□