Concurrent Abstract Predicates

– long version –

(Draft, submitted for publication)

Thomas Dinsdale-Young[1], Mike Dodds[2], Philippa Gardner[1], Matthew Parkinson[2], and Viktor Vafeiadis[2]

[1] Imperial College, London
[2] University of Cambridge

Abstract. Abstraction is key to understanding and reasoning about large computer systems. Abstraction is easy to achieve if the relevant data structures are disjoint, but rather difficult when they are partially shared, as is the case for concurrent modules. We present a program logic for reasoning abstractly about data structures that gives us 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 large physical systems, we use both abstraction and locality to hide irrelevant aspects of the system. For example, when building a house in London, we do not consider the gravitational forces exerted by the 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, while abstraction gives us a structured view of the system, because components can be used purely in terms of their abstract properties.

Locality can be often used directly to achieve a measure of abstraction. Using separation logic [18] we can prove that a module runs within a particular structure, disjointly from other modules. If the structure is manipulated only by module functions, it can be represented just in terms of its abstract properties, by an abstract predicate [23]. For example, we can implement a set by a linked list. This set can be represented by an abstract predicate asserting that, for example,"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 that, after deleting 6, that "the set is $\{5\}$".

Often, however, fine-grained reasoning at the abstract level cannot be captured by traditional abstract predicates. It can be captured by the abstract reasoning introduced by context logic [4, 5], which reasons at the same level of abstraction as the module. With our set-module example, the client can reason about a set in terms of its individual elements, manipulating the element 6 separately from the rest of the set. This fine-grained abstraction is not supported using traditional abstract predicates, because locality in the abstraction need not correspond to locality in the implementation [8]. If the set module is implemented as a list, then to manipulate individual elements of the set, the implementation traverses the global list structure. Therefore, individual elements are not represented disjointly in the implementation, and fine-grained reasoning is not possible using simple abstract predicates.

For concurrency, fine-grained reasoning is essential for compositional reasoning about modules shared between threads. For example, we may wish to manipulate individual elements of the set in separate threads, whereas, in the implementation, threads may access a shared list structure simultaneously. This sharing breaks the abstract view of disjointess between the elements. Even the humble lock breaks disjointness, since acquiring the lock necessarily involves a race to decide who acquires the lock. Existing systems for concurrent reasoning either assume locks as primitive or do not provide abstraction for locks.

We present a program logic which allows abstraction in the presence of sharing, introducing *concurrent abstract predicates*. These predicates present a fiction of disjointness [8]; 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 as 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 of these predicates assert properties about the same shared structure, and both can be used at the same time by separate concurrent 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 [9], 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 is shared.

With our program logic, a module implementation can be verified using a low-level specification. This low-level specification can be abstracted to give a high-level specification expressed using concurrent abstract predicates. Clients of the module can then be verified purely in terms of this abstract specification, without reference to the module's implementation. We demonstrate this methodology by providing two implementations of a lock library satisfying the same abstract lock specification, and then 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. We therefore demonstrate that concurrent abstract predicates provide the necessary abstraction for compositional reasoning about concurrent systems.

In this paper, we use a very simple low-level language with pointers and functions for exposition. The work naturally extends to object-oriented languages using Parkinson and Bierman's logic for Java [23]. The object-oriented features are orthogonal to concurrency abstraction we present in this paper.

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 lock implementations satisfying the specification.

2.1 Lock Specification

A typical lock module has the functions lock(x) and unlock(x), for acquiring and releasing a lock respectively. It also has a mechanism for constructing locks, such as makelock(n), which allocates a lock followed by a contiguous block of memory of size n. We specify these functions using the following local Hoare triples:

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

This abstract specification, which is independent of the underlying implementation, is presented by the module to the client.[3] The assertions $\mathsf{isLock}(x)$ and $\mathsf{Locked}(x)$ are abstract predicates. $\mathsf{isLock}(x)$ asserts that the lock can be acquired by the thread, while $\mathsf{Locked}(x)$ asserts that the thread holds the lock. Below we give a concrete interpretation of these predicates for a simple compare-and-swap lock. The logical connective $*$ is the separating conjunction from separation logic: an assertion $p * q$ asserts that the state can be split disjointly into two parts, one satisfying p and the other satisfying q.

The abstract predicates satisfy the following axioms, which the module also presents to the client:

$$\begin{aligned} \mathsf{isLock}(x) &\iff \mathsf{isLock}(x) * \mathsf{isLock}(x) \\ \mathsf{Locked}(x) * \mathsf{Locked}(x) &\iff \mathsf{false} \end{aligned}$$

The first axiom allows the client to share freely the knowledge that x is a lock that can be locked.[4] The second axiom implies 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 $\mathrm{lock}(x)$ is called twice in succession, then the program will not terminate as the post-condition is not satisfiable.

2.2 Example: A Compare-and-Swap Lock

Consider a simple compare-and-swap lock implementation:

```
lock(x) {
 local b;
 do
   ⟨b := !CAS(&x, 0, 1)⟩
 while(b)
}
```

```
unlock(x) {
 ⟨[x] := 0⟩
}
```

```
makelock(n) {
 local x := alloc(n+1);
 [x] := 1;
 return x;
}
```

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 not just interpreted 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 internal 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

[3] This specification resembles those used in the work of Gotsman *et al.* [13] and Hobor *et al.* [17] on dynamically-allocated locks. However, our system is much more general, as it can reason about abstract specifications other than locks.

[4] We do not record the splittings of $\mathsf{isLock}(x)$, although we could use permissions [3, 2] to explicitly track this information.

state is indivisible so that all threads maintain a consistent view of it. This is expressed by the logical equivalence $\boxed{P}^r_{I(\vec{x})} * \boxed{Q}^r_{I(\vec{x})} \Leftrightarrow \boxed{P \wedge Q}^r_{I(\vec{x})}$ for shared regions. The possible actions on the state are declared by the environment $I(\vec{x})$.

The permission assertion $[\mathrm{A}]^r_\pi$ specifies that the thread has permission π to perform action A over region r, provided the action is declared in the environment. Following Boyland [3], 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, 1, denoting that the thread can do the action but the environment cannot.[5] We now have the machinery to interpret our lock predicates concretely:

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

The abstract predicate $\mathsf{isLock}(x)$ is interpreted by the concrete, implementation-specific assertion on the right-hand side. This specifies that the local state 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 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 will have 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 change the shared state. They have the form $\mathrm{A}\colon P \rightsquigarrow Q$, where assertion P describes the part of the shared state required to do the action and Q describes the part of the state after the action. The actions for the lock module are

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

The 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 state). The movement of $[\textsc{Unlock}]^r_1$ into local state allows the locking thread to release the lock afterwards. Note that local state is not explicitly represented in the action; since interference only happens on shared state, actions do not need to be prescriptive about local state.

[5] The semantics also contains a zero permission, 0, denoting that the thread may not do the action but the environment may. However, this is not a useful assertion to make, so we forbid it.

The 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 $[\text{UNLOCK}]^r_1$ permission into the shared state. The thread must have $[\text{UNLOCK}]^r_1$ in its local state in order to move it to the shared state as a result of the action.

Notice that UNLOCK is self-referential. The action moves exclusive permission on itself out of local state. Consequently, a thread can only apply 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 will 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:

- LOCK, but this action does not apply, as x has value 1 in the shared state of $\mathsf{Locked}(x)$; and
- UNLOCK, but this action also does not apply, as full permission on it is in the local state of $\mathsf{Locked}(x)$.

The module implementer must show that the concrete interpretation of the predicates satisfies the axioms presented to the client. In our example, the first axiom, that only a single $\mathsf{Locked}(x)$ can exist, follows from the presence in the local state of full permission on UNLOCK. The second, 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.

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 local state. The third assertion specifies that the permission $[\text{UNLOCK}]^r_1$ has moved from the local state to the shared region r as stipulated by the unlock action. This assertion is not, however, stable under intereference 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 postcondition, 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 $[x]$ in the shared state region from 0 to 1. This update is allowed because of the permission $[\text{LOCK}]^r_\pi$ in the local state and the action in $I(r, x)$. The postcondition of the CAS specifies that either location x has value 1 and the unlock permission has moved into the local state as stipulated by the LOCK action, or nothing has happened and the precondition is still satisfied. This postcondition 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 a *repartitioning*

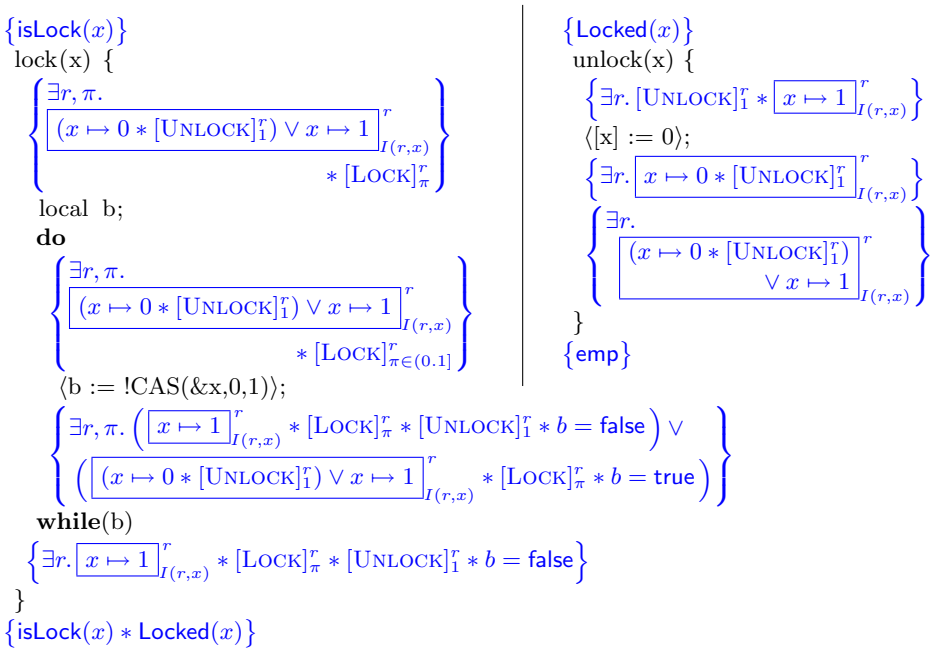


Fig. 1. Verifying the compare-and-swap lock implementation: lock and unlock.

implication $\Longrightarrow$, which enables us to repartition the state between regions and to create regions. In particular, we have the implication

$$P \Longrightarrow \exists r. \boxed{P}^r_{I(\vec{x})} * \mathrm{all}(I(\vec{x}))$$

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

$$x \mapsto 1 \Longrightarrow \exists r. \boxed{x \mapsto 1}^r_{I(r,x)} * [\textsc{Lock}]^r_1 * [\textsc{Unlock}]^r_1$$

The final postcondition results from the definitions of $\mathsf{isLock}(x)$ and $\mathsf{Locked}(x)$, and recalling that $\boxed{x \mapsto 1}^r_{I(r,x)} \Leftrightarrow \boxed{x \mapsto 1}^r_{I(r,x)} * \boxed{x \mapsto 1}^r_{I(r,x)}$.

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, while 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.

$\{\mathsf{emp}\}$
makelock(n) {
  local x := alloc(n + 1);
$\{x \mapsto \_ * (x+1) \mapsto \_ * \ldots * (x+n) \mapsto \_\}$
  [x] := 1;
$\{x \mapsto 1 * (x+1) \mapsto \_ * \ldots * (x+n) \mapsto \_\}$
$\left\{\exists r.\, \boxed{x \mapsto 1}^{r}_{I(r,x)} * [\textsc{Lock}]^{r}_{1} * [\textsc{Unlock}]^{r}_{1} * (x+1) \mapsto \_ * \ldots * (x+n) \mapsto \_\right\}$
  return x;
}
$\{\exists x.\, \mathrm{ret} = x \wedge \mathsf{isLock}(x) * \mathsf{Locked}(x) * (x+1) \mapsto \_ * \ldots * (x+n) \mapsto \_\}$

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

The proof rule for atomic commands is

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

The bodies of atomic commands do not contain other atomic commands, nor do they contain parallel composition. They can thus be specified using separation logic. The first premise, $\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\}$, is therefore a valid triple in sequential separation logic, where p, q denote separation logic assertions that do not specify predicates, shared regions or interference.

The second premise, $\Delta \vdash P \Longrightarrow^{\{p\}\{q\}} Q$, is more complex. It says that the interference allowed by P enables the state to be repartitioned to Q, given the change to memory given by $\{p\}\{q\}$. In our example, when the CAS performs the update the change is $\{x \mapsto 0\}\{x \mapsto 1\}$. We also require that P and Q are *stable*, so that they cannot be falsified by concurrently executing threads. Precondition and postcondition stability is a general requirement that our proof rules have, which for presentation purposes we keep implicit in the rest of this paper.

The implication $P \Longrightarrow Q$ used earlier for constructing a new region is a shorthand for $P \Longrightarrow^{\{\mathsf{emp}\}\{\mathsf{emp}\}} Q$, i.e. a repartitioning where no concrete state changes. We use this implication in the rule of consequence, which we use to move permissions between regions.

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

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 programs [25], interference had to be specified explicitly for each function. Here we can prove a specification for a

module, and then represent the specification abstractly, without mentioning the interference internal to the module.

As we have seen, our predicates can describe the internal interference of a module. Given this, we can define high-level specifications for a module where abstract predicates correspond to invariant assertions about the state of the module (that is, they are 'self-stable'). As these abstract assertions are invariant, we can hide the predicate definitions 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).

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

This rule defines a module consisting of functions $f_1 \ldots f_n$ and uses it to verify a client specification $\{P\}C\{Q\}$.[6] The rule should 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 thepredicate 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 have shown that our lock specification is satisfied by a simple compare-and-swap lock. We now consider a more complex algorithm: a lock that issues tickets to clients contending for the lock. This algorithm is the standard lock algorithm used in current versions of Linux, and 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 our abstract lock specification.

The lock operations are defined as follows:

[6] We could equally have applied this to a class using [23] rather than a set of functions. However, to keep the presentation simple we just use functions.

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

```
unlock(x) {
 ⟨x.owner++;⟩
}
```

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

Here field names are encoded as offsets (.next = 0, .owner = −1).

The implementation assumes an atomic operation INCR that reads a location and increments the stored value. To acquire the lock, a client atomically increments x.next and reads it into 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 holds 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 can increment x.owner.

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

$$T(t,x) \stackrel{\text{def}}{=} \begin{pmatrix} \textsc{Take}\colon\ \exists k.\,([\,\textsc{Next}(k)\,]_1^t * x.\text{next} \mapsto k \quad \rightsquigarrow \quad x.\text{next} \mapsto (k+1)) \\ \textsc{Next}(k)\colon\ x.\text{owner} \mapsto k \quad \rightsquigarrow \quad x.\text{owner} \mapsto (k+1) * [\textsc{Next}(k)]_1^t \end{pmatrix}$$

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 state 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 next field in the Take action.

The concrete interpretation of the predicates is as follows:

$$\mathsf{isLock}(x) \quad \equiv \quad \exists t.\, \boxed{\begin{array}{l}\exists k, k'.\, x.\text{owner} \mapsto k * x.\text{next} \mapsto k' \\ * \, k \le k' * \circledast k'' > k'.\ [\textsc{Next}(k'')]_1^t\end{array}}^{t}_{T(t,x)} * [\textsc{Take}]_\pi^t$$

$$\mathsf{Locked}(x) \quad \equiv \quad \exists t, k.\, \boxed{x.\text{owner} \mapsto k * \mathsf{true}}^{t}_{T(t,x)} * [\textsc{Next}(k)]_1^t$$

($\circledast$ is the lifting of $*$ to sets; it is the separating (multiplicative) version of $\forall$.)

$\mathsf{isLock}(x)$ requires values x.next and x.owner to be in the shared state, and that a permission on $\textsc{Next}(k)$ is in the shared state for each value greater than the current ticket x.next. It also requires a permission on Take to be in local state. $\mathsf{Locked}(x)$ requires just that there is an exclusive permission on $\textsc{Next}(k)$ in local state 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 Take cannot affect the x.owner field. Self-stability for $\mathsf{isLock}(x)$ is ensured by the fact that Take preserves membership of the invariant for region t.

The predicate axioms follow immediately from the predicate definitions, as for the compare-and-swap lock.

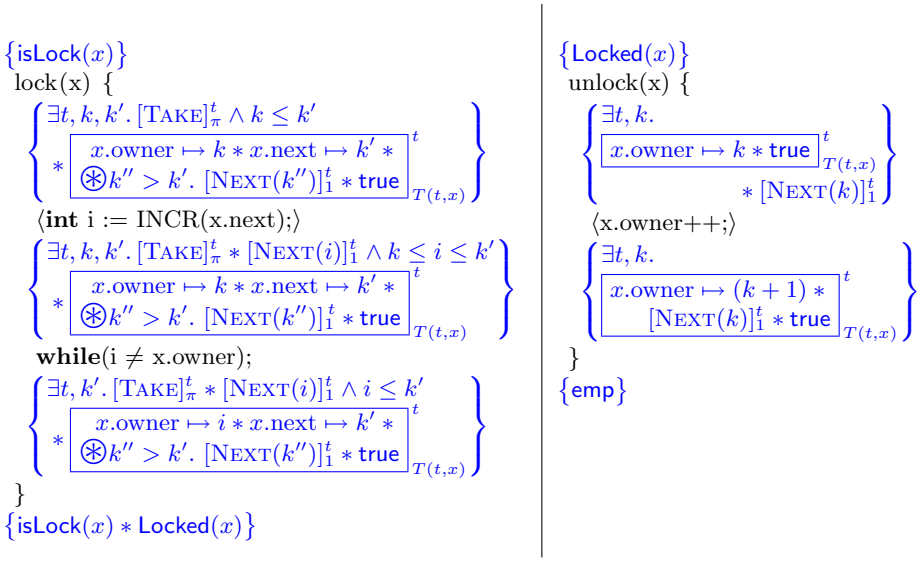


Fig. 3. Proofs for the ticketed lock module operations: lock and unlock.

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 state, and unlock(x) returns it to the shared state. (We omit the proof of makelock, which is similar to the previous 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 concurrent specifications can be used to verify client programs, which may themselves be modules satisfying abstract specifications. We illustrate this by defining a concurrent set module specification and giving two implementations. The set implementations depend for their correctness on the lock specification presented in the previous section.

3.1 A Set Module Specification

A typical set module has three functions: contains(h, v), add(h, v) and remove(h, v). These functions have the following abstract specifications:

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

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

These assertions do not only capture 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. For example, consider the command

$$\text{remove}(h, v_1) \parallel \text{remove}(h, v_2)$$

This command should succeed if it has the permissions to change the values v_1 and v_2 (where $v_1 \neq v_2$), and it should yield a set without v_1 and v_2. This intuition is captured by the proof outline shown in Fig. 4.

$$\begin{array}{c} \{\mathsf{own}(h,v_1) * \mathsf{own}(h,v_2)\} \\ \{\mathsf{own}(h,v_1)\} \ \Big\| \ \{\mathsf{own}(h,v_2)\} \\ \text{remove}(h,v_1) \ \Big\| \ \text{remove}(h,v_2) \\ \{\mathsf{out}(h,v_1)\} \ \Big\| \ \{\mathsf{out}(h,v_2)\} \\ \{\mathsf{out}(h,v_1) * \mathsf{out}(h,v_2)\} \end{array}$$

Fig. 4. Proof outline for set module client.

3.2 Example: The Coarse-grained Set

Consider a coarse-grained set implementation, based on the lock module and the sequential set operations scontains(y, v), sadd(y, v) and sremove(y, v).

```
contains(h,v) {
  lock(h.lock);
  scontains(h.set,v);
  unlock(h.lock);
}
```

```
add(h,v) {
  lock(h.lock);
  sadd(h.set,v);
  unlock(h.lock);
}
```

```
remove(h,v) {
  lock(h.lock);
  sremove(h.set,v);
  unlock(h.lock);
}
```

Interpretation of Abstract Predicates. We assume a sequential set predicate $\mathsf{Set}(y, xs)$ that asserts that the sequential 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. The sequential set operations have the following specifications with respect to Set:

$$\begin{array}{lll}
\{\mathsf{Set}(y, vs)\} & \text{scontains}(y, v) & \{\mathsf{Set}(y, vs) * \text{ret} = (v \in vs)\} \\
\{\mathsf{Set}(y, vs)\} & \text{sadd}(y, v) & \{\mathsf{Set}(y, \{v\} \cup vs)\} \\
\{\mathsf{Set}(y, vs)\} & \text{sremove}(y, v) & \{\mathsf{Set}(y, vs \setminus \{v\})\}
\end{array}$$

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

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

The $\textsc{SGap}(vs)$ action allows the thread to return the set containing vs 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.

To define concrete interpretations for the predicates, we first define $P_\in(h, v)$ and $P_\notin(h, v)$:

$$\begin{array}{rcl}
\mathsf{allgaps}(s) & \equiv & \circledast ws.\, [\textsc{SGap}(ws)]^s_1 \\
P_\lhd(h, v, s) & \equiv & \exists vs.\ v \lhd vs \wedge \left(\begin{array}{l} (\mathsf{allgaps}(s) * \mathsf{Set}(h.\text{set}, vs)) \\ \quad \vee \mathsf{Locked}(h.\text{lock}) * ([\textsc{SGap}(vs)]^s_1 \mathrel{-\!\!\circledast} \mathsf{allgaps}(s)) \end{array} \right) \\
& & \text{where } \lhd = \in \text{ or } \lhd = \notin
\end{array}$$

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

$$\begin{array}{rcl}
\mathsf{in}(h, v) & \equiv & \exists s.\, \mathsf{isLock}(h.\text{lock}) * [\textsc{SChange}(v)]^s_1 * \boxed{P_\in(h, v, s)}^s_{C(s,h)} \\
\mathsf{out}(h, v) & \equiv & \exists s.\, \mathsf{isLock}(h.\text{lock}) * [\textsc{SChange}(v)]^s_1 * \boxed{P_\notin(h, v, s)}^s_{C(s,h)}
\end{array}$$

The $\mathsf{in}(h, v)$ predicate gives a thread the permissions needed to acquire the lock, $\mathsf{isLock}(h.\text{lock})$, and to change whether v is in the set, $[\text{SCHANGE}(v)]^s_1$. 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 a negation on the set contains.

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

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

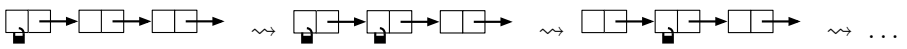
Verifying the Set Implementation. Given the definitions above, we can verify the implementations of the set module. Fig. 5 shows a proof of $\text{add}(h, v)$ when the value is not in the set. The case where the value is in the set, and the proofs of remove and contains follow a similar structure.

The most interesting steps of this proof are those before and after the operation sadd(h.lock), when the permissions $[\text{SCHANGE}(v)]^s_1$ and $[\text{SGAP}(vs)]^s_1$ are used to repartition between shared and local state. These steps are purely logical repartitioning of the state.

3.3 The Fine-grained Set

Our previous implementation of a concurrent set used a single global lock. We now consider a set implementation that uses a sorted list with one lock per node in the list. Our algorithm (adapted from [15, §9.5]) is given in Fig. 6.

The three module functions use the function $\text{locate}(h, x)$ that traverses the list from the head h up to the position for a node holding value x, whether or not such a node is present. It begins by locking the initial 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,

$\{\mathsf{out}(h, v)\}$
add(h,v)
$\left\{\exists s.\, \mathsf{isLock}(\mathrm{h.lock}) * [\textsc{SChange}(\mathrm{v})]_1^s * \boxed{P_{\notin}(\mathrm{h}, \mathrm{v}, s)}^{s}_{C(s,h)}\right\}$
lock(h.lock)
$\left\{\exists s.\, \mathsf{isLock}(\mathrm{h.lock}) * \mathsf{Locked}(\mathrm{h.lock}) * [\textsc{SChange}(\mathrm{v})]_1^s * \boxed{P_{\notin}(\mathrm{h}, \mathrm{v}, s)}^{s}_{C(s,h)}\right\}$
// use SChange *permission to extract* Set *predicate and* SGap *permission*
$\left\{\begin{array}{l}\exists s.\, \mathsf{isLock}(\mathrm{h.lock}) * [\textsc{SGap}(vs \cup \{\mathrm{v}\})]_1^s * [\textsc{SChange}(\mathrm{v})]_1^s * \mathsf{Set}(\mathrm{h.set}, vs) \\ \qquad * \boxed{\mathsf{Locked}(\mathrm{h.lock}) * ([\textsc{SGap}(vs \cup \{\mathrm{v}\})]_1^s \multimap\!\circledast\, \mathsf{allgaps}(s))}^{s}_{C(s,h)}\end{array}\right\}$
sadd(h.set,v)
$\left\{\begin{array}{l}\exists s.\, \mathsf{isLock}(\mathrm{h.lock}) * [\textsc{SGap}(vs \cup \{\mathrm{v}\})]_1^s * [\textsc{SChange}(\mathrm{v})]_1^s * \mathsf{Set}(\mathrm{h.set}, vs \cup \{\mathrm{v}\}) \\ \qquad * \boxed{\mathsf{Locked}(\mathrm{h.lock}) * ([\textsc{SGap}(vs \cup \{\mathrm{v}\})]_1^s \multimap\!\circledast\, \mathsf{allgaps}(s))}^{s}_{C(s,h)}\end{array}\right\}$
// use SGap *permission to put back* Set *and* SGap *permission*
$\left\{\exists s.\, \mathsf{isLock}(\mathrm{h.lock}) * \mathsf{Locked}(\mathrm{h.lock}) * [\textsc{SChange}(\mathrm{v})]_1^s * \boxed{P_{\in}(\mathrm{h}, \mathrm{v}, s)}^{s}_{C(s,h)}\right\}$
unlock(h.lock)
$\left\{\exists s.\, \mathsf{isLock}(\mathrm{h.lock}) * [\textsc{SChange}(\mathrm{v})]_1^s * \boxed{P_{\in}(\mathrm{h}, \mathrm{v}, s)}^{s}_{C(s,h)}\right\}$
}
$\{\mathsf{in}(h, v)\}$

Fig. 5. Proof of the add(h, v) operation for the coarse-grained set module.

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

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

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

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

Fig. 6. Lock-coupling list algorithm.

and so we require the isLock predicate to record permissions. Creating a lock gives full permission, locking requires only fractional permission and disposing a lock requires full permission in this scheme. Lock disposal is specified as

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

and for both lock implementations we have given, the function simply deallocates the 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.

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

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

$$\text{LGap}(n,t,x)\colon \begin{cases} \mathsf{Locked}(n) \;\rightsquigarrow\; \mathsf{link}(n,t) * [\text{LGap}(n,t,x)]^s_1 \\ \forall v_1,v_2.\begin{pmatrix}\mathsf{Locked}(n)\\ *\,\mathsf{val}(n,v_1)\\ *\,\mathsf{val}(t,v_2)\end{pmatrix} \rightsquigarrow \begin{pmatrix}[\text{LGap}(n,t,x)]^s_1 * \mathsf{node}(y,t,x)\\ *\,\mathsf{link}(n,y) * \mathsf{val}(n,v_1)\\ *\,\mathsf{val}(t,v_2) *\; \wedge\, v_1{<}x{<}v_2\end{pmatrix} \\ \begin{pmatrix}\mathsf{Locked}(n) *\\ \mathsf{Locked}(t) * \mathsf{val}(t,x)\end{pmatrix} \rightsquigarrow \begin{pmatrix}[\text{LGap}(n,t,x)]^s_1 *\\ [\text{LGap}(t,y,x)]^s_1 * \mathsf{link}(n,y)\end{pmatrix} \end{cases}$$

$$\text{LChange}(x)\colon \quad \forall n,t.\; [\text{LGap}(n,t,x)]^s_1 * \mathsf{link}(n,t) \quad \rightsquigarrow \quad \mathsf{Locked}(n)$$

Intuitively, the $\text{LGap}(n,t,x)$ actions allow a link to be replaced between n and t, such that it preserves the list's contents up to inclusion of x. Locked (and so, hidden) nodes of the list correspond to permissions on the action LGap.

The first LGap action simply puts the link back. The second updates the link to point to a new node y whose tail is t and value is x. We require that inserting this node maintains the ascending order of the list. This action is the basis of adding x to the set. The final action allows the link to be updated to skip an element, if that element is x. This is the basis of removing x from the set.

The $\text{LChange}(x)$ action allows a thread holding the lock to a node n to access it, by obtaining $\mathsf{link}(n,t)$, and to modify it up to adding or removing a node with value x, by obtaining the permission $[\text{LGap}(n,t,x)]^s_1$.

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

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

Fig. 7. Predicates for lock-coupling list.

predicates. The slsg ('sorted 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:

$$
\begin{aligned}
\mathsf{in}(h, x) &\equiv \exists s.\, \exists \pi > 0.\, \mathsf{isLock}(h, \pi) * [\textsc{LChange}(x)]^s_1 * \boxed{L_\in(h, x, s)}^s_{F(s,h)} \\
\mathsf{out}(h, x) &\equiv \exists s.\, \exists \pi > 0.\, \mathsf{isLock}(h, \pi) * [\textsc{LChange}(x)]^s_1 * \boxed{L_\notin(h, x, s)}^s_{F(s,h)}
\end{aligned}
$$

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

Verifying the set implementation Fig. 8 gives a sketch-proof for locate(x). The main loop searches through the list checking if the current element, c, is less than the element being searched for x. At the start of the loop, the thread has p locked, signified by having the $\textsc{Gap}(p, c, x)$ permission in its local state. The first step of the loop locks the new current element c. This is allowed as the $\mathsf{link}(p, c)$ predicate contains a isLock permissions for c. After, locking the thread additionally has a $\mathsf{Locked}(c)$ predicate in its local state.

The proof then uses the action in the local state $\textsc{LChange}(x)$, to swap the local Locked predicate, for the Gap action associate with c and the link between c and the next node. In some sense, we are publishing that we have locked this node. This thread now has two nodes locked in the list and can unlock the first. This unlocking first involves putting the Gap permission into the shared state, which returns the Locked predicate for that node to the threads local state. This allows the previous node to 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

Fig. 8. Proof for locate(x).

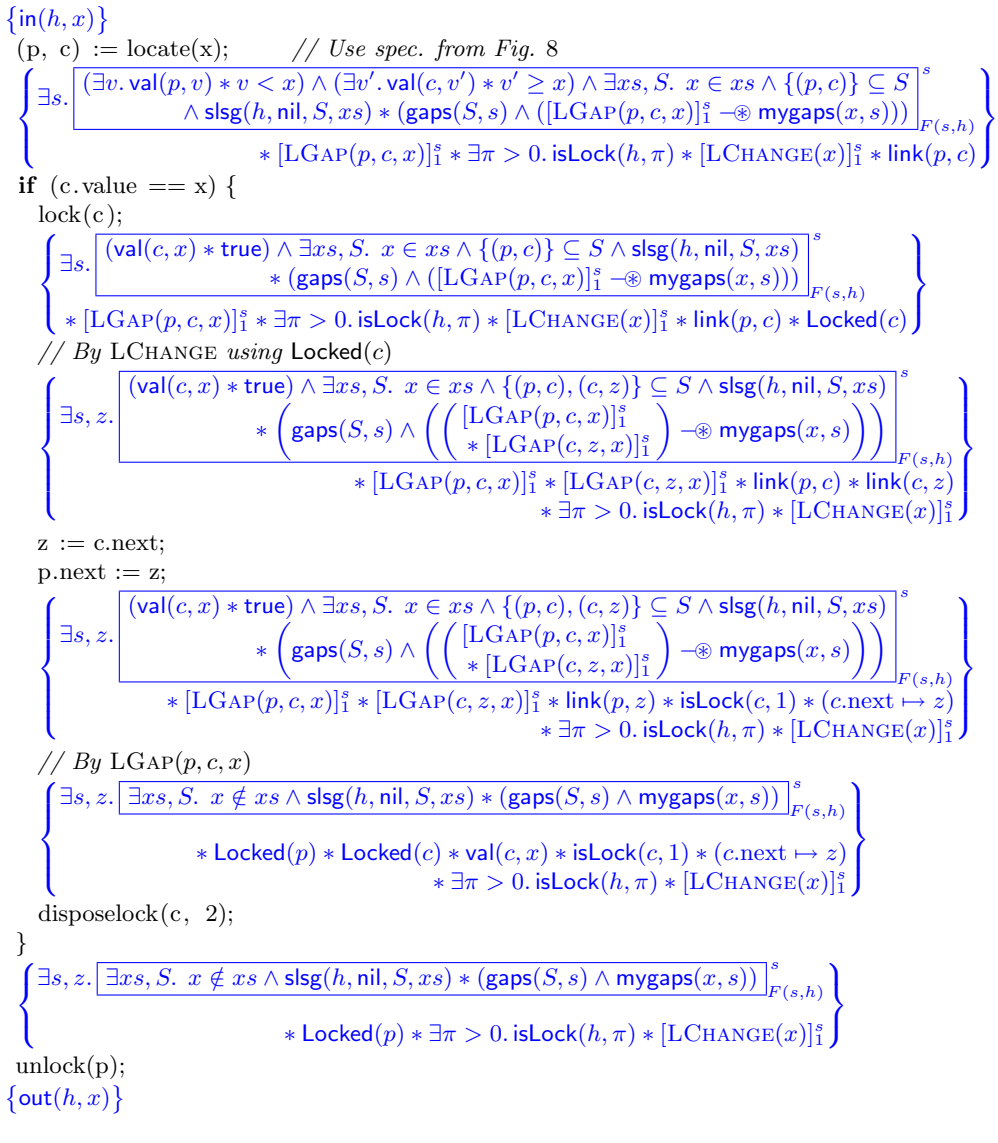


Fig. 9. Proof-sketch for remove(x).

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.

4 Semantics and Soundness

4.1 Assertion Syntax

Recall from §2.3 that our proof judgments have the form $\Delta, \Gamma \vdash \{P\}\, C\, \{Q\}$. Here, P and Q are *assertions* in the the set Assn. We also define a set of *basic assertions*, BAssn, which omit permissions, regions and predicates. Regions in assertions are annotated by *interference assertions* in the set IAssn. Δ is an *axiom definition* in the set Axioms. Finally, Γ is a *function specification* in the set $\mathsf{Triples}$. The syntax is defined as follows.

$$(\mathsf{Assn}) \quad P, Q ::= \mathsf{emp} \mid E_1 \mapsto E_2 \mid P * Q \mid P \mathrel{-\!\!\circledast} Q \mid \mathsf{false} \mid P \Rightarrow Q \mid \exists x.\, P \mid [\gamma(E_1, \ldots, E_n)]^R_\pi \mid \boxed{P}^R_I \mid \alpha(E_1, \ldots, E_n) \mid \circledast x.\, P$$

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

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

$$(\mathsf{Axioms}) \quad \Delta ::= \varnothing \mid \forall \vec{x}.\, P \implies Q \mid \forall \vec{x}.\, \alpha(\vec{x}) \equiv P \mid \Delta_1, \Delta_2$$

$$(\mathsf{Triples}) \quad \Gamma ::= \varnothing \mid \Gamma, \{P\} f \{Q\}$$

In the above definitions, 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.

4.2 Assertion Model

Let (Heap, $\uplus$, $\emptyset$) be any separation algebra [6] representing *machine states* (or *heaps*). Typically, we take Heap to be the set of finite partial functions from locations to values, and $\uplus$ to be the union of partial functions with disjoint domains: the standard separation logic model. We let h denote a heap.

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).

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

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 a syntactic interference assertion, from the set IAssn, that determines how threads may modify the shared region. A *shared state* in SState is defined to be a finite partial function mapping region names to logical states and interference assertions:

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

A *world* in World is a pair of a local (logical) state and a shared state, subject to a well-formedness condition. Informally, the well-formedness condition ensures that all parts of the state are disjoint and that each permission corresponds with an appropriate region; we defer the formal definition.

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

Given a logical state l, we write l_{H} and l_{P} to stand for the heap and permission assignment components respectively. We also write w_{L} and w_{S} to stand for the local and shared components of the world w respectively.

Recall from §2.2 that actions can be self-referential. For example, the action UNLOCK moves the permission $[\textsc{Unlock}]^r_1$ from shared to local 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 sequence of value parameters (*e.g.* $[\textsc{SChange}(v)]^s_1$ corresponds to the token $(s, \textsc{SChange}, v)$ with permission 1):

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

The semantics of a token is defined by an *interference environment* (see §4.4).

Permission assignments in Perm associate 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.[7]

The composition operator $\oplus$ 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

[7] This is the fractional permission model of Boyland [3]. As in Dodds *et al.* [9], we can extend our permission structure with a fourth possibility called a *deny*: a non-exclusive permission *prohibiting* both the environment and thread from performing the action. This requires only minor changes to the rely and guarantee functions presented later in this section – other definitions stay the same.

assignments: $l \oplus l' \stackrel{\text{def}}{=} (l_{\text{H}} \uplus l'_{\text{H}},\ l_{\text{P}} \oplus l'_{\text{P}})$. Composition on worlds is defined by composing local states and requiring that shared states be identical:

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

We write $\mathbf{0}_{\mathsf{Perm}}$ for the empty permission (which assigns zero permission to every token, i.e. $\lambda t.0$), and $[t \mapsto \pi]$ for the permission mapping the token t to π and all other tokens to 0.

The operator $\lfloor w \rfloor$ collapses a world w to a logical state. Here $\bigoplus$ is the natural lifting of $\oplus$ to sets. The operator $\lfloor\!\lfloor w \rfloor\!\rfloor$ further collapses w to a heap:

$$\lfloor w \rfloor \stackrel{\text{def}}{=} w_{\text{L}} \oplus \left(\bigoplus_{r \in \text{dom}(w_{\text{S}})} w_{\text{S}}(r)\right) \qquad\qquad \lfloor\!\lfloor w \rfloor\!\rfloor \stackrel{\text{def}}{=} (\lfloor w \rfloor)_{\text{H}}$$

The *action domain* of an interference assertion, $\text{adom}(I)$ is the set of action names and parameters that the specification allows.

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

We are finally in a position to define well-formedness for a world, $\mathsf{wf}(w)$.

$$\begin{aligned} \mathsf{wf}(w) \stackrel{\text{def}}{\iff} & \exists l \in \mathsf{LState}.\, l = \lfloor w \rfloor \wedge \\ & \forall r, \gamma, \vec{v}.\, l_{\text{P}}(r, \gamma, \vec{v}) > 0 \implies \exists l', I.\, w_{\text{S}}(r) = (l', I) \wedge (\gamma, \vec{v}) \in \text{adom}(I) \end{aligned}$$

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 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, so we assume an appropriate semantics. The semantics of IAssns can also depend on the semantics of free variables. We define $(\!|-|\!)_- : \mathsf{IAssns} \times \mathsf{Interp} \to \mathsf{IAssns}$ to replace the free variables with their values.

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

$$\begin{aligned} \delta \in \mathsf{PEnv} &\stackrel{\text{def}}{=} \mathsf{PName} \times \mathsf{Val}^* \to \mathcal{P}(\mathsf{World}) \\ i \in \mathsf{Interp} &\stackrel{\text{def}}{=} \mathsf{Var} \to \mathsf{Val} \end{aligned}$$

We assume that $\mathsf{RName} \cup (0, 1] \subset \mathsf{Val}$, so that variables may range over region names and fractions.

The cell assertion $\mapsto$, the separating star $*$ and the existential separating implication $-\!\circledast$ are standard. The quantifier $\circledast$ is infinitary version of $*$; that is,

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

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

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

the multiplicative analogue of $\forall$. The empty assertion emp asserts that the local state and permission are empty, but that the shared state can contain anything.

Abstract predicates, $\alpha(E_1, \ldots, E_n)$, are used to encapsulate concrete properties. For example, in the lock specification (§2.1), we used $\mathsf{Locked}(x)$ to assert that x is held by the current thread. The meaning of an abstract predicate is simply looked up the in the predicate environment, δ.

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 $[\![\pi]\!]_i$.

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), P asserts that the shared state for a lock is either locked or unlocked, and I defines the meaning of actions Lock and Unlock. We use $(\!|I|\!)_i$ as we need to bind the location x and region r to the correct values.

Separating conjunction behaves as conventional (non-separating) conjunction between shared-state assertions over the same region; that is: $\boxed{P}^r_I * \boxed{Q}^r_I \iff \boxed{P \wedge Q}^r_I$. We permit nesting of shared-state assertions. However, nested assertions can always be rewritten in equivalent unnested form:

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

In this paper, we only use nesting to ensure that shared and unshared elements can be referred to by a single abstract predicate. In future, nesting may be useful for defining mutually recursive modules.

4.4 Environment Semantics

An interference assertion defines the actions that are permitted over a region. For example, in the compare-and-swap lock implementation, (§2.2) the assertion $I(r, x)$ defines the action Lock as $x \mapsto 0 * [\text{Unlock}]^r_1 \rightsquigarrow x \mapsto 1$.

Semantically, an interference assertion is defined as a map from tokens to sets of shared-state pairs (what we call an *interference environment*):

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

A primitive interference assertion defines an interference environment that maps the token $(r, \gamma, \vec{v})$ to an action relation corresponding to transitions from states satisfying $\boxed{P}^r_I$ to $\boxed{Q}^r_I$. The relation does not involve local state, and only the region r of the shared state changes. The action Lock defines a relation from shared states where the lock region is unlocked, to ones where it is locked. Composition of interference assertions is defined by union of relations.

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

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

Given a region name r and an interference assertion I, $\mathsf{all}(I, r)$ is the logical state assigning full permission to all tokens with region r whose semantics is defined by I.

$$\mathsf{all}(I, r) \stackrel{\text{def}}{=} \left(\emptyset,\ \bigoplus\nolimits_{(\gamma, \vec{v}) \in \mathrm{adom}(I)} [(r, \gamma, \vec{v}) \mapsto 1]\right)$$

The guarantee G_δ describes what updates to the world the thread is allowed to perform. The thread can update its local state as it pleases, but changes to the shared state must be according to some action 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 the shared lock. Without this restriction other threads could potentially unlock the lock. It is also very important that updates preserve the total amount permission in the world ($\lfloor w \rfloor_\mathrm{P} = \lfloor w' \rfloor_\mathrm{P}$), so that the thread cannot acquire permission to do an action out of thin air.

Moreover, the thread can create a new region by giving away some of its local state and gaining full permission on the newly created region. This is described by G^c. Conversely, it can destroy any region that it fully owns and grab ownership of the state it protects (cf. $(G^\mathrm{c})^{-1}$).

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

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

We now define formally the notion of *update implication*, $P \Longrightarrow_\delta^{\{p\}\{q\}} Q$. This states that a rewrite from p to q is permitted in a larger world satisfying P, and that the resulting world satisfies Q. As the definition of update implication erases the divisions between regions, it permits repartitioning between regions. Repartitioning is sound provided it is permitted by the guarantee, $(G_\delta)^*$.

Definition 1 (Update Implication). *$P \Longrightarrow_\delta^{\{p\}\{q\}} Q$ holds iff, for every world w_1 in $[\![P]\!]_{\delta,i}$, there exists a heap h_1 in $[\![p]\!]_i$ and a residual heap h' such that*

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

Note that if $p = q = \mathsf{emp}$, then the update implication preserves the concrete state, and only allows the world to be repartitioned. Hence we call this special case *repartitioning implication* and write $P \Longrightarrow_\delta Q$ as a shorthand for $P \Longrightarrow_\delta^{\{\mathsf{emp}\}\{\mathsf{emp}\}} Q$. Recall from the proof of the compare-and-swap lock implementation (§2.2) that repartitioning implication was used to create a new shared regions when making a lock.

The rely R_δ describes the possible world updates that the environment can do. Intuitively, it models 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 (cf. R^{c}) or can destroy an existing region (cf. $(R^{\mathrm{c}})^{-1}$) provided that no permission for that region exists elsewhere in the world.

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

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

These definitions allow us to define stability of assertions. We say that an assertion is *stable* if and only if it cannot be falsified by the interference from other threads that it permits.

Definition 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, we say that a predicate environment is stable if and only if all the predicates it defines are stable.

Definition 3 (Predicate Environment Stability). *$\mathsf{pstable}(\delta)$ holds iff for all $X \in \mathrm{rng}(\delta)$, for all w and w', if $w \in X$ and $(w, w') \in R_\delta$, then $w' \in X$.*

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

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

$$\frac{\Delta; \Gamma \vdash \{P\}\ C\ \{Q\} \qquad \Delta \vdash \mathsf{stable}(R)}{\Delta; \Gamma \vdash \{P * R\}\ C\ \{Q * R\}}\ (\textsc{Frame}) \qquad \frac{\Delta; \Gamma \vdash \{P'\}\ C\ \{Q'\} \qquad \Delta \vdash P \Longrightarrow P' \qquad \Delta \vdash Q' \Longrightarrow Q}{\Delta; \Gamma \vdash \{P\}\ C\ \{Q\}}\ (\textsc{Conseq})$$

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

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

Fig. 11. Selected proof rules.

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

$$\begin{aligned} [\![\varnothing]\!] &\stackrel{\text{def}}{=} \{\delta \mid \mathsf{pstable}(\delta)\} & [\![\Delta_1, \Delta_2]\!] &\stackrel{\text{def}}{=} [\![\Delta_1]\!] \cap [\![\Delta_2]\!] \\ [\![\forall \vec{x}.\, \alpha(\vec{x}) \equiv P]\!] &\stackrel{\text{def}}{=} \{\delta \mid \mathsf{pstable}(\delta) \wedge \forall \vec{v}.\ \delta(\alpha, \vec{v}) = [\![P]\!]_{\delta, [\vec{x} \mapsto \vec{v}]}\} \\ [\![\forall \vec{x}.\, P \Rightarrow Q]\!] &\stackrel{\text{def}}{=} \{\delta \mid \mathsf{pstable}(\delta) \wedge \forall \vec{v}.\ [\![P]\!]_{\delta, [\vec{x} \mapsto \vec{v}]} \subseteq [\![Q]\!]_{\delta, [\vec{x} \mapsto \vec{v}]}\} \end{aligned}$$

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:

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

We require that atomic statements $\langle C \rangle$ are not nested and that function names $f_1 \ldots f_n$ for a let are pairwise distinct. Here c stands for basic commands, modelled semantically as subsets of $\mathcal{P}(\mathsf{Heap} \times \mathsf{Heap})$ satisfying the locality conditions of Calcagno *et al.* [6].

Judgements about such programs have the form $\Delta; \Gamma \vdash \{P\}\, C\, \{Q\}$. This judgement 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.

A selection of the proof rules for our Hoare-style program logic are shown in Fig. 11. These rules are modified from RGSep [25] and deny-guarantee [9]. All

of the rules in our program logic carry an implied assumption that the pre- and post-conditions of their judgements are stable.

The judgement $\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\}$ appearing in the Atomic and Prim rules is a judgment in standard sequential separation logic. The other minor judgments in the rules 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 $\forall \delta \in [\![\Delta]\!]\,.\, \delta \in [\![\Delta']\!]$.

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 predicates are satisfiable; otherwise the premise of a proof rule could be vacuously true. We ensure this holds by checking that only positive occurrences of the predicate appear in its definition.

The Atomic and Conseq rule were discussed in depth in §2.3. That section also discussed a rule for modules, which can be derived as follows:

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

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.

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 $C, h \xrightarrow{\eta} \mathit{fault}$.

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

To define the meaning of judgments, we first define the notion of a logical configuration $(C, w, \eta, \delta, i, Q)$ being safe for at least n steps:

Definition 4 (Configuration safety). *C, w, η, δ, i, Q $safe_0$ always holds; and C, w, η, δ, i, Q $safe_{n+1}$ iff the following four conditions hold:*

1. *For all w', if $(w, w') \in R_\delta$ then $C, w', \eta, \delta, i, Q$ $safe_n$;*
2. *$\neg((C, \llbracket w \rrbracket) \xrightarrow{\eta} fault)$;*
3. *For all C' and h', if $(C, \llbracket w \rrbracket) \xrightarrow{\eta} (C', h')$, then there there exists w' such that $h' = \llbracket w' \rrbracket$, $(w, w') \in (G_\delta)^*$, and $C', w', \eta, \delta, i, Q$ $safe_n$; and*
4. *If $C = \mathbf{skip}$, then there exists w' such that $\llbracket w \rrbracket = \llbracket w' \rrbracket$, $(w, w') \in (G_\delta)^*$, and $w' \in [\![Q]\!]_{\delta,i}$.*

This definition says that a configuration is safe provided that: (1) changing the w 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 $(G_\delta)^*$ in the third and fourth conjuncts allows the world to be repartitioned.

We define the meaning of judgements in terms of configuration safety:

Definition 5 (Judgement Semantics). *$\Delta; \Gamma \models \{P\}\, C\, \{Q\}$ holds iff*

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

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

Theorem 1 (Soundness). *If $\Delta; \Gamma \vdash \{P\}\, C\, \{Q\}$, then $\Delta; \Gamma \models \{P\}\, C\, \{Q\}$.*

We prove this by structural induction on the proof rules. See Appendix B for full details. The most interesting case is the Par rule. The simplicity of this rule embodies the compositionality of our logic. To prove it sound, the following lemma is required.

Lemma 1 (Abstract state locality).

If $(C, \llbracket w_1 \oplus w_2 \rrbracket) \xrightarrow{\eta} (C', h)$ and $C, w_1, \eta, \delta, i, Q$ $safe_{n+1}$, then $\exists w_1', w_2'$ such that $(C, \llbracket w_1 \rrbracket) \xrightarrow{\eta} (C', \llbracket w_1' \rrbracket)$ and $h = \llbracket w_1' \oplus w_2' \rrbracket$ and $(w_1, w_1') \in G_\delta$ and $(w_2, w_2') \in R_\delta$.

Proof. To prove this, we require that basic commands obey a concrete locality assumption. We must also prove that 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 9. □

The lemma shows that a program execution only affects the parts of the world required for it to run safely. The soundness of Par follows because the lemma shows that threads are safely contained within their abstract footprints.

5 Conclusions and Related Work

We have presented a logic that provides abstraction in a concurrent setting. It brings together three streams of research: (1) abstract predicates [23], using predicates to abstract the internal details of a module or class; (2) deny-guarantee [9], reasoning about concurrent programs using rely-guarantee and separation logic; and (3) abstract local reasoning [8], formalising the fiction of disjointness obtained when reasoning at the abstract level.

Our work on concurrent abstract predicates has been strongly influenced by O'Hearn's concurrent separation logic (CSL) [21]. CSL takes statically allocated locks as a primitive. With CSL, we can reason about programs with these 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 to deal with dynamically-allocated locks [13, 17] and re-entrant locks [14], by providing new proofs rules and assertions. This work uses invariants to abstract the state over which interference acts. Parkinson *et al.* [22] have shown how the CSL approach can be used to reason about concurrent algorithms that do not use locks. However, representing the complex interactions of concurrent algorithms in a single invariant can require auxiliary state making proofs less tractable.

Jacobs and Piessens [19] have developed an approach to reasoning abstractly based on CSL. They use a logic similar to Gotsman [13] to reason about dynamically allocated locks with associated invariants. Their logic uses auxiliary state to express the temporal nature of interference. To deal with auxiliary state in a modular way they add a special implication that allows the auxiliary state to be changed in any way that satisfies the invariant. This implication is remarkably similar to our repartitioning implication $\Longrightarrow$. Their implication can be used in the specification of a module, allowing the auxiliary state of the client to be updated during the module's execution. We believe that this technique could be soundly applied with our repartitioning implication, which would simplify the use of the lock specification in the two set algorithms.

An alternative approach to using invariants is to abstract interference over the shared state by relations modelling the interaction of different threads: the rely-guarantee method [20]. There have been two recent logics that combine RG with separation logic: RGSep [25] and SAGL [11]. Both logics allow more elegant proofs of concurrent algorithms than the invariant-based proofs, but they have the serious drawback that interference on the shared state cannot be abstracted. Feng's Local Rely-Guarantee [10] improves the locality of RGSep and SAGL, by scoping interference with a precise footprint and having special rules for framing and hiding interference, but this approach still has no support for abstraction. Meanwhile, Dinsdale-Young, Gardner and Wheelhouse used RGSep to analyse a concurrent B-tree algorithm [7], and conjectured ideas about abstraction which led to this current collaboration.

We have built on deny-guarantee reasoning [9], a reformulation of rely-guarantee introduced to reason about dynamically scoped concurrency. This paper has combined a simplified form of deny-guarantee with the earlier work on RGSep.

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 (e.g., read-only, read-write,...).

The concept of the fiction of disjointness presented here arose from Gardner *et al.*'s work on abstract local reasoning about program modules using context logic [4, 5, 12]. Recently, Dinsdale-Young, Gardner and Wheelhouse have shown how to justify such abstract local reasoning about modules by relating it to separation logic reasoning about their non-local implementations [8]. Reasoning abstractly about modules thus provides a fiction of separation, which is not necessarily mirrored in their implementations.

We should point out that proofs in our proof system seem to be slightly more complex in practice than RGSep and SAGL, as can be seen by comparing the lock-coupling list proof with the RGSep one [25]. This might just 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 [16]. Linearisability provides a fiction of atomicity allowing "reason[ing] about properties of concurrent objects given just their (sequential) specifications" [16]. 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 the one logical formalism.

Acknowledgements Thanks to Richard Bornat, Alexey Gotsman, Suresh Jagannathan, Mohammad Raza, John Wickerson and Mark Wheelhouse for useful discussions and feedback. This work was supported by an EPSRC doctoral training account (Dinsdale-Young), 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. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
3. J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.
4. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL*, 2005.
5. C. Calcagno, P. Gardner, and U. Zarfaty. Local reasoning about data update. *Festschrift* Computation, Meaning and Logic: Articles dedicated to Gordon Plotkin, 172, 2007.
6. 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.

7. T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Local reasoning about a concurrent B*-list algorithm. Talk and technical report, see `http://www.doc.ic.ac.uk/~td202/`, 2009.
8. T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstract local reasoning. Imperial technical report, 2010.
9. M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.
10. X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.
11. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.
12. P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local reasoning about dom. In *PODS*, 2008.
13. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.
14. C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's Reentrant Locks. In *APLAS*, pages 171–187, 2008.
15. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, 2008.
16. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.
17. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.
18. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, Jan. 2001.
19. 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.
20. C. B. Jones. Annoted bibliography on rely/guarantee conditions. `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf`, 2007.
21. P. W. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.
22. M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, pages 297–302, Jan. 2007.
23. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
24. V. Vafeiadis. *Modular Fine-Grained Concurrency Verification*. PhD thesis, University of Cambridge, July 2007.
25. 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 [24]. However, this prior semantics required two operational semantics – a logical semantics with proof annotations to separate and a machine semantics erasing such annotations. Using Vafeiadis's new approach to soundness of RGSep, Definition 4, we can avoid this requirement for two semantics.

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

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

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

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

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

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

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

Fig. 12. Operational 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 a function $c\colon \mathsf{States} \times \mathsf{States}$.

Theorem 2 (Well-definedness). *For any program C, state h and environment η, there exists a transition $(C,h) \xrightarrow{\eta} (C',h')$ or $(C,h) \xrightarrow{\eta} \mathit{fault}$.*

Proof. Holds trivially by induction for most cases. The only interesting case is the rule dealing with atomic commands. For this, we observe that a derivation sequence beginning with C cannot result in the program $\langle C \rangle$, so the derivation must be well-founded. □

B Soundness of proof system

Lemma 2 (Skip safety). *If $\mathsf{stable}_\delta(Q)$ and $w \models_\delta Q$ then $\mathbf{skip}, w, \eta, \mathbb{I}, \delta, Q$ 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 $w' \models Q$. 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. □

Assumption 1 (Primitive locality) *We assume that elements of the primitive update function* c: $\mathsf{States} \times \mathsf{States}$ *are local. That is:*

1. *If no* h_2 *exists such that* $(h_1 \oplus h, h_2) \in c$*, then no* h_2' *exists such that* $(h_1, h_2') \in c$*, and*
2. *if* $(h_1 \oplus h, h_2) \in c$ *and there exists an* h_3 *such that* $(h_1, h_3) \in c$ *then there exists an* h_4 *such that* $(h_1, h_4) \in c$ *and* $h_2 = h \oplus h_4$*.*

Lemma 3 (Concrete locality). *We need both single-step and multi-step versions of locality.*

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

Proof. As in the previous lemma, the proof needs both single-step and multi-step versions of locality. The result is proved by induction over the structure of a faulting derivation or derivation sequence.

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

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 any 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 3 and the first inductive assumption. □

Lemma 5 (Guarantee locality).
If $(w_1, w_1') \in G_\delta$ and $w_1 \oplus w_2$ is defined, then there exists w_2' such that $(w_1 \oplus w_2, w_1' \oplus w_2') \in G_\delta$ and $w_1' \oplus w_2'$ is defined.

Proof. By the definition of state composition, $(w_1)_\mathrm{S} = (w_1 \oplus w_2)_\mathrm{S}$ 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. □

Lemma 6 (Guarantee locality II). *If $(w_1, w_1') \in G_\delta$ and $(w_2, w_2') \in R_\delta$ and $w_1 \oplus w_2$ defined and $w_1' \oplus w_2'$ defined, then $(w_1 \oplus w_2, w_1' \oplus w_2') \in G_\delta$.*

Proof. Similar to previous. □

Lemma 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. By the definition of state composition, $(w_1)_\mathrm{S} = (w_2)_\mathrm{S} = (w_1 \oplus w_2)_\mathrm{S}$, and $(w_1')_\mathrm{S} = (w_2')_\mathrm{S} = (w_1' \oplus w_2')_\mathrm{S}$. Also, by permission composition, if $((w_1 \oplus w_2)_\mathrm{L})_\mathrm{P}(r, \gamma, \vec{v}) \in [0, 1)$ then $((w_1)_\mathrm{L})_\mathrm{P}(r, \gamma, \vec{v}) \in [0, 1)$ and $((w_2)_\mathrm{L})_\mathrm{P}(r, \gamma, \vec{v}) \in [0, 1)$. If w allows a region to be created or removed, then so will both w_1 and w_2. The rest follows immediately by the definition of rely. □

Lemma 8 (Containment of rely in guarantee).
If $w_1 \oplus w_2$ is defined and $(w_1, w_1') \in G_\delta$ then there exists a w_2' such that $(w_2, w_2') \in R_\delta$ 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. □

Lemma 9 (Abstract state locality).
If $(C, \llbracket w_1 \oplus w_2 \rrbracket) \xrightarrow{\eta} (C', h)$ and $C, w_1, \eta, \delta, i, Q$ safe_{n+1}, then $\exists w_1', w_2'$ such that $(C, \llbracket w_1 \rrbracket) \xrightarrow{\eta} (C', \llbracket w_1' \rrbracket)$ and $h = \llbracket w_1' \oplus w_2' \rrbracket$ and $(w_1, w_1') \in G_\delta$ and $(w_2, w_2') \in R_\delta$.

Proof. We first construct a remainder state h' from the local portion of w_2, so $h' = \lfloor (w_2)_{\mathrm{L}} \rfloor$. By the definition of composition $\llbracket w_1 \oplus w_2 \rrbracket = \llbracket w_1 \rrbracket \oplus h'$.

By appeal to Lemma 3 (Concrete single-step locality) there exists a h'' such that $C, \llbracket w_1 \rrbracket \xrightarrow{\eta} C', h''$ and $h' \oplus h'' = h$. By the definition of safety there must exist a w_1' such that $\llbracket w_1 \rrbracket = h''$ and $(w_1, w_1') \in G_\delta$.

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'. It therefore holds trivially that $\llbracket w_1' \rrbracket \oplus h' = \llbracket w_1' \oplus w_2' \rrbracket$, proving the main requirement. It remains to prove that $(w_2, w_2') \in R_\delta$. This follows immediately by appeal to Lemma 8. □

Lemma 10 (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. $\neg((C_1 \| C_2, \llbracket w \rrbracket) \xrightarrow{\eta} \mathit{fault})$
3. $(C_1 \| C_2, \llbracket w \rrbracket) \xrightarrow{\eta} (C', h') \implies \exists w'. \llbracket w' \rrbracket = h' \wedge (w, w') \in (G_\delta)^* \wedge C', w', \eta, \delta, i, Q_1 * Q_2$ safe$_n$

For the first clause, we assume that $(w, w') \in R_\delta$. By Lemma 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 $C_2, w_2', \eta, \delta, i, Q_2$ safe$_n$ and $C_1, w_1', \eta, \delta, i, Q_1$ 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 4 (Concrete fault locality) the threads must also fault in $\llbracket w_1 \rrbracket$ or $\llbracket w_2 \rrbracket$ 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 reducible. We first consider the **skip** case. We pick the value of w' as w. By the semantics, $C' = \mathbf{skip}$ and $h' = \llbracket w \rrbracket$. The first and second requirements of this clause hold trivially, and the third holds by Lemma 2 (Skip safety) and the assumptions of safety.

Now consider the case where C_1 or C_2 is reducible. We only consider C_1; the C_2 case is identical. It must hold that $(C_1, \llbracket w \rrbracket) \xrightarrow{\eta} (C_1', h')$ and $C' = C_1' \| C_2$. By appeal to Lemma 9 (Abstract locality) we have that $(C_1, \llbracket w_1 \rrbracket) \xrightarrow{\eta} (C_1', \llbracket w_1' \rrbracket)$ and $(w_1, w_1') \in G_\delta$ and $(w_2, w_2') \in R_\delta$ and $\llbracket w_1' \oplus w_2' \rrbracket = h'$. Consequently by the safety assumptions we have $C_1', w_1', \eta, \delta, i, Q_1$ safe$_n$. and $C_2, w_2', \eta, \delta, i, Q_2$ safe$_n$. Hence by inductive hypothesis, we have $C_1' \| C_2, w_1' \oplus w_2', \eta, \delta, i, Q_1 * Q_2$ safe$_n$. By Lemma 6 (Guarantee locality II), we get $(w_1 \oplus w_2, w_1' \oplus w_2') \in G_\delta$ □

Lemma 11. *If $C, w_1, \eta, \delta, i, Q$ safe$_n$ and $w = w_1 \oplus w_2$ and $w_2, i \models F$ 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. $\neg((C, \llbracket w \rrbracket) \xrightarrow{\eta} \mathit{fault})$
3. $(C, \llbracket w \rrbracket) \xrightarrow{\eta} (C', h') \implies \exists w'. \llbracket w' \rrbracket = h' \wedge (w, w') \in (G_\delta)^* \wedge C', w', \eta, \delta, i, Q * F$ safe_n
4. $C = \mathbf{skip} \implies \exists w'. (w, w') \in (G_\delta)^* \wedge \llbracket w \rrbracket = \llbracket w' \rrbracket \wedge w', i \models_\delta Q * F$

For the first clause, we can assume $(w, w') \in R_\delta$ which with Lemma 7 (Rely locality) gives $(w_1, w_1') \in R_\delta$ and $(w_2, w_2') \in R_\delta$ and $w' = w_1' \oplus w_2'$. As F is stable, we know $w_2', i \models F$. By the inductive assumption we thus have $C, w_1', \eta, \delta, i, Q * F$ safe_n, which completes the case.

The second clause holds by the contrapositive of Lemma 4 (Concrete fault locality).

For the third clause, we assume a transition $C, \llbracket w \rrbracket \xrightarrow{\eta} C', h'$. By appeal to Lemma 9 (Abstract state locality) there exists a transition $C, \llbracket w_1 \rrbracket \xrightarrow{\eta} C', h''$ and there exist worlds w_1', w_2' such that $\llbracket w_1' \oplus w_2' \rrbracket = h'$. This satisfies the first requirement. It also holds that $(w_1, w_1') \in G_\delta$, so the second requirement follows by Lemma 5 (Guarantee locality). The third requirement follows immediately from the inductive assumption.

For the fourth clause assume $C = \mathbf{skip}$. By assumption we know $(w_1, w_1') \in (G_\delta)^*$ and $\llbracket w_1 \rrbracket = \llbracket w_1' \rrbracket$ and $w_1', i \models_\delta Q$. By Lemma 5 (Guarantee locality) we know $(w_1 \oplus w_2, w_1' \oplus w_2') \in (G_\delta)^*$ and $(w_2, w_2') \in (R_\delta)^*$. As F is stable, we know $w_2', i \models F$, and hence $w_1' \oplus w_2', i \models Q * F$ as required. □

Lemma 12 (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 a w, i such that $w, i \models P * F$. Note that by the definition of $*$, $\exists w_1, w_2$ such that $w_1, i \models P$, $w_2, i \models F$, and $w = w_1 \oplus w_2$. Thus the result follows directly from Lemma 11. □

Lemma 13.
If $\delta, \eta \models_n \{P'\}\, C\, \{Q\}$*, and if* $(w, w') \in (R_\delta)^*$ *then there exists* w'' *such that* $(w', w'') \in G_\delta$ *and* $\llbracket w' \rrbracket = \llbracket w'' \rrbracket$ *and* $w'' \in [\![P']\!]_{i,\delta}$*, then* C, w, η, δ, i, Q *safe*$_n$.

Proof. Induction on n. The zero case is trivial. For the inductive case we must prove 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. $\neg((C, \llbracket w \rrbracket) \xrightarrow{\eta} \mathit{fault})$
3. $(C, \llbracket w \rrbracket) \xrightarrow{\eta} (C', h') \implies \exists w'. \llbracket w' \rrbracket = h' \wedge (w, w') \in (G_\delta)^* \wedge C', w', \eta, \delta, i, Q$ safe_n
4. $C = \mathbf{skip} \implies \exists w'. (w, w') \in (G_\delta)^* \wedge \llbracket w \rrbracket = \llbracket w' \rrbracket \wedge w', i \models_\delta Q$

First clause follows directly from induction. Second and third clauses hold trivially as $\llbracket \_ \rrbracket$ is preserved by guarantee steps. Fourth clause is trivial. □

Lemma 14 (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. Follows directly from Lemma 13, and the definitions of $\Longrightarrow$ and stable. □

Lemma 15. *If* $C, w, \eta, \delta, i, Q'$ safe_n *and* $Q' \Longrightarrow_\delta Q$*, then* C, w, η, δ, i, Q safe_n.

Proof. By induction on n. First three clauses of the definition follow trivially. For the fourth, we can assume

$$\exists w'.\, (w, w') \in (G_\delta)^* \wedge [\![w]\!] = [\![w']\!] \wedge w', i \models_\delta Q'$$

We must prove

$$\exists w'.\, (w, w') \in (G_\delta)^* \wedge [\![w]\!] = [\![w']\!] \wedge w', i \models_\delta Q$$

This follows by definition of $\Longrightarrow$ and transitivity. □

Lemma 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. Follows directly from previous lemma. □

Lemma 17 (Atomic safety). *If* $\vdash_{\mathsf{SL}} \{p\}C\{q\}$ *and* $P \Longrightarrow_\delta^{\{p\}\{q\}} Q$*, 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 such that $w, i \models_\delta P$. 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. $\neg(\langle C\rangle, [\![w]\!] \xrightarrow{\eta} \mathit{fault})$
3. $\langle C\rangle, [\![w]\!] \xrightarrow{\eta} C', h' \implies \exists w'.\, [\![w']\!] = h' \wedge (w, w') \in (G_\delta)^* \wedge C', w', \eta, \mathbb{I}, \delta, Q$ safe_n

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

By the semantics of $\vdash_{\mathsf{SL}} \{p\}C\{q\}$, for any h such that $h \models p$, it holds that $\neg((C, h) \xrightarrow{\eta}{}^* \mathit{fault})$. By the definition of update implication, there must exist states h', h'' such that $[\![w]\!] = h' \oplus h''$ and $h' \models p$. The second clause therefore follows by the contrapositive of Lemma 4 (Concrete fault locality).

For clause 3, we assume that there exists a transition $\langle C\rangle, [\![w]\!] \xrightarrow{\eta} C', h_3$. By reduction rules, $C' = \mathbf{skip}$. By Lemma 4 (Concrete fault locality) there exists a transition $\langle C\rangle, h' \xrightarrow{\eta} \mathbf{skip}, h_4$ such that $h_4 \oplus h'' = h_3$. By the semantics of $\vdash_{\mathsf{SL}} \{p\}C\{q\}$, it must hold that $h_4 \models q$. By the definition of update implication, there exists a w_2 such that $w_2 \models Q$, $(w, w_2) \in G_\delta$, and $h_4 \oplus h_3 = [\![w_2]\!]$. Consequently $h_3 = [\![w_2]\!]$, proving the first two requirements. As Q is stable, the third requirement is ensured by Lemma 2 (Skip safety). □

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

Proof. We first claim that:

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

when $\Delta \vdash \mathsf{stable}(R)$. 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, $\mathbb{I}$, and $\delta' \in [\![\Delta]\!]$, there exists a $\delta.\, \delta \in [\![\Delta \wedge \forall \overline{x}.\ \alpha(\overline{x}) \Leftrightarrow R]\!]$ such that

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

and for any assertion P, it holds that

$$w \models_{\delta'} P \iff w \models_{\delta} P$$

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. □

Theorem 3 (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 2 (Skip safety).

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

Holds by Lemma 10 (Parallel safety decomposition).

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

Holds by Lemma 12 (Frame safety).

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

Holds by Lemma 17 (Atomic safety).

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

For simplicity we treat the P and Q implications separately - no loss of generality results. The P case follows from Lemma 14 (Pre-partitioning safety), while the Q case follows from Lemma 16 (Post-partitioning safety).

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

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