Concurrent Abstract Predicates

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

[1] Imperial College, London, {td202,pg}@doc.ic.ac.uk
[2] University of Cambridge, {md466,mjp41,vv216}@cl.cam.ac.uk

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

1 Introduction

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

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

This combination of abstract predicates with the low-level locality from separation logic supports coarse-grained reasoning about modules, where each data structure is represented by a single abstract predicate. However, we often need to reason about modules in a fine-grained manner, where many abstract predicates refer to properties of the same data structure. For example, a fine-grained

specification for the set module would allow element 6 to be manipulated separately from the rest of the set. Fine-grained reasoning of this sort is advocated by context logic [3].

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

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

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

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

2 Informal Development

We develop our core idea, to define abstract specifications for concurrent modules and prove that concrete module implementations satisfy these specifications.

We motivate our work using a lock module, one of the simplest examples of concurrent resource sharing. We define an abstract specification for locks, and give two implementations satisfying the specification.

2.1 Lock Specification

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

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

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

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

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

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

The first axiom allows the client to share freely the knowledge that x is a lock.[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 `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;
}
```

(Here angle brackets denote atomic statements.)

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

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

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 module state.

To describe this internal interface, we extend separation logic with two assertions, the shared region assertion $\boxed{P}^r_{I(\vec{x})}$ and the permission assertion $[\mathrm{A}]^r_\pi$. The shared region assertion $\boxed{P}^r_{I(\vec{x})}$ specifies that there is a shared region of memory, identified by label r, and that the entire shared region satisfies P. The shared state is indivisible so that all threads maintain a consistent view of it. This is expressed by the logical equivalence $\boxed{P}^r_{I(\vec{x})} * \boxed{Q}^r_{I(\vec{x})} \Leftrightarrow \boxed{P \wedge Q}^r_{I(\vec{x})}$. 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 [2], the permission π can be the fractional permission, $\pi \in (0, 1)$, denoting that both the thread and the environment can do the action, or the full permission, $\pi = 1$, denoting that the thread can do the action but the environment cannot.[5] We now have the machinery to interpret our lock predicates concretely:

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

The abstract predicate $\mathsf{isLock}(x)$ is interpreted by the concrete, implementation-specific assertion on the right-hand side. This specifies that the 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 has it).

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

The actions permitted on the lock's shared region are declared in $I(r, x)$. Actions describe how either the current thread or the environment may modify the shared 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

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

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

The LOCK action requires that the shared region contains the unlocked lock ($x \mapsto 0$) and full permission $[\text{UNLOCK}]^r_1$ 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 ($[\text{UNLOCK}]^r_1$ has gone from the shared state). The movement of $[\text{UNLOCK}]^r_1$ 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.

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 must remain true. Self-stability ensures that a client can use these predicates without having to consider the module's internal interference. For example, assume that the predicate $\mathsf{Locked}(x)$ is true. There are two actions the environment can perform that can potentially affect the location x:

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

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 interference from the environment since another thread could acquire the lock. It does imply the fourth assertion, which is stable under such interference. The semantics of assertions allows us to forget about the shared region, resulting in the post-condition, emp.

For the `lock` case, the key proof step is the atomic compare-and-swap command in the loop. If successful, this command updates the location referred to by x in the shared state region from 0 to 1. This update is allowed because of

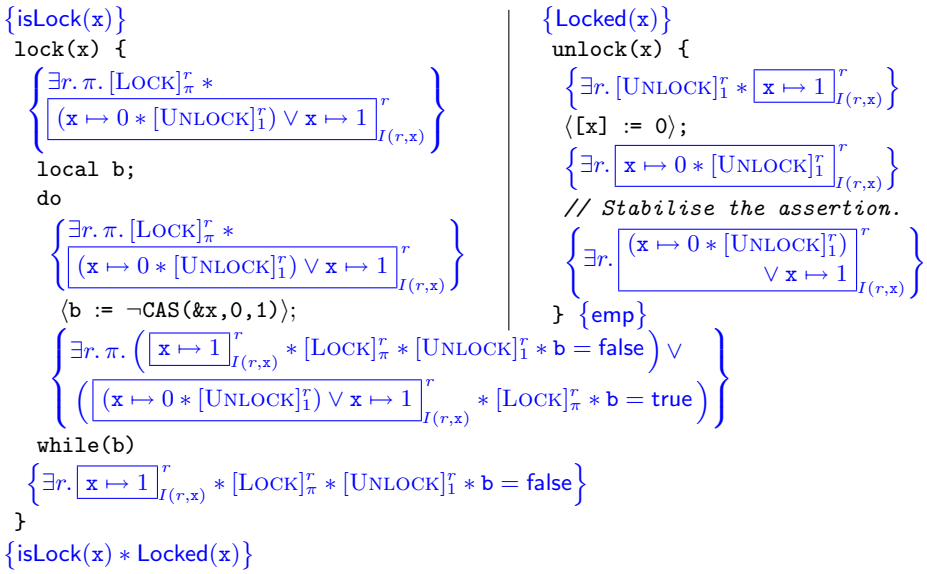


Fig. 1. Verifying the compare-and-swap lock implementation: `lock` and `unlock`.

the permission $[\textsc{Lock}]^r_\pi$ in the local state and the action in $I(r,x)$. The post-condition 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 pre-condition is still satisfied. This post-condition is stable and so the Hoare triple is valid.

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

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

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

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

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

2.3 The Proof System

We give an informal description of the proof system, with the formal details given in §4. Judgements in our proof system have the form $\Delta; \Gamma \vdash \{P\}C\{Q\}$, where Δ contains predicate definitions and axioms, and Γ presents abstract specifications of the functions used by C. The local Hoare triple $\{P\}C\{Q\}$ has the fault-avoiding partial-correctness interpretation advocated by separation logic: if the

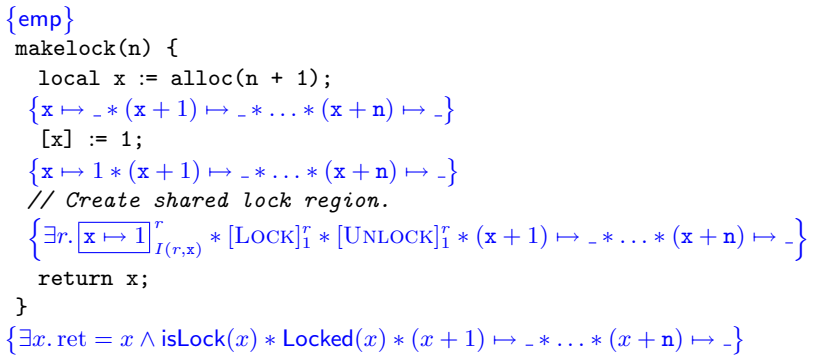


Fig. 2. Verifying the compare-and-swap lock implementation: `makelock`.

program C is run from a state satisfying P then it will not fault, but will either terminate in a state satisfying Q or not terminate at all.

The proof rule for atomic commands is

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

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 triple in sequential separation logic, where p, q denote separation logic assertions that do not specify predicates, shared regions or interference.

The second premise, $\Delta \vdash P \Longrightarrow^{\{p\}\{q\}} Q$, says that the interference allowed by P enables the state to be repartitioned to Q, given the change to memory specified by $\{p\}\{q\}$. In our example, 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. Pre-condition and post-condition stability is a general requirement that our proof rules have, which for presentation purposes we keep implicit in the rest of the paper.

The repartitioning arrow $P \Longrightarrow Q$ 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 repartitioning in the rule of consequence to move resources between regions. The operator $\Longrightarrow$ includes conventional implication, so this rule of consequence subsumes the traditional one.

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

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 concurrent algorithms [23], 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.

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

This rule defines a module consisting of functions $f_1 \ldots f_n$ and uses it to verify a client specification $\{P\}C\{Q\}$.

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

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

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

2.4 Example: A Ticketed Lock

We now consider another, more complex lock algorithm: a lock that issues tickets to clients contending for the lock. This algorithm is used in current versions of Linux, 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:

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

```
unlock(x) {
 ⟨x.owner++;⟩
}
```

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

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

The implementation assumes an atomic operation INCR that increments a stored value and returns the original value. To acquire the lock, a client atomically increments x.next and reads it to a variable i. The value of i becomes the client's ticket. The client waits for x.owner to equal its ticket value i. Once this is the case, the client 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:

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

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

($\circledast$ is the lifting of $*$ to sets; it is the multiplicative analogue 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 the 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 its definition is invariant under the action Take.

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

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

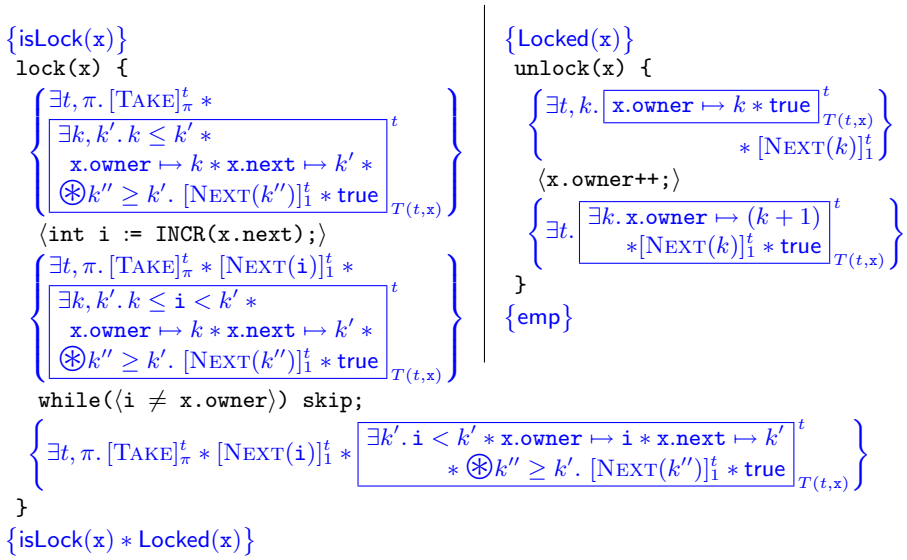


Fig. 3. Proofs for the ticketed lock module operations: lock and unlock.

3 Composing Abstract Specifications

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

3.1 A Set Module Specification

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

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

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 a shorthand for the disjunction of these two predicates.

These assertions 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.

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

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

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

Fig. 4. Proof outline for the set module client.

3.2 Example: The Coarse-grained Set

Consider a coarse-grained set implementation, based on the lock module described in §2.1 and the sequential set operations $\texttt{scontains}(\texttt{h}, \texttt{v})$, $\texttt{sadd}(\texttt{h}, \texttt{v})$ and $\texttt{sremove}(\texttt{h}, \texttt{v})$.

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

```
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 coarse-grained 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:

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

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

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

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

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

$$\mathsf{allgaps}(s) \equiv \circledast ws.\, [\textsc{SGap}(ws)]^s_1$$

$$P_\triangleleft(h, v, s) \equiv \exists vs.\ v \triangleleft vs \wedge \begin{pmatrix} (\mathsf{allgaps}(s) * \mathsf{Set}(h.\mathrm{set}, vs)) \\ \vee\ \mathsf{Locked}(h.\mathrm{lock}) * ([\textsc{SGap}(vs)]^s_1 \mathrel{-\!\!\circledast} \mathsf{allgaps}(s)) \end{pmatrix}$$
where $\triangleleft = \in$ or $\triangleleft = \notin$

$\mathsf{allgaps}$ defines the set of all $\textsc{SGap}$ permissions. $P_\in(h, v, s)$ is used to assert that the shared state contains either the set with contents vs, where $v \in vs$, and all possible $\textsc{SGap}$ 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 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:

$$\mathsf{in}(h, v) \equiv \exists s.\, \mathsf{isLock}(h.\mathrm{lock}) * [\textsc{SChange}(v)]^s_1 * \boxed{P_\in(h, v, s)}^s_{C(s,h)}$$

$$\mathsf{out}(h, v) \equiv \exists s.\, \mathsf{isLock}(h.\mathrm{lock}) * [\textsc{SChange}(v)]^s_1 * \boxed{P_\notin(h, v, s)}^s_{C(s,h)}$$

The $\mathsf{in}(h, v)$ predicate gives a thread the permissions needed to acquire the lock, $\mathsf{isLock}(h.\mathrm{lock})$, and to change whether v is in the set, $[\textsc{SChange}(v)]^s_1$. 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 $\notin$ in place of $\in$.

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

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 `add(h, v)` when the value is not in the set. The case where the value is in the set, and the proofs of `remove` and `contains` follow a similar structure.

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

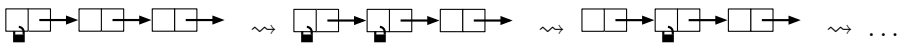
$\{\mathsf{out}(\mathtt{h},\mathtt{v})\}$
`add(h,v)`
$\{\exists s.\, \mathsf{isLock}(\mathtt{h.lock}) * [\textsc{SChange}(\mathtt{v})]^s_1 * \boxed{P_{\notin}(\mathtt{h},\mathtt{v},s)}^{s}_{C(s,\mathtt{h})}\}$
`lock(h.lock);`
$\{\exists s.\, \mathsf{isLock}(\mathtt{h.lock}) * \mathsf{Locked}(\mathtt{h.lock})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

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

3.3 Example: 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 [13, §9.5]) is given in Fig. 6. We omit `contains` for space reasons.

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

Proof sketch We can verify this algorithm with our logic. The details are given in the technical report [5]. Here, we just present the intuition behind the proof.

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

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

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

Fig. 6. Lock-coupling list algorithm.

As with the course-grained example, we have actions LCHANGE and LGAP, parameterised by value v. An LCHANGE permission allows a thread to take locked parts of the list out of the shared state into its local state, acquiring LGAP permissions and giving up the appropriate Locked predicates. These LGAP permissions allow the thread to return the parts of the list it acquired, having possibly inserted or removed a node with value v, and to regain the Locked predicates. We can then give the definition for $\mathsf{in}(h, v)$, or for $\mathsf{out}(h, v)$, as shared regions where the list starting at h contains, or does not contain, the value v, respectively. Both predicates include the full permission for LCHANGE on v. The list definition must track gaps for the currently locked segments. These gaps correspond to missing LGAP permissions.

4 Semantics and Soundness

We present the model for interpreting our assertions and program judgements, and sketch a proof of soundness of our logic. Details of the proof are given in [5].

4.1 Assertion Syntax

Recall from §2.3 that our proof judgements have the form $\Delta, \Gamma \vdash \{P\}\, C\, \{Q\}$. Here, P and Q are *assertions* in 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 Triples. The syntax is defined as follows:

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

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

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

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

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

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 [4] representing *machine states* (or *heaps*). Typically, we take Heap to be the standard heap model: the set of finite partial functions from locations to values, where $\uplus$ is the union of partial functions with disjoint domains. 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 to an appropriate region; we defer the formal definition of well-formedness:

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

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 $[\text{UNLOCK}]^r_1$ from local to shared state. Our semantics breaks the loop by distinguishing between the syntactic representation

of an action and its semantics. Actions are represented syntactically by *tokens*, consisting of the region name, the action name and a sequence of value parameters (*e.g.* the permission $[\text{SCHANGE}(v)]^s_1$ pertains to the token $(s, \text{SCHANGE}, v)$):

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

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.[6]

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 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 assignment (which assigns zero permission value 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.

We define the operator $\lfloor (l, s) \rfloor$ which collapses a pair of a logical state l and shared state s to a single logical state, and the operator $\lVert (l, s) \rVert$ which gives the heap component of $\lfloor (l, s) \rfloor$; we use $\bigoplus$, the natural lifting of $\oplus$ to sets:

$$\lfloor (l, s) \rfloor \stackrel{\text{def}}{=} l \oplus \left(\bigoplus\nolimits_{r \in \text{dom}(s)} s(r) \right) \qquad \lVert (l, s) \rVert \stackrel{\text{def}}{=} (\lfloor (l, s) \rfloor)_{\text{H}}$$

The *action domain* of an interference assertion, $\text{adom}(I)$, is the set of action names and their appropriate parameters:

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

We are finally in a position to define well-formedness of worlds, $\mathsf{wf}(l, s)$:

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

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

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

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

Fig. 7. Semantics of assertions. The cases for conjunction, implication, existential, etc. are standard, simply distributing over the local and shared state.

4.3 Assertion Semantics

Fig. 7 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, and just 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 parameterised by a *predicate environment*, δ, mapping abstract predicates to their semantic definitions, and an *interpretation*, i, mapping logical variables to values:

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

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

The cell assertion $\mapsto$, the separating star $*$ and the existential separating implication $-\!\!\circledast$ are standard. The quantifier $\circledast$ is the iterated version of $*$; that is, the finite, multiplicative analogue of $\forall$. The empty assertion emp asserts that the local state and permission assignment 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 given by the predicate environment, δ.

The permission assertion $[\gamma(E_1, \ldots)]^r_\pi$ states that the token $(\llbracket r \rrbracket_i, \gamma, \llbracket E_1 \rrbracket_i \ldots)$ is associated with permission value $\llbracket \pi \rrbracket_i$.

A shared-state assertion $\boxed{P}^r_I$ asserts that P holds for region $\llbracket r \rrbracket_i$ in the shared state, and that the region's interference is given by the interference assertion, $\llbracket I \rrbracket_i$. 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 $\llbracket I \rrbracket_i$ 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 the 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 defines a map from tokens to sets of shared-state pairs (what we call an *interference environment*):

$$\llbracket - \rrbracket_- : \mathsf{IAssn} \times \mathsf{PEnv} \to \mathsf{Token} \to \mathcal{P}(\mathsf{SState} \times \mathsf{SState})$$

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.

$$\llbracket I_1, I_2 \rrbracket_\delta(r, \gamma, \vec{v}) \stackrel{\text{def}}{=} \llbracket I_1 \rrbracket_\delta(r, \gamma, \vec{v}) \cup \llbracket I_2 \rrbracket_\delta(r, \gamma, \vec{v})$$

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

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 defined in I.

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

The guarantee G_δ describes which updates, from world w to w', the thread is allowed to perform. A thread can update its local state as it pleases, but any change to a shared region, r, must correspond to an action, $\gamma(\vec{v})$, for which the thread has sufficient permission, $(w_{\mathrm{L}})_{\mathrm{P}}(r, \gamma, \vec{v}) > 0$. For example, in the CAS lock proof (§2.2), the thread must hold permission 1 on Unlock before unlocking. Without this restriction, other threads could potentially unlock the lock.

It is important that each update preserves the total amount of permission in the world, that is, $\lfloor w \rfloor_P = \lfloor w' \rfloor_P$, so that threads cannot acquire permissions out of thin air. This does *not* hold for heaps, as we permit memory allocation.

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. $(G^{\mathrm{c}})^{-1}$).

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

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

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

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

We now define the notion of *repartitioning* with respect to an update from p to q, written $P \Longrightarrow_\delta^{\{p\}\{q\}} Q$. This asserts any world w_1 satisfying P can be collapsed to a heap $\lVert w_1 \rVert$ that has a subheap h_1 satisfying the separation logic assertion p; furthermore, when this subheap is replaced with any subheap h_2 that satisfies q, the resulting heap can be reconstructed into a world w_2 that satisfies Q and for which the transition from w_1 to w_2 is permitted by the guarantee, $\widehat{G}_\delta$. The guarantee limits the repartitioning that takes place between the regions.

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

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

Note that, if $p = q = \mathsf{emp}$, then the repartitioning preserves the concrete state, and only allows the world to be repartitioned. We write $P \Longrightarrow_\delta Q$ as a shorthand for $P \Longrightarrow_\delta^{\{\mathsf{emp}\}\{\mathsf{emp}\}} Q$. Recall from the proof of the compare-and-swap lock implementation (§2.2) that repartitioning 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* $X \in \mathrm{ran}(\delta)$, *for all* w *and* w', *if* $w \in X$ *and* $(w, w') \in R_\delta$, *then* $w' \in X$.

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

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

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:

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

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

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

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

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

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

Fig. 8. Selected proof rules.

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

Judgements about such programs have the form $\Delta; \Gamma \vdash \{P\}\, C\, \{Q\}$. 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. 8. These rules are modified from RGSep [23] and deny-guarantee [8]. 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 Atomic and Prim is a judgement in standard sequential separation logic. The other minor judgements are defined semantically to quantify over all $\delta \in [\![\Delta]\!]$: $\Delta \vdash P \Longrightarrow^{\{p\}\{q\}} Q$ means $\forall \delta \in [\![\Delta]\!]\,.\, P \Longrightarrow_{\delta}^{\{p\}\{q\}} Q$ (and similarly without a superscript); $\Delta \vdash \mathsf{stable}(P)$ means $\forall \delta \in [\![\Delta]\!]\,.\, \mathsf{stable}_\delta(P)$; and $\Delta \vdash \Delta'$ means $[\![\Delta]\!] \subseteq [\![\Delta']\!]$.

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

of a proof rule could be vacuously true. We ensure this by requiring that all occurrences of the predicate in its definition are positive.

The PAR rule is the key rule for disjoint concurrency. In this rule we exploit our fiction of disjointness to prove safety for concurrent programs. Our set-up allows us to define a simple parallel rule while capturing fine-grained interactions. The ATOMIC and CONSEQ rule were discussed in §2.3. That section also discussed a rule for modules, which can be derived using PRED-I, LET and PRED-E.

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

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

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

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

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

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

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

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

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

Proof is by by structural induction. Most interesting is PAR, which embodies the compositionality of our logic. The proof requires the following lemma.

Lemma 1 (Abstract state locality).
If $(C, \llbracket w_1 \oplus w_2 \rrbracket) \xrightarrow{\eta} (C', h)$ *and* $C, w_1, \eta, \delta, i, Q$ $safe_n$*, then* $\exists w_1', w_2'$ *such that* $(C, \llbracket w_1 \rrbracket) \xrightarrow{\eta} (C', \llbracket w_1' \rrbracket)$*,* $h = \llbracket w_1' \oplus w_2' \rrbracket$*,* $(w_1, w_1') \in \widehat{G}_\delta$*, and* $(w_2, w_2') \in (R_\delta)^*$*.*

Proof. We require that basic commands obey a concrete locality assumption. We must prove that the rely and guarantee obey similar locality lemmas. The lemma then follows from the definition of configuration safety. □

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

5 Conclusions and Related Work

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

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

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

An alternative to using invariants is to abstract interference over the shared state by relations modelling the interaction of different threads: the rely-guarantee method [18]. There have been two recent logics that combine RG with separation logic: RGSep [23] and SAGL [10]. Both allow more elegant proofs of concurrent algorithms than the invariant-based approaches, but they have the drawback

that interference on the shared state cannot be abstracted. Feng's Local Rely-Guarantee [9] improves the locality of RGSep and SAGL by scoping interference with a precise footprint, but this approach still has no support for abstraction. Many of our ideas on abstraction originated in Dinsdale-Young, Gardner and Wheelhouse's work on using RGSep to analyse a concurrent B-tree algorithm [6, 22].

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

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

Proofs in our proof system are slightly more complex in practice than RGSep and SAGL, as can be seen by comparing the lock-coupling list proof with the RGSep one [23]. This may be the penalty that we pay for having greater modularity although, as we acquire more experience with doing proofs using concurrent abstract predicates, we expect to be able to reduce this complexity.

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

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

References

[1] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320, 2007.
[2] J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.
[3] C. Calcagno, P. Gardner, and U. Zarfaty. Local reasoning about data update. *Festschrift* Computation, Meaning and Logic: Articles dedicated to Gordon Plotkin, 172, 2007.

[4] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Symp. on Logic in Comp. Sci. (LICS'07)*, pages 366–378, 2007.
[5] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. Technical Report 777, University of Cambridge Computer Laboratory, 2010.
[6] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Local reasoning about a concurrent B*-list algorithm. Talk and unpublished report, see `http://www.doc.ic.ac.uk/~pg/`, 2009.
[7] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Locality refinement. Technical Report DTR10-8, Imperial College London, 2010.
[8] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.
[9] X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.
[10] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.
[11] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.
[12] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's Reentrant Locks. In *APLAS*, pages 171–187, 2008.
[13] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, 2008.
[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.
[15] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.
[16] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, Jan. 2001.
[17] B. Jacobs and F. Piessens. Modular full functional specification and verification of lock-free data structures. Technical Report CW 551, Katholieke Universiteit Leuven, Department of Computer Science, June 2009.
[18] C. B. Jones. Annoted bibliography on rely/guarantee conditions. `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf`, 2007.
[19] P. W. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 2007.
[20] M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, pages 297–302, Jan. 2007.
[21] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.
[22] P. Pinto. Reasoning about B^{Link} trees. Advanced masters ISO project, Imperial College London, 2010. Supervised by Dinsdale-Young, Gardner and Wheelhouse.
[23] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.