Modular Reasoning for Deterministic Parallelism

Mike Dodds
University of Cambridge, UK
md466@cl.cam.ac.uk

Suresh Jagannathan
Purdue University, Indiana
(work done while on sabbatical at Cambridge)
suresh@cs.purdue.edu

Matthew J. Parkinson
Microsoft Research Cambridge, UK
mattpark@microsoft.com

Abstract

Weaving a concurrency control protocol into a program is difficult and error-prone. One way to alleviate this burden is *deterministic parallelism*. In this well-studied approach to parallelisation, a sequential program is annotated with sections that can execute concurrently, with automatically injected control constructs used to ensure observable behaviour consistent with the original program.

This paper examines the formal specification and verification of these constructs. Our high-level specification defines the conditions necessary for correct execution; these conditions reflect program dependencies necessary to ensure deterministic behaviour. We connect the high-level specification used by clients of the library with the low-level library implementation, to prove that a client's requirements for determinism are enforced. Significantly, we can reason about program and library correctness without breaking abstraction boundaries.

To achieve this, we use *concurrent abstract predicates*, based on separation logic, to encapsulate racy behaviour in the library's implementation. To allow generic specifications of libraries that can be instantiated by client programs, we extend the logic with higher-order parameters and quantification. We show that our high-level specification abstracts the details of deterministic parallelism by verifying two different low-level implementations of the library.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification—Correctness proofs; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Languages, Theory, Verification

Keywords Separation Logic, Concurrent Abstract Predicates, Concurrency, Futures

1. Introduction

Writing safe and efficient concurrent programs is challenging because it requires programmers not only to parcel useful units of work into threads that can be executed in parallel, but also to weave suitable concurrency control to coordinate the access of these threads to shared data. To enable effective reasoning about concurrent programs, however, it is essential to devise *modular* abstractions whose implementations can be hidden behind well-defined interfaces, allowing clients to reason about correctness in terms of abstract, rather than concrete, behaviour.

In this paper, we consider the verification of one such concurrency construct: barriers. In *deterministic* parallelization, code regions in a sequential program are executed concurrently. While the parallelized program is internally nondeterministic, control constructs are used to ensure that it exhibits the same deterministic observable behaviour as its sequential counterpart. Automatic parallelization of this kind has been well-studied for loop-intensive numerical computations. However, it is also possible to extract parallelism from irregularly structured sequential programs, where program dependencies are not readily apparent [4, 22, 25].

One way to achieve deterministic parallelism is through compiler-injected barriers [19]. We can think of these barriers as resource management operations that enforce the original sequential order (aka program dependencies). A resource could be any program variable, data structure, memory region, lock, etc. for which ownership guarantees are essential in order to enforce deterministic semantics. We assume barrier implementations are provided as part of a library.

While the intuition behind using such barriers is quite simple, there are many possible implementations. Verifying that an implementation adheres to this intuition is challenging for several reasons.

First, the patterns of signalling in a barrier implementation are highly non-local. To access a resource, a barrier must wait until all logically preceding threads have indicated that it is safe to do so; these logically preceding threads represent sources in a dependency graph. This abstract view of resource-transfer does not fit with the structure of a highly concurrent implementation, making it difficult to avoid breaking abstraction boundaries.

Furthermore, compiler optimizations might strive to identify the earliest point in a thread's execution path from where a resource is no longer required. In some cases, this means threads can release resources without ever acquiring them, so that subsequent signalling of this resource by its ancestors to its descendants can bypass it altogether. An ancestor of a thread is a computation that logically precedes it under sequential execution, and a descendent is a computation that logically follows it. Implementations of barriers must allow a thread to renounce the acquisition of a resource in this way.

Finally, barriers may have to treat reads and writes differently to ensure preservation of sequential behaviour. Although many reads can be performed concurrently, they must be sequentialized with respect to writes. Moreover, reads must be sequentialized with respect to other reads, if there is an intervening write in the sequential order.

In this paper, we show how to reason in a modular way about implementations of such barriers. To do this, we use concurrent abstract predicates [6], a technique based on separation logic that enables abstract reasoning about concurrent modules. Our logic

POPL'11, January 26–28, 2011, Austin, Texas, USA.

allows us to reason about both high-level behavioural properties and low-level implementation details. This approach allows fine-grained reasoning about behaviour, meaning that each thread can be given access to exactly the behaviour it needs to run according to the abstract specification. This behavioural reasoning is *local*, meaning even non-local descriptions of the shared state can be encapsulated and abstracted.

By leveraging concurrent abstract predicates in this way, we take a first step towards the formal specification and verification of a system for deterministic parallelism. While full verification of compiler analyses, transformations, and library implementations is our ultimate goal, we focus here on just the verification problem for libraries. We present a high-level specification for reasoning about barriers for deterministic parallelism, independent of their low-level implementation. We prove that two low-level implementations of these barriers implement our high-level specification.

In the presence of runtime thread creation and dynamic (heap-allocated) data, our specification must also be both generic and dynamic, in the sense that it must be able to construct signals at runtime that protect arbitrary resources. To support the transfer of arbitrary resources between threads, we have extended the concurrent abstract predicates approach to support higher-order predicate parameters, and higher-order quantification. The controlled resources are represented by propositional arguments to abstract predicates.

We make the following contributions:

1. We develop a high-level abstract specification for reasoning about libraries that implement barriers used to enforce deterministic parallelism. This specification can express complex behaviours such as the dynamic construction of new barriers and out-of-order signalling between threads.
2. We provide proofs that two implementations of such barriers satisfy our high-level specification. The first implementation naïvely sequentializes signalling, while the second aggregates information from logically earlier threads to avoid this bottleneck.
3. We extend prior work on concurrent abstract predicates to support higher-order parameters and quantifications, following higher-order separation logic [3]. By allowing propositional parameters, we can define predicates that take invariants as arguments, to enable abstract reasoning about resource transfer.

An extended version of this paper containing full proofs is available as a technical report [9].

2. A Specification for Deterministic Parallelism

In this section, we describe the behaviour of a library providing barriers for enforcing deterministic parallelism. We define a high-level specifications for these barriers, which allow us to prove that programs parallelised using these barriers preserve sequential behaviour.

We assume that code sections believed to be amenable for parallelization have been identified, and the program split accordingly into threads. We assume a total logical ordering on threads, such that executing the threads serially in the logical order gives the same result as the original (unparallelised) program.

Barriers are associated with resources (e.g., program variables, data structures, etc.) that are to be shared between concurrently-executing program segments. There are two sorts of barriers. A *grant* barrier notifies logically later threads that the current thread will no longer use the resource. A *wait* barrier blocks until all logically prior threads have signalled that they will no longer use the resource (i.e., have issued grants). We assume barriers have been appropriately injected by a compiler to ensure that all salient data dependencies in the original program are respected.

Consider the following function `f`; here `*` corresponds to non-deterministic choice, so `sleep(*)` waits for an arbitrary period of time:

```
f(x,y,v) {
  if(x<10) {
    y:=y+v; x:=x+v;
  } else { sleep(*); }
}
```

Suppose now that we run two instances of `f` in sequence:

```
x:=0; y:=0; f(x,y,5); f(x,y,11);
```

When this program terminates, location `x` and `y` will both hold 16.

Here, the second call to `f` may have to wait for the first call to finish its arbitrarily long `sleep`, even though the first call will do nothing more once it wakes. We parallelise this function by constructing two new functions `f1` and `f2`. We run both concurrently, but require that `f1` passes control of `x` and `y` to `f2` *before* sleeping, allowing `f2` to continue executing.

```
f1(x,y,v,i) {
  if(x<10) {
    y:=y+v; x:=x+v;
    grant(i);
  } else {
    grant(i);
    sleep(*);
} }
```

```
f2(x,y,v,i) {
  wait(i);
  if(x<10) {
    y:=y+v; x:=x+v;
  } else {
    sleep(*);
} }
```

```
x:=0; y:=0; i:=newchan(); f1(x,y,5,i)||f2(x,y,11,i);
```

The barriers in `f1` and `f2` ensure that the two threads wait exactly until the resources they require can be safely modified, without violating sequential program dependencies. The correct ordering is enforced by barriers that communicate through a channel; in the example, `newchan` creates the channel `i`. Assuming the barriers are correctly implemented, the resulting behaviour is equivalent to that of the original sequential program.

2.1 Verifying a Client Program

How can we verify that our parallelised program based on `f1` and `f2` has the same specification as the original sequential program? Typically, one would incorporate signalling machinery as part of a parallelization program analysis. Clients would then reason about program behaviour using the operational semantics of the barrier implementation. Validating the correctness of parallelisation with respect to the sequential program semantics would therefore require a detailed knowledge of the barrier implementation. Any changes to the implementation could entail reproving the correctness of the parallelisation analysis.

In contrast, we reason about program behaviour in terms of abstract specifications for `grant`, `wait` and `newchan`. Such an approach has the following advantages: (1) Implementors can modify their underlying implementation and be sure that relevant program properties are preserved by the implementation, and (2) client proofs (in this case, proofs involving compiler correctness) can be completed without knowledge of the underlying implementation.

We will reason about `f1` and `f2` using separation logic. We write the following assertion to denote that `x` points to value v and `y` to value v', and that `x` and `y` are distinct: $\texttt{x} \mapsto v * \texttt{y} \mapsto v'$. To reason about the parallel composition of threads, we use the PAR rule of concurrent separation logic [20]:

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\ \textsc{Par}$$

Now, to reason about `f1` and `f2`, we must be able to encode the fact that `f1` can give up access to `x` and `y` by calling `grant(i)`,

while f2 can retrieve access to them by calling wait(i). To use the parallel rule, we must be able to give the two threads star-separated preconditions.

We encode these two facts by defining two predicates, fut and req, corresponding to the *future resource*, the resource that can be acquired from logically earlier threads, and the *required resource*, the resource that must be supplied to logically later threads. We read these as follows:

$\mathsf{fut}(i, P)$ – By calling wait on i, the thread will acquire a resource satisfying the assertion P.

$\mathsf{req}(i, P)$ – By calling grant on i when holding a resource satisfying P, the thread will lose the resource P.

These predicates are *abstract*; each instantiation of the library will define them differently; the client program knows nothing about how they are actually defined. The client only depends on an abstract specification that captures the intuitive meaning of the predicates:

$$\begin{array}{ccc} \{\mathsf{emp}\} & i := \texttt{newchan()} & \{\mathsf{req}(i,P) * \mathsf{fut}(i,P)\} \\ \{\mathsf{fut}(i,P)\} & \texttt{wait}(i) & \{P\} \\ \{\mathsf{req}(i,P) * P\} & \texttt{grant}(i) & \{\mathsf{emp}\} \end{array}$$

(Note that this is a weaker version of our full specification, given in Fig. 2.)

The specification of newchan is noteworthy. This specification is implicitly universally quantified for all assertions P, meaning that we can construct a predicate for any assertion.[1] New fut and req predicates can be constructed at run-time using newchan, meaning we can construct an arbitrarily large number of channels for use in the program.

Given these two predicates, we can define the following specifications for f1 and f2.

$$\left\{\begin{array}{l} v_1 < 10 \wedge x \mapsto v_1 * y \mapsto v_2 \\ \quad * \mathsf{req}\left(i, \begin{array}{l} x \mapsto (v_1{+}v) * \\ y \mapsto (v_2{+}v) \end{array}\right) \end{array}\right\} \ \texttt{f1}(x,y,v,i) \ \{\mathsf{emp}\}$$

$$\left\{\begin{array}{l} v_3 < 10 \wedge \\ \mathsf{fut}(i, x \mapsto v_3 * y \mapsto v_4) \end{array}\right\} \ \texttt{f2}(x,y,v,i) \ \left\{\begin{array}{l} x \mapsto (v_3{+}v) \\ * \, y \mapsto (v_4{+}v) \end{array}\right\}$$

The specification for f1 says that the thread must supply the req predicate with the resources x and y such that the value in x is less than 10. The specification for f2 says that the thread can receive x and y with the value in x less than 10. Fig. 1 gives sketch-proofs for these two specifications.

Given this specification, the proof for the main program goes through as follows:

$\{\mathsf{x} \mapsto \_ * \mathsf{y} \mapsto \_\}$
x:=0; y:=0; i:=newchan();
$\{\mathsf{x} \mapsto 0 * \mathsf{y} \mapsto 0 * \mathsf{req}(i, x \mapsto 5 * y \mapsto 5) * \mathsf{fut}(i, x \mapsto 5 * y \mapsto 5)\}$
f1(x,y,5,i) || f2(x,y,11,i) // *Parallel rule.*
$\{x \mapsto 16 * y \mapsto 16\}$

This proof establishes that the post-condition for the parallelised version of the program is identical to the post-condition for the original sequential version.

[1] In the full specification, we impose an extra requirement that P is *stable*, meaning invariant under concurrent interference, but this holds trivially for unshared assertions such as $x \mapsto v * y \mapsto v'$.

$\left\{\begin{array}{l} v_1 < 10 \wedge x \mapsto v_1 * y \mapsto v_2 * \\ \mathsf{req}\,(i, x \mapsto (v_1{+}v) * y \mapsto (v_2{+}v)) \end{array}\right\}$
if(x<10) {
  y:=y+v; x:=x+v;
  $\left\{\begin{array}{l} x \mapsto (v_1{+}v) * y \mapsto (v_2{+}v) * \\ \mathsf{req}\,(i, x \mapsto (v_1{+}v) * y \mapsto (v_2{+}v)) \end{array}\right\}$
  grant(i); // *Abstract spec.*
  $\{\mathsf{emp}\}$
} else ... // *Contradiction as* $v_1 < 10$.
$\{\mathsf{emp}\}$

$\{v_3 < 10 \wedge \mathsf{fut}\,(i, x \mapsto v_3 * y \mapsto v_4)\}$
if(x<10) {
  wait(i); // *Abstract spec.*
  $\{v_1 < 10 \wedge x \mapsto v_3 * y \mapsto v_4\}$
  y:=y+v; x:=x+v;
  $\{x \mapsto (v_3{+}v) * y \mapsto (v_4{+}v)\}$
} else ... // *Contradiction as* $v_3 < 10$.
$\{x \mapsto (v_3{+}v) * y \mapsto (v_4{+}v)\}$

Figure 1. Proofs for f1 and f2.

2.2 Generalising to Many Threads

Suppose we want to run many copies of the function f in sequence, for example over an array of values vs. We might have the following sequential program:

```
for(j:=0; j<max; j++){ f(x,y,vs[j]); }
```

To parallelise this program, we want each call to f to run in a separate thread. To do this, f must be modified to contain calls to *both* grant and wait. Intuitively, each call to f receives the resource from logically earlier threads (those invoked in earlier loop iterations) with wait, then releases it to logically later threads (those invoked in later loop iterations) using grant.

To allow many threads to access the same resource in sequence, we can construct a *chain* of channels. A wait barrier called on a channel waits for grant barriers on *all* preceding channels. We use the ordering in chains of channels to model the logical ordering between a sequence of parallelised threads.

A chain initially consists of a singleton channel constructed using newchan. We introduce an operation split that allows us to insert a new channel into the chain. The specification of split takes a req predicate for an existing channel and creates a new fut and req predicate representing the new channel. The new channel is inserted into the chain immediately before the existing channel. We extend the req predicate with an additional argument identifying the preceding channel in the chain. The split operation's specification is given in Fig. 2.

There are two more potential sources of parallelism in f. First, in the original transformation involving f1 and f2, we did not distinguish between the resources x and y. However, we need to gain access to y only if we take the first branch of the conditional. Otherwise we can release y to logically future threads. To realise this parallelism in the new version of f, we use two chains of channels: one for x, and one for y.

Second, we can exploit the ability to *renounce* access to a resource without acquiring it first. In the simple specification given

SPECS:

$$\{\mathsf{fut}(i,P)\} \quad \texttt{wait}(i) \quad \{P\}$$

$$\left\{\begin{array}{l}\mathsf{req}(i,i',P)\ *\\ P \vee \left(\begin{array}{l}\mathsf{fut}(i',P')\\ *\ (P' \mathrel{-\!\!*} P)\end{array}\right)\end{array}\right\} \quad \texttt{grant}(i) \quad \{\mathsf{emp}\}$$

$$\left\{\begin{array}{l}\mathsf{req}(i,i',P)\\ *\ \mathsf{stable}(Q)\end{array}\right\} \quad j,j' := \texttt{split}(i) \quad \left\{\begin{array}{l}\mathsf{req}(j,j',P)\\ \quad *\ \mathsf{fut}(j',Q)\\ *\ \mathsf{req}(j',i',Q)\end{array}\right\}$$

$$\{\mathsf{stable}(P)\} \quad i := \texttt{newchan}() \quad \left\{\begin{array}{l}\mathsf{fut}(i,P)\\ *\ \mathsf{req}(i,\mathsf{nil},P)\end{array}\right\}$$

AXIOMS:

$$\left(\begin{array}{l}\mathsf{fut}(i,P) * (P \mathrel{-\!\!*} (P_1 * P_2))\\ *\ \mathsf{stable}(P_1) * \mathsf{stable}(P_2)\end{array}\right) \implies \mathsf{fut}(i,P_1) * \mathsf{fut}(i,P_2)$$

Figure 2. Full abstract specification for deterministic parallelism.

above, we can only call `grant` if we hold the required resource. However, this is often not necessary. For example, if we take the second branch of the conditional in `f`, we do not need the resource `y`. It is safe to notify future threads that `y` is available, conditional on all logically prior threads releasing it, even though the thread itself never acquired access to the resource.

Renunciation can be a powerful technique for parallelisation. Suppose a thread is logically last in a chain of threads accessing a resource. Suppose the thread takes an execution path rendering it unnecessary to ever access the resource. Without renunciation, a call to `grant` will block until all earlier threads have finished with the resource. With renunciation, the thread can pass the barrier and continue executing, irrespective of the status of logically earlier threads.

To support renunciation, we modify the specification for `grant` (see Fig. 2). This new specification allows a thread to discharge a req using the preceding fut predicate. In other words, the thread gives up the ability to ever acquire the resource, and instead forwards this capability to future threads. When the resource becomes available from logically prior threads, the *next* thread in the logical order will receive it. The assertion $(P' \mathrel{-\!\!*} P)$ is used to convert the state supplied by the future to the state required by the next thread.[2]

In Fig. 2, we also add an *axiom* to our specification. This is a fact about the library predicates that clients of the library can make use of. The axiom allows resource splitting. This axiom asserts that when a thread can receive a resource P using identifier i, access to that resource can be split between two threads, potentially before the resource is available. The assertion $(P \mathrel{-\!\!*} (P_1 * P_2))$ asserts that P can be split into P_1 and P_2.

We now define `fp`, top of Fig. 3, a version of `f` which is safe to run in parallel with many copies of itself. This function takes arguments `ix` and `iy` representing the next points in the two channel sequences, and `ixp` and `iyp` representing the immediately prior points. We verify `fp` against the following specification:

$$\left\{\begin{array}{l}\mathsf{req}(\mathtt{ix},\mathtt{ixp},\mathtt{x} \mapsto \_) * \mathsf{fut}(\mathtt{ixp},\mathtt{x} \mapsto \_) *\\ \quad \mathsf{req}(\mathtt{iy},\mathtt{iyp},\mathtt{y} \mapsto \_) * \mathsf{fut}(\mathtt{iyp},\mathtt{y} \mapsto \_)\end{array}\right\} \quad \texttt{fp(...)} \quad \left\{\mathsf{emp}\right\}$$

A proof of this specification is given in Fig. 4. Note that we only assert basic memory safety in this specification. We could verify

[2] A resource satisfies $P' \mathrel{-\!\!*} P$ iff its combination with any disjoint resource satisfying P' produces a resource satisfying P.

```
fp(x,y,v,ix,iy,ixp,iyp) {
  wait(ixp);
  if (x<10) {
    wait(iyp);
    y:=y+v; grant(iy);
    x:=x+v; grant(ix);
  } else {
    grant(ix); grant(iy);
    sleep(*);
} }

ixf:=newchan(); iyf:=newchan();
for(j:=0; j<max; j++){
  v:=vs[j];
  ixl:=ixn; (ixf,ixn):=split(ixf);
  iyl:=iyn; (iyf,iyn):=split(iyf);
  future( fp(x,y,v,ixn,iyn,ixl,iyl) );
}
wait(ixf); wait(iyf);
```

Figure 3. Example parallelisation of `f` and a client. The `future` annotation marks the call to `fp` as a source of deterministic parallelism.

```
1  {req(ix,ixp,x ↦ _) * fut(ixp,x ↦ _) *
    req(iy,iyp,y ↦ _) * fut(iyp,y ↦ _)}
2  fp(x,y,v,ix,iy,ixp,iyp) {
3    wait(ixp);
4    {x ↦ _ * req(ix,ixp,x ↦ _) *
     req(iy,iyp,y ↦ _) * fut(iyp,y ↦ _)}
5    if (x<10) {
6      wait(iyp);
7      {x ↦ _ * y ↦ _ *
       req(ix,ixp,x ↦ _) * req(iy,iyp,y ↦ _)}
8      y:=y+v; grant(iy);
9      {x ↦ _ * req(ix,ixp,x ↦ _)}
10     x:=x+v; grant(ix);
11   } else {
12     grant(ix);
13     {req(iy,iyp,y ↦ _) * fut(iyp,y ↦ _)}
14     grant(iy);
15     {emp}
16     sleep(*);
17 } }
18 {emp}
```

Figure 4. Proof for parallelised program `fp`.

more complex properties by giving the fut and req predicates stronger invariants.

Line 14 of the proof is noteworthy. There, the precondition *does not* assert that the thread has access to $\mathtt{y} \mapsto \_$; rather, it asserts it can acquire access by calling `wait`. Instead of doing this, the thread renounces access to the resource, giving it up without ever having it.

```
g(x) {                    gp(x,r,rp,w,wp) {
  if(*) {                   if(*) {
    sleep(*);                 grant(r);
    read(x);                  sleep(*);
  }                           wait(rp);
  else {                      read(x);
    write(x);                 grant(w);
} }                         } else {
                              wait(rp); wait(wp);
                              write(x);
                              grant(r); grant(w);
                          } }
```

Figure 5. Example function g and its parallelisation.

The parallelised version of the main program is given at the bottom of Fig. 3. We give the following sketch-proof for this example. Here the predicates $\mathsf{req}(\mathtt{ixf}, \mathtt{ixn}, \mathsf{true})$ and $\mathsf{req}(\mathtt{iyf}, \mathtt{iyn}, \mathsf{true})$ are dummy req predicates used to represent the logically latest element of the sequential order. The predicate array stands for the array of values.

$\{\mathsf{array}(\mathtt{vs}, \mathtt{max})\}$
`ixf:=newchan(); iyf:=newchan();`
$\{\mathsf{array}(\mathtt{vs}, \mathtt{max}) * \mathsf{req}(\mathtt{ixf}, \mathsf{nil}, \mathsf{true}) * \mathsf{fut}(\mathtt{ixf}, x \mapsto \_) * \mathsf{req}(\mathtt{iyf}, \mathsf{nil}, \mathsf{true}) * \mathsf{fut}(\mathtt{iyf}, y \mapsto \_)\}$
`for(j:=0; j<max; j++){`
`  ixl:=ixn; (ixf,ixn):=split(ixf);`
$\{\mathsf{array}(\mathtt{vs}, \mathtt{max}) * \mathsf{req}(\mathtt{ixf}, \mathtt{ixn}, \mathsf{true}) * \mathsf{fut}(\mathtt{ixn}, x \mapsto \_) * \mathsf{req}(\mathtt{ixn}, \mathtt{ixl}, x \mapsto \_) * \mathsf{fut}(\mathtt{ixl}, x \mapsto \_) * \mathsf{req}(\mathtt{iyf}, \mathtt{iyn}, \mathsf{true}) * \mathsf{fut}(\mathtt{iyn}, y \mapsto \_)\}$
`  iyl:=iyn; (iyf,iyn):=split(iyf);`
$\{\mathsf{array}(\mathtt{vs}, \mathtt{max}) * \mathsf{req}(\mathtt{ixf}, \mathtt{ixn}, \mathsf{true}) * \mathsf{fut}(\mathtt{ixn}, x \mapsto \_) * \mathsf{req}(\mathtt{ixn}, \mathtt{ixl}, x \mapsto \_) * \mathsf{fut}(\mathtt{ixl}, x \mapsto \_) * \mathsf{req}(\mathtt{iyf}, \mathtt{iyn}, \mathsf{true}) * \mathsf{fut}(\mathtt{iyn}, y \mapsto \_) * \mathsf{req}(\mathtt{iyn}, \mathtt{iyl}, y \mapsto \_) * \mathsf{fut}(\mathtt{iyl}, y \mapsto \_)\}$
`  future( fp(x,y,vs[j],ixn,iyn,ixl,iyl) );`
`}`
`wait(ixf); wait(iyf); // GC dummy req predicates.`
$\{\mathsf{array}(\mathtt{vs}, \mathtt{max}) * x \mapsto \_ * y \mapsto \_\}$

We have shown that our parallelised version of the program is memory-safe. With a little more effort, we could verify the behaviour of the program. Crucially, even though this program features many threads running at once, with complex communication between threads, each individual thread is able to reason locally, without dealing with other threads or the implementation of the barriers.

2.3 Relating Reads and Writes

Further parallelism is available by refining read and write accesses to a resource. Consider the function g given in Fig. 5. It is safe for parallel threads to read x at the same time. However, it is important that writes to x are sequentialised, and that groups of reads are sequentialised *with respect to writes*. If two groups of reading threads are separated by a writing thread, the logically later group must wait for the writer to finish before reading.

To exploit this, we split reading and writing into two channels. We use r and rp for reads, and w and wp for writes. w and r are the outgoing channels, while wp and rp are the incoming channel. As soon as the thread nondeterministically takes the first branch of the

1 $\{\mathsf{fut}(\mathtt{rp}, \mathtt{x} \overset{\pi}{\mapsto} \_) * \mathsf{fut}(\mathtt{wp}, \mathtt{x} \overset{\pi'}{\mapsto} \_) \wedge \pi{+}\pi' = 1 * \mathsf{req}(\mathtt{r}, \mathtt{rp}, \mathtt{x} \overset{\frac{1}{2}\pi}{\mapsto} \_) * \mathsf{req}(\mathtt{w}, \mathtt{wp}, \mathtt{x} \overset{\pi'+\frac{1}{2}\pi}{\longmapsto} \_)\}$
2 `if(*) {`
3 `  // Apply the future splitting axiom to rp.`
4 $\{\mathsf{fut}(\mathtt{rp}, \mathtt{x} \overset{\frac{1}{2}\pi}{\mapsto} \_) * \mathsf{fut}(\mathtt{rp}, \mathtt{x} \overset{\frac{1}{2}\pi}{\mapsto} \_) * \mathsf{fut}(\mathtt{wp}, \mathtt{x} \overset{\pi'}{\mapsto} \_) * \mathsf{req}(\mathtt{r}, \mathtt{rp}, \mathtt{x} \overset{\frac{1}{2}\pi}{\mapsto} \_) * \mathsf{req}(\mathtt{w}, \mathtt{wp}, \mathtt{x} \overset{\pi'+\frac{1}{2}\pi}{\longmapsto} \_)\}$
5 `  grant(r);`
6 $\{\mathsf{fut}(\mathtt{rp}, \mathtt{x} \overset{\frac{1}{2}\pi}{\mapsto} \_) * \mathsf{fut}(\mathtt{wp}, \mathtt{x} \overset{\pi'}{\mapsto} \_) * \mathsf{req}(\mathtt{w}, \mathtt{wp}, \mathtt{x} \overset{\pi'+\frac{1}{2}\pi}{\longmapsto} \_)\}$
7 `  sleep(*); wait(rp);`
8 $\{\mathtt{x} \overset{\frac{1}{2}\pi}{\mapsto} \_ * \mathsf{fut}(\mathtt{wp}, \mathtt{x} \overset{\pi'}{\mapsto} \_) * \mathsf{req}(\mathtt{w}, \mathtt{wp}, \mathtt{x} \overset{\pi'+\frac{1}{2}\pi}{\longmapsto} \_)\}$
9 `  read(x); grant(w);`
10 $\{\mathsf{emp}\}$
11 `} else { ... }`
12 $\{\mathsf{emp}\}$

Figure 6. Proof for parallelised program gp.

conditional, it can use the read channel to signal that later threads can read. In contrast, a thread that wishes to write must wait for both the read and write channels. The parallelised program is given in Fig 5.

In separation logic, read and write access are often controlled by *fractional permissions* [5]. Each thread can hold either full permission, 1, on a location x, denoted $x \mapsto v$, or fractional permission $\pi \in (0..1)$, denoted $x \overset{\pi}{\mapsto} v$. Full permission gives the thread exclusive permission to write, while fractional permission gives non-exclusive permission to read. Fractional permissions compose by addition, as follows:

$$x \overset{\pi}{\mapsto} v * x \overset{\pi'}{\mapsto} v \iff x \overset{\pi+\pi'}{\mapsto} v \qquad \text{if } \pi+\pi' \leq 1.$$

We give the function gp the following specification:

$$\left\{\begin{array}{l}\mathsf{req}(\mathtt{r}, \mathtt{rp}, x \overset{\frac{1}{2}\pi}{\mapsto} \_) * \mathsf{fut}(\mathtt{rp}, x \overset{\pi}{\mapsto} \_) * \\ \mathsf{req}(\mathtt{w}, \mathtt{wp}, x \overset{\pi'+\frac{1}{2}\pi}{\longmapsto} \_) * \mathsf{fut}(\mathtt{wp}, x \overset{\pi'}{\mapsto} \_) \\ \wedge\, \pi{+}\pi' = 1\end{array}\right\} \ \mathtt{gp(..)} \ \{\mathsf{emp}\}$$

This specification says that when a thread receives a fractional permission from the read channel, only half of it has to be sent on to future threads using the read channel. The other half can be supplied on the write channel. This allows a thread to keep the ability to read, while notifying future threads that they also can read.

Fig. 6 shows a sketch-proof for the program. We elide the writing branch of the conditional as it is straightforward. The most notable proof step is line 3, where the specification's resource-splitting axiom is used to divide up access to the fut predicate. Half is used to discharge the req predicate r, allowing logically later threads to read, while half is used to allow the current thread to read. In this way, many threads can simultaneously have fractional access to the resource.

3. Verifying a Simple Implementation

So far, we have given an abstract specification for deterministic parallelism. The specification was independent of the implementation

```
grant(i) {
  if(i.prev!=nil)
    wait(i.prev);
  ⟨i.bit:=1⟩;
}

split(i) {
  n:=alloc(bit);
  n.prev:=i.prev;
  n.bit:=0;
  i.prev:=n;
  return (i,n);
}

wait(i) {
  while(i.bit!=1)
    skip;
}

newchan() {
  i:=alloc(bit);
  i.prev:=nil;
  i.bit:=0;
  return i;
}
```

Figure 7. Implementation of signalling library.

of the barrier. In this section, we show how such a specification can be justified by giving a simple implementation of the `wait` and `grant` barriers, and verifying our abstract specification against this concrete implementation.

The implementation is given in Fig. 7. This implementation supports resource transfer using a sequence of nodes, each of which has a `bit` field and a `prev` field. Each fut / req pair is associated with a single node, and the order of the sequence represents the logical ordering. The implementation requires that bits are set in sequential order. In §5 we consider a more sophisticated implementation that allows out-of-order signalling, and show that it also implements our abstract specification.

The `wait` barrier simply waits for the immediately preceding bit to be set. As bits are set in order, with logically earlier threads setting their bits before logically later ones, this suffices to show that all the earlier bits in the order have been set.

Recall from the previous section that our specification permits threads to renounce the ability to access a resource, meaning that `grant` can be called before `wait` within the same thread. To ensure that bits are set in sequential order, `grant` must wait for the previous bit to be set before setting its own bit. The implementation uses the `prev` field of the bit to call `wait`, and then sets its bit when it exits. Bits are set atomically by `grant`, denoted by $\langle - \rangle$.

The constructor functions `newchan` and `split` are implemented by allocating a new bit; `split` inserts a bit into the order by redirecting the `prev` pointer of the existing bit to point to the newly allocated bit. This allows computations to dynamically instantiate sub-computations that have internal deterministic parallelism.

3.1 Proof Approach

To prove the correctness of our module's functions, we use concurrent abstract predicates [6]. We extend this work with higher-order quantification, allowing us to prove specifications that abstract over the particular resource held by the predicate.

Concurrent abstract predicates extend standard separation logic with two new kinds of construct allowing explicit reasoning about sharing and interference. The first are *named shared regions*, denoted by boxed assertions of the form

$$\boxed{P}^r_I$$

This asserts that the region r contains a resource satisfying P, and nothing else. This region is shared between the current thread and an arbitrary number of other threads. The permitted state changes over the region are controlled by the *interference environment*, I.

The second are *capabilities*, resources controlling the updates that a thread can perform. In order to mutate the contents of a shared region, a thread requires a capability in its local state, denoted:

$$[\text{ACTION}]^r_\pi$$

This is a permission for the operation ACTION on the region r. The exact operation denoted by the name ACTION is determined by the interference environment for region r. Suppose that ACTION denoted the ability to rewrite the value in a shared address x from 0 to 1. Then we would have the following interference environment:

$$I(x) \quad \triangleq \quad (\text{ACTION}: \quad x \mapsto 0 \quad \rightsquigarrow \quad x \mapsto 1)$$

A capability $[\text{ACTION}]^r_\pi$ controls both whether the operation is permitted to the local thread, and whether it can be performed by the environment. Following deny-guarantee [8], exactly what is allowed and denied is determined by the permission level, π. We write 1 if the thread can exclusively perform the action, $\mathsf{g}z$ if the thread and the environment can perform the action, and $\mathsf{d}z$ if neither the thread nor the environment can perform the action. The value $z \in (0..1)$ is used to track the amount of permission, allowing capabilities to be split and combined.

Updates performed by a thread must be permitted according to the capabilities held by the thread in local state. So-called *abstract updates*, those that do not modify the underlying heap-state, can be performed at any time by a thread. We write $P \Longrightarrow Q$ to denote that P can be abstractly updated to give Q.

As assertions describe shared states that can be updated by other threads, we need to be able to describe assertions that will remain true no matter what the environment does. We describe these assertions as *stable*. Capabilities specify exactly what behaviours the environment can perform, giving fine-grained control of stability. For example, the following assertion is stable, because I specifies that the only way region r can be mutated is by the ACTION operations, and the exclusive capability to perform this operation is held by the thread in local state:

$$\boxed{x \mapsto 0}^r_{I(x)} * [\text{ACTION}]^r_1$$

Our logic includes an assertion $\mathsf{stable}(P)$ that holds if P is stable.

An abstract specification for a module consists of abstract predicates, function specifications and axioms. To show that a concrete implementation of a module corresponds to a particular abstract specification, we must supply concrete definitions for the module's abstract predicates, and then show that the following three properties hold: (1) the module implementation satisfies the abstract specifications, given the concrete predicate definitions; (2) the predicate definitions are stable; and (3) the axioms hold, given the concrete predicate definitions.

For simplicity, we assume that resources are garbage collected, rather than being explicitly deallocated. This means that we can safely remove star-conjuncts from assertions, and we often use this to clean up the post-conditions for operations.

3.2 Verifying the Implementation

Next, we prove that the implementation satisfies the abstract specification. We give definitions to the fut and req predicates in Fig. 8. (In all our predicate definitions, we assume unbound variables are existentially quantified.)

The definition of $\mathsf{req}(i, i', P)$ captures three pieces of information: First, that there exists a shared bit at address i. Second, that i' is the immediate predecessor of i, and it can be read by the thread. Third, that the thread must supply the resource P before setting the bit at i.

In this definition, we use two auxiliary predicates: pa and box. The predecessor access predicate $\mathsf{pa}(i)$ asserts that i is either nil, or it is a shared bit that can be read. This ensures that the thread that holds req is able to access the preceding bit. The predicate $\mathsf{box}(i, P, \pi)$ asserts that the thread can exchange the resource P for

$$\mathsf{req}(i, i', P) \triangleq \boxed{i.\mathrm{bit} \mapsto 0}^{r}_{I(i)} * \mathsf{box}(i, P, 1) * i.\mathrm{prev} \mapsto i' * \mathsf{pa}(i')$$

$$\mathsf{pa}(i) \triangleq i = \mathsf{nil} \vee \boxed{i.\mathrm{bit} \mapsto \_}^{r}_{I(i)}$$

$$\mathsf{box}(i, P, \pi) \triangleq [\mathrm{PUT}]^{r'}_1 * \boxed{\begin{array}{l} i.\mathrm{bit} \mapsto 0\ \boxed{}^{r}_{I(i)} * [\mathrm{SET}]^{r}_{\mathbf{d}\pi} \vee \\ \mathsf{box}(i, P_1, \pi_1) * \mathsf{box}(i, P_2, \pi_2) * \\ \quad P \mathrel{-\!\!*} (P_1 * P_2) \wedge \pi_1 + \pi_2 = \pi \end{array}}^{r'}_{J(i,P,\pi,r)}$$

$$\mathsf{fut}(i, P) \triangleq \mathsf{stable}(P) * [\mathrm{GET}]^{r'}_1 * \boxed{\begin{array}{l} [\mathrm{SET}]^{r}_{\mathbf{d}\pi} * \boxed{i.\mathrm{bit} \mapsto 0}^{r}_{I(i)} \\ \quad \vee P * \boxed{i.\mathrm{bit} \mapsto \_}^{r}_{I(i)} \end{array}}^{r'}_{J(i,P,\pi,r)}$$

Figure 8. Collected predicate definitions.

the permission $\mathbf{d}\pi$ on SET for the bit i. Hence, in order to acquire the full permission to set the shared bit, the thread must supply the resource P to the predicate $\mathsf{box}(i, P, 1)$. That is, the following abstract update holds:

$$\boxed{i.\mathrm{bit} \mapsto 0}^{r}_{I(i)} * \mathsf{box}(i, P, \pi) * P \implies \boxed{i.\mathrm{bit} \mapsto 0}^{r}_{I(i)} * [\mathrm{SET}]^{r}_{\mathbf{d}\pi}$$

Below, we prove that the abstract implication holds. For the moment, we just note that boxes are used to control the splitting of resources according to the splitting axiom. Note that, the definition of box is recursive, as it mentions the box predicate inside the shared region, and in the interference on the shared region. The fixed point exists by first finding a solution ignoring the interference environment, and then restricting the interference environment by the resulting solution.

Finally, we give a definition to $\mathsf{fut}(i, P)$. This assertion must capture one essential piece of information: that either the shared bit at i is zero, or the resource P is available for collection.

In these definitions, names surrounded with square brackets are capabilities. The semantics of such capabilities are defined by the interference environments. We define two environments. The first, $I(i)$ defines the interference over the shared bit i. This environment includes only a single operation, the ability to set the shared bit:

$$I(i) \triangleq (\mathrm{SET}: \ i.\mathrm{bit} \mapsto 0 \ \rightsquigarrow \ i.\mathrm{bit} \mapsto 1)$$

The interference environment $J(i, P, \pi, r)$ defines the interference over the resource-holding regions.

$$J(i, P, \pi, r) \triangleq \left(\begin{array}{l} \mathrm{PUT}: \left\{ \begin{array}{l} [\mathrm{SET}]^{r}_{\mathbf{d}\pi} \ \rightsquigarrow \ P \\ \left(\begin{array}{l} \mathsf{box}(i, P_1, \pi_1) * \mathsf{box}(i, P_2, \pi_2) * \\ P \mathrel{-\!\!*} (P_1 * P_2) \wedge \pi_1 + \pi_2 = \pi \end{array} \right) \ \rightsquigarrow \ \mathsf{emp}\,, \end{array} \right. \\ \\ \mathrm{GET}: \left\{ \begin{array}{l} P \ \rightsquigarrow \ \mathsf{emp} \\ [\mathrm{SET}]^{r}_{\mathbf{d}\pi} \ \rightsquigarrow \left(\begin{array}{l} \mathsf{box}(i, P_1, \pi_1) * \mathsf{box}(i, P_2, \pi_2) * \\ P \mathrel{-\!\!*} (P_1 * P_2) \wedge \pi_1 + \pi_2 = \pi \end{array} \right) \end{array} \right. \end{array} \right)$$

Intuitively the first case for PUT allows the thread to push the resource P into the shared state, and retrieve a fractional permission to SET the shared bit. The first case for GET allows the thread to retrieve the resource P. The second cases are used in resource splitting; see below for details.

The first obligation for showing that our module implements the abstract specification is to use our program logic to prove the module functions' specifications. Proofs for `grant`, `wait` and `split` are given in Fig. 9. The proof of `newchan` is almost identical to the proof of `split`, and hence omitted.

The proof of `grant` operates by first appealing to the specification of `wait` to recover the full resource from a possible fut predicate. We also use the specification $\{\mathsf{pa(i)}\}\texttt{wait(i)}\{\mathsf{emp}\}$, which can be proved trivially. It then exchanges the resource and the box predicate for permission to set the shared bit. Finally it sets the shared bit and forgets all the remaining resource.

The proof of `wait` spins until the bit field is 1, which excludes the case where the resource is not present. As the resource can only be removed by the `wait` thread, this assertion is stable under interference. The thread uses the GET to recover the resource, and garbage-collects all the other resources.

The proof of `split` (and `newchan`) allocates a new piece of memory, sets the `pred` and `bit` fields to appropriate values, then creates the fut and req predicates by wrapping the new memory in a shared region.

The second obligation we must discharge is to show that the predicates are stable. To do this, we check each of the predicate definitions to make sure that each shared region assertion is invariant under permitted interference.

3.3 Resource Splitting Using Boxes

Our specification requires that we can split fut predicates according to the axiom given in Fig. 2.

We use the box predicate to support this splitting in our concrete implementation. Intuitively, each box initially shares its shared region with a fut predicate. Then, if that fut predicate is split, the box instead contains a pair of boxes representing the shared state for the two new fut predicates.

The definition of a predicate $\mathsf{box}(i, P, \pi)$, Fig. 8, either allows the thread to access the SET permission, or contains two boxes with resources P_1 and P_2 such that $P \mathrel{-\!\!*} P_1 * P_2$. In the proofs of the module's operations, we relied on the assumption that a predicate $\mathsf{box}(i, P, 1)$ and resource P can be exchanged for a permission to set the shared bit $i.\mathrm{bit}$. We now justify this assumption with a proof.

Our definition of box is the least fixed point of the recursive definition. We reason inductively, hence it suffices to prove that the entailment holds when box is defined as false, and under the assumption that the disjunction holds. The base case of the induction holds trivially as $\mathsf{box}(i, P, \pi)$ is false. In the first inductive case, we assume that the left disjunct in the shared region holds. The proof is given in Fig 10 **(a)**. In the second case, we assume that the property holds for $\mathsf{box}(i, P_1, \pi_1)$ and $\mathsf{box}(i, P_2, \pi_2)$. The proof is given in Fig. 10 **(b)**.

The proof given in Fig. 10 **(c)** shows that the future-splitting axiom holds for the concrete predicate definitions. We only show the left case for the disjunction; the right case is easy. This proof uses the GET action while at the same time creating new regions for the two new futures. This completes the proof that our simple implementation corresponds to our high-level specification.

4. Logic and Semantics

In this section, we present the syntax and semantics of our logic. It extends the previous work on concurrent abstract predicates [6] with higher-order parameters and quantification following the work of Biering *et al.* on higher-order separation logic [3].

Our assertion logic is a typed higher-order separation logic extended with predicates that denote the ability to change the state and a connective for expressing sharing. The syntax of the assertion

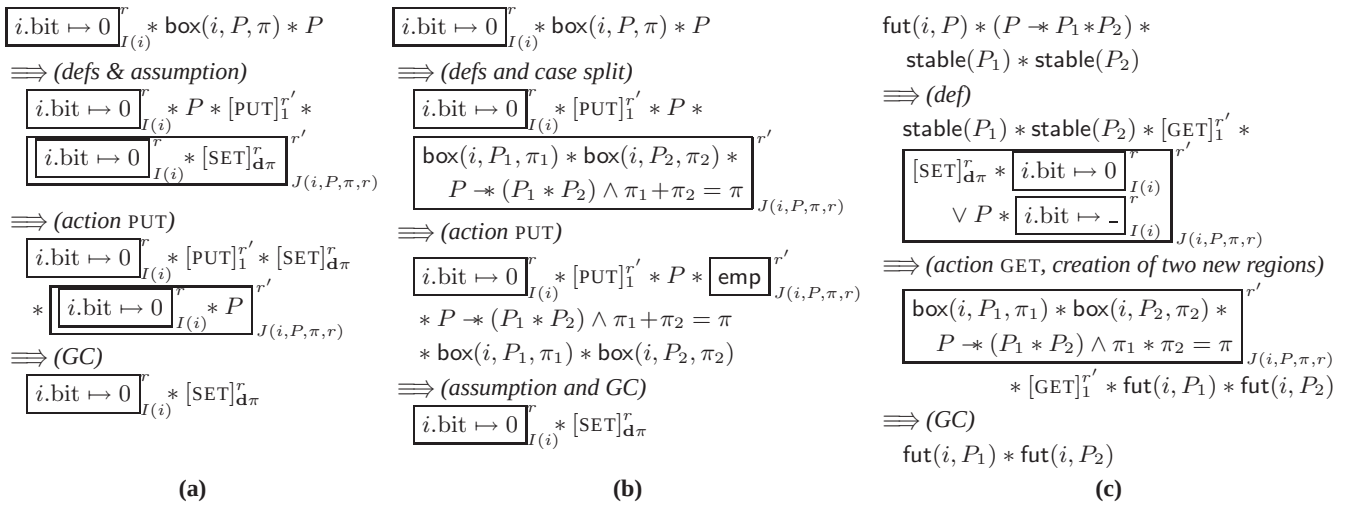

Figure 10. Proofs of abstract updates

language is as follows:

$$
\begin{array}{rcl}
\tau & ::= & \mathsf{Int} \mid \mathsf{Frac} \mid \mathsf{Region} \mid \mathsf{Asn} \mid \tau \to \tau \\
P, Q, L, M, \Delta & ::= & \mathsf{false} \mid P \Rightarrow Q \mid \exists x{:}\tau.\, P \mid L\, M \mid \lambda x{:}\tau.\, M \\
& \mid & E \mid \mathsf{emp} \mid P * Q \mid P \mathrel{-\!\!*} Q \\
& \mid & L \mapsto M \mid \mathsf{stable}(P) \mid [\gamma(\vec{M})]^r_\pi \mid \boxed{\mathrm{P}}^r_I \\
I & ::= & \gamma(\vec{x}) : \exists \vec{y} : \vec{\tau}(P \rightsquigarrow Q) \mid I, I \\
\pi & ::= & 1 \mid \mathbf{d}L \mid \mathbf{g}L
\end{array}
$$

where r ranges over region names, γ over token names, and v over values. We lift expressions E from the programming language to the logic. Note that we use P, Q, Δ when the term is of type Asn. Terms are typed in the obvious way, and we will implicitly assume all definitions are well-typed.

Our propositions have three important aspects they describe: (1) the contents of the state, (2) the capability to change state, and (3) a partitioning of these contents and capabilities between local and shared regions.

For completeness, the full semantics of terms is given in Fig. 11. Below we will only describe the salient features of the semantics. A more thorough explanation can be found in [6].

Model We model propositions with Worlds that have three components: a local component, LWorld, that specifies the current local state and local capabilities; a shared component, SWorld, that specifies the current shared state and shared capabilities; and an interference environment, IEnv, that specifies the possible interference (or protocol) on the shared component of the world. The shared component is split into many named regions, each of which is modelled by an LWorld.

A local world, LWorld, is modelled by a partial heap, Heap, specifying the locally accessible state, and a capability mapping, Capab, mapping from actions in Action to permission to perform that action in DG. Each action is mapped to either a full permission 1, an exclusive permission to perform that action, that hence prohibits the environment from performing the action; a guarantee permission $\mathbf{g}z$, a non-exclusive permission to perform that action; a deny permission $\mathbf{d}z$, a non-exclusive prohibition on the action that also prevents the environment performing it; and an empty permission 0 that does not allow the action but does not prohibit the environment from performing it. Following Boyland [5], the z components of deny and guarantee are used to track how much permission is required to re-establish exclusive permission.

Members of Action comprise a Region, a Token and a sequence of Val arguments. An action's semantic meaning as interference over a shared region is defined by an interference environment, in the set IEnv. The definition of an interference environment as a relation over SWorld enforces the restriction that the interpretation of an action does not allow you to change the interference interpretation of any actions.

Model operations As we are building a separation logic we require a composition operator on worlds that will be used to interpret the separating conjunction and separating implication. We use the standard operation from separation logic for combining heaps, $h \oplus h'$, by disjoint partial function combination.

We use the deny-guarantee composition model [8] for DG. This has 0 as the unit of $\oplus$. It combines two guarantee (or deny) permissions by combining their fractional components to produce a guarantee (or deny) permission with the sum of the fractions. If the fractions sum to 1 then it lifts to 1. If the fractions sum to more than 1, then combination is undefined. This is then lifted to the function space in the obvious way.

We define the composition of LWorld as the combination on both components; and on World as the combination on the LWorld component, where the SWorld and IEnv components are equal.

We define other useful operations on the model that aid in the definition of the semantics: $\lfloor s \rfloor$ collapses all the shared regions into a single one; l_H gives the heap component of l; l_P gives the permission component of l; and $\lVert w \rVert$ collapses a world into a single heap.

Finally, we define the set of well-formed worlds, WFW. A world is well-formed iff all the regions and the local component can be combined, each capability is defined in the interference environment, and the capabilities only mention valid regions.

Types The types are semantically interpreted as in Figure 11. We use i for interpretations of the free variables in a term: it is a dependent product from a variable to the denotation of the type of that variable. We interpret propositions on the powerset of worlds.

Terms The interpretation of false, $\Rightarrow$, $\exists$, $*$, $-\!\!*$, variables, function application, and function abstraction λ are standard.

The predicate emp specifies that the local component of the heap is empty and makes no restriction on the shared part, the

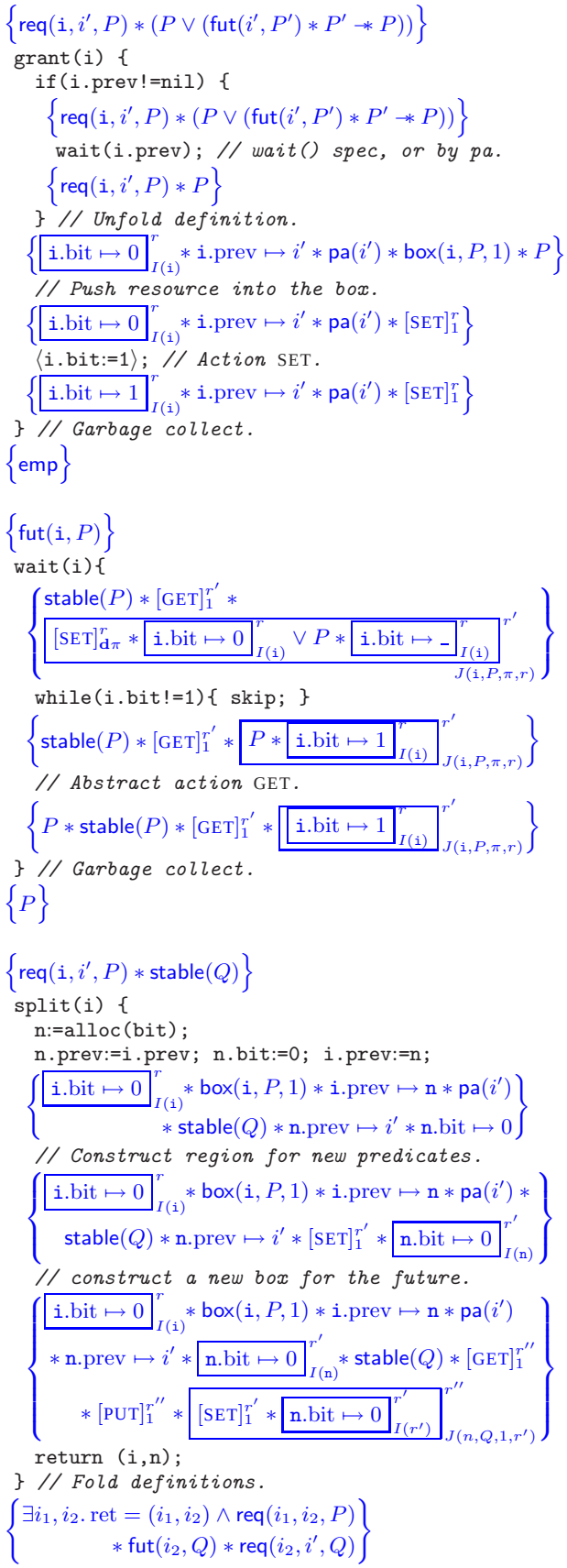

Figure 9. Proofs for `grant`, `wait` and `split`.

interference environment, or the capability. The points-to predicate $L \mapsto M$ specifies that the location L contains the value M in the local world, and that the heap contains nothing else.

The capability $[\gamma(\vec{M})]^r_\pi$ that says the local world contains the π permission on region r for action γ with parameters $\vec{M}$. The assertion $\mathsf{stable}(P)$ says that P will remain true given the permitted interference on the shared world. That is, if we start in a world satisfying P and take a step in R, then we must still satisfy P. The shared assertion $\boxed{P}^r_I$ says that the shared region r satisfies the assertion P and that region's interference is specified by I.

Interference We define several relations giving the possible updates to the shared world as a result of the thread and the environment. Following Jones [16], we call the interference permitted to the environment the *rely* and the interference permitted to the local thread the *guarantee*.

We define the semantics of the interference specfication as a relation of SWorlds. For a particular update $P \rightsquigarrow Q$, we specify that a part of the pre-state must satisfy P, and replacing that part with a part satisfying Q gives the post-state. We also allow the action to increase the number of regions in an unspecified way. This will allow actions both to repartition and to create new shared regions simultaneously.

We allow the dynamic creation of regions. The relations $\mathsf{R_c}$ and $\mathsf{G_c}$ model this creation. The first, $\mathsf{R_c}$, specifies the world-change if the environment creates a region. The environment can only create a region if it does not already exist. It adds a new shared region, and the relevant definition to the interference environment. The second, $\mathsf{G_c}$, specifies the world-change if the current thread created a region. This differs from the rely as all of the permissions on actions for the new region are given to the current thread.

The global rely relation, R, allows any action in the inference environment that is not explicitly prohibited with a deny permission or a full permission, as well as the creation of regions. We restrict R to well-formed worlds.

The global guarantee, G, allows any action for which there is either a full permission or a guarantee permission. The guarantee requires that the permissions and heap domain must be the same before and after the action, upto repartioning between regions. This ensures that permissions and heap cannot be created out of thin air. We also allow region creation, and restrict G to well-formed worlds.

Program logic We give the proof rules for our program logic in Figure 12. The judgements are of the form $\Delta;\Gamma \vdash \{P\}\ C\ \{Q\}$ where Δ is an assumption about the logical context, and Γ is an assumption about the procedures in the context of the form

$$\{P_1\}f_1\{Q_1\}, \ldots, \{P_n\}f_n\{Q_n\}.$$

We use Δ to encode the assumptions about the abstract predicates and their axioms

We assume a standard semantics of programs [6] where $(C,h) \xrightarrow{\eta} (C',h')$ denotes a successful reduction in the procedure context η (a mapping from procedure names to commands); and $(C,h) \xrightarrow{\eta} \mathit{fault}$ denotes a memory access problem. We then define the semantics of judgments as follows:

DEFINITION 1 (Configuration safety). *C, w, η, i, Q safe$_0$ always holds; and*
C, w, η, i, Q safe$_{n+1}$ iff the following four conditions hold:

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

Model

$$\pi \in \mathsf{DG} \triangleq \{1,0\} \uplus \{tz \mid z \in (0,1) \wedge t \in \{\mathbf{d},\mathbf{g}\}\} \quad a \in \mathsf{Action} \triangleq \mathsf{Region} \times \mathsf{Token} \times \mathsf{Val}^* \quad \rho \in \mathsf{Capab} \triangleq \mathsf{Action} \to \mathsf{DG}$$
$$h \in \mathsf{Heap} \triangleq \mathsf{Address} \to \mathsf{Val} \uplus \{\bot\} \quad l \in \mathsf{LWorld} \triangleq \mathsf{Heap} \times \mathsf{Capab} \quad s \in \mathsf{SWorld} \triangleq \mathsf{Region} \rightharpoonup \mathsf{LWorld}$$
$$\mathcal{I} \in \mathsf{IEnv} \triangleq \mathsf{Action} \rightharpoonup \mathcal{P}(\mathsf{SWorld} \times \mathsf{SWorld}) \quad w \in \mathsf{World} \triangleq \mathsf{LWorld} \times \mathsf{SWorld} \times \mathsf{IEnv}$$

Model operations

$$\bot \oplus v \triangleq v \oplus \bot \triangleq v \quad (t,z) \oplus (t,z') \triangleq \mathbf{1} \text{ if } z + z' = 1 \quad h_1 \oplus h_2 \triangleq \lambda v.\, h_1(v) \oplus h_2(v) \text{ if } \forall v.\, h_1(v) = \bot \vee h_2(v) = \bot$$
$$\pi \oplus 0 \triangleq 0 \oplus \pi \triangleq \pi \quad (t,z) \oplus (t,z') \triangleq (t, z+z') \text{ if } z + z' < 1 \quad \rho_1 \oplus \rho_2 \triangleq \lambda v.\, \rho_1(v) \oplus \rho_2(v) \text{ if } \forall v.\, \rho_1(v) \oplus \rho_2(v) \text{ defined}$$
$$(h_1,\rho_1) \oplus (h_2,\rho_2) \triangleq (h_1 \oplus h_2, \rho_1 \oplus \rho_2) \text{ if } h_1 \oplus h_2 \text{ and } \rho_1 \oplus \rho_2 \text{ are defined} \quad (h,\rho)_H \triangleq h \quad \lfloor s \rfloor \triangleq \oplus_{r \in \mathrm{dom}(s)} s(r)$$
$$(l_1,s_1,\mathcal{I}_1) \oplus (l_2,s_2,\mathcal{I}_2) \triangleq (l_1 \oplus l_2, s_1, \mathcal{I}_1) \text{ if } l_1 \oplus l_1 \text{ defined} \wedge s_1 = s_2 \wedge \mathcal{I}_1 = \mathcal{I}_2 \quad (h,\rho)_P \triangleq \rho \quad [\![(l,s,\mathcal{I})]\!] \triangleq (l \oplus \lfloor s \rfloor)_H$$
$$\mathsf{WFW} \triangleq \{(l,s,\mathcal{I}) \mid (l \oplus \lfloor s \rfloor) \text{ defined} \wedge \mathrm{dom}((l \oplus \lfloor s \rfloor)_P) \subseteq \mathrm{dom}(\mathcal{I}) \wedge (\forall r.\, r \in \mathrm{dom}(s) \Leftrightarrow \exists \gamma, \vec{v}.\, (r,\gamma,\vec{v}) \in \mathrm{dom}(\mathcal{I}))\}$$

Types

$$[\![\mathsf{Int}]\!] \triangleq \mathbb{Z} \quad [\![\mathsf{Region}]\!] \triangleq \mathsf{Region} \quad [\![\tau_1 \to \tau_2]\!] \triangleq [\![\tau_1]\!] \to [\![\tau_2]\!] \quad [\![\mathsf{Asn}]\!] \triangleq \mathcal{P}(\mathsf{World})$$

Terms

$$[\![\mathsf{false}]\!]_i \triangleq \emptyset \quad [\![x]\!]_i \triangleq i(x)$$
$$[\![P \Rightarrow Q]\!]_i \triangleq \{w \mid w \notin [\![P]\!]_i \vee w \in [\![Q]\!]_i\} \quad [\![L\ M]\!]_i \triangleq [\![L]\!]_i([\![M]\!]_i) \quad [\![P * Q]\!]_i \triangleq \{w \oplus w' \mid w \in [\![P]\!]_i \wedge w' \in [\![Q]\!]_i\}$$
$$[\![\exists x\colon \tau.\, P]\!]_i \triangleq \bigcup_{v \in [\![\tau]\!]} [\![P]\!]_{i[x \mapsto v]} \quad [\![\lambda x\colon \tau.\, M]\!]_i \triangleq \lambda v.\, [\![M]\!]_{i[x \mapsto v]} \quad [\![P \mathrel{-\!\!*} Q]\!]_i \triangleq \{w \mid \forall w' \in [\![P]\!]_i.\, w \oplus w' \in [\![Q]\!]_i\}$$
$$[\![\mathsf{emp}]\!]_i \triangleq \{((\emptyset,\rho),s,\mathcal{I})\} \quad [\![L \mapsto M]\!]_i \triangleq \{(([[\![L]\!]_i \mapsto [\![M]\!]_i],\rho),s,\mathcal{I})\} \quad \left[\!\!\left[[\gamma(\vec{M})]^r_\pi \right]\!\!\right]_i \triangleq \{((\emptyset,\rho),s,\mathcal{I}) \mid \rho(r,\gamma,[\![\vec{M}]\!]_i) \geq \pi\}$$
$$[\![\mathsf{stable}(P)]\!]_i \triangleq \{w \mid \forall w_1 \in [\![P]\!]_i.\, (w_1,w_2) \in \mathsf{R} \Rightarrow w_2 \in [\![P]\!]_i\} \quad \left[\!\!\left[\boxed{P}^r_I \right]\!\!\right]_i \triangleq \{((\emptyset,\rho),s,\mathcal{I}) \mid (s(r),s,\mathcal{I}) \in [\![P]\!]_i \wedge \mathcal{I}(r) = [\![I]\!]_{i,r}\}$$

Interference

$$[\![\gamma(\vec{x}) : \exists \vec{y} : \vec{\tau}(P \rightsquigarrow Q)]\!]_{i,r}(r,\gamma',\vec{v}) \triangleq \left\{ s_1,s_2 \;\middle|\; \begin{array}{l} \exists \vec{v'} \in [\![\vec{\tau}]\!], \mathcal{I}, l_0, l_1, l_2.\, (l_1,s_1,\mathcal{I}) \in [\![P]\!]_{i[\vec{x} \mapsto \vec{v}, \vec{y} \mapsto \vec{v'}]} \wedge (l_2,s_2,\mathcal{I}) \in [\![Q]\!]_{i[\vec{x} \mapsto \vec{v}, \vec{y} \mapsto \vec{v'}]} \\ \quad \wedge\, \gamma = \gamma' \wedge s_1(r) = l_1 * l_0 \wedge s_2(r) = l_2 * l_0 \\ \quad \wedge\, \forall r' \in \mathrm{dom}(s_1).\, r' \neq r \Rightarrow s_1(r') = s_2(r') \end{array} \right\}$$
$$[\![I, I']\!]_{i,r} \triangleq [\![I]\!]_{i,r} \cup [\![I']\!]_{i,r}$$
$$\mathsf{R_c} \triangleq \{(l,s,\mathcal{I}),(l,s',\mathcal{I} \cup \mathcal{I}') \mid r \notin \mathrm{dom}(s) \wedge s' = s[r \mapsto l'] \wedge \mathrm{rdom}(\mathcal{I}') = \{r\}\}$$
$$\mathsf{G_c} \triangleq \{(l,s,\mathcal{I}),(l',s',\mathcal{I} \cup \mathcal{I}') \mid r \notin \mathrm{dom}(s) \wedge s' = s[r \mapsto l_1] \wedge l \oplus \mathsf{all}(\mathcal{I}') = l_1 \oplus l' \wedge \mathrm{rdom}(\mathcal{I}') = \{r\}\}$$
$$\text{where} \quad \mathsf{all}(\mathcal{I}') \triangleq \bigoplus_{(r,\gamma,\vec{v}) \in \mathrm{dom}(\mathcal{I}')} (\emptyset, [r,\gamma,\vec{v} \mapsto 1]) \quad \text{and} \quad \mathrm{rdom}(\mathcal{I}) = \{r \mid (r,\_,\_) \in \mathrm{dom}(\mathcal{I})\}$$
$$\mathsf{R} \triangleq \left(\left\{ (l,s,\mathcal{I}),(l,s',\mathcal{I} \cup \mathcal{I}') \;\middle|\; \exists a.\, (s,s') \in \mathcal{I}(a) \wedge (\lfloor s \rfloor \oplus l)_P(a) \notin \{1, \mathbf{d}z\} \wedge \mathrm{dom}(s') \setminus \mathrm{dom}(s) = \mathrm{rdom}(\mathcal{I}') \right\} \cup \mathsf{R_c} \right) \cap \mathcal{P}(\mathsf{WFW}^2)$$
$$\mathsf{G} \triangleq \left(\left\{ \begin{array}{l} (l,s,\mathcal{I}), \\ (l',s',\mathcal{I} \cup \mathcal{I}') \end{array} \;\middle|\; \begin{array}{l} \exists a.\, (s,s') \in \mathcal{I}(a) \wedge (l'_P)(a) \in \{1,(\mathsf{g},\_)\} \wedge \mathrm{dom}(s') \setminus \mathrm{dom}(s) = \mathrm{rdom}(\mathcal{I}') \\ \quad \wedge\, (\lfloor s \rfloor \oplus l)_P \oplus \mathsf{all}(\mathcal{I}') = (\lfloor s' \rfloor \oplus l')_P \wedge \mathrm{dom}((\lfloor s \rfloor \oplus l)_H) = \mathrm{dom}((\lfloor s' \rfloor \oplus l')_H) \end{array} \right\} \cup \mathsf{G_c} \right) \cap \mathcal{P}(\mathsf{WFW}^2)$$

Ancillary definitions

$$P \Longrightarrow_i^{\{p\}\{q\}} Q \triangleq \forall w \in [\![P]\!]_i.\, \exists h \in [\![p]\!]_i.\, \forall h_2 \in [\![q]\!]_i.\, \exists h'\, w_2.\, h \oplus h' = [\![w]\!] \wedge h_2 \oplus h' = [\![w_2]\!] \wedge (w,w_2) \in \mathsf{G} \wedge w_2 \in [\![Q]\!]$$
$$[\![\Delta]\!] \triangleq \{i \mid [\![\Delta]\!]_i = [\![\mathsf{Asn}]\!]\} \quad \Delta \models \Delta' \triangleq [\![\Delta]\!] \subseteq [\![\Delta']\!]$$
$$\Delta \models P \Longrightarrow^{\{p\}\{q\}} Q \triangleq \forall i \in [\![\Delta]\!].\, P \Longrightarrow_i^{\{p\}\{q\}} Q \quad \Delta \models P \Longrightarrow Q \triangleq \forall i \in [\![\Delta]\!].\, P \Longrightarrow_i^{\{\mathsf{emp}\}\{\mathsf{emp}\}} Q$$

Figure 11. Semantics of assertions

$$\frac{\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\}}{\Delta;\Gamma \vdash \{p\}\ C\ \{q\}}\ (\textsc{Prim}) \quad \frac{\Delta;\Gamma \vdash \{P_1\}\ C_1\ \{Q_1\} \quad \Delta;\Gamma \vdash \{P_2\}\ C_2\ \{Q_2\}}{\Delta;\Gamma \vdash \{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\ (\textsc{Par}) \quad \frac{\alpha \notin \Gamma, P, Q \quad \Delta;\Gamma \vdash \{P\}C\{Q\}}{(\exists \alpha.\Delta);\Gamma \vdash \{P\}C\{Q\}}\ (\textsc{Exists})$$

$$\frac{\{P\}\ f\ \{Q\} \in \Gamma}{\Delta;\Gamma \vdash \{P\}\ f\ \{Q\}}\ (\textsc{Call}) \quad \frac{\vdash_{\mathsf{SL}} \{p\}\ C\ \{q\} \quad \Delta \models P \Longrightarrow^{\{p\}\{q\}} Q}{\Delta;\Gamma \vdash \{P\}\ \langle C \rangle\ \{Q\}}\ (\textsc{Atomic}) \quad \frac{\Delta;\Gamma \vdash \{P\}\ C\ \{Q\} \quad \Delta \models \mathsf{stable}(R)}{\Delta;\Gamma \vdash \{P * R\}\ C\ \{Q * R\}}\ (\textsc{Frame})$$

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

Figure 12. Selected proof rules from [6]. All rules assume that the pre- and post-conditions of their judgements are stable.

DEFINITION 2 (Judgement Semantics). $\Delta;\Gamma \models \{P\}\,C\,\{Q\}$ *holds iff*

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

where $[\![\Gamma]\!]_{n,i} \triangleq \{\eta \mid \forall\{P\}f\{Q\} \in \Gamma.\ \models_{\eta,i,n} \{P\}\eta(f)\{Q\}\}$ *and* $\models_{\eta,i,n} \{P\}C\{Q\} \triangleq \forall w \in ([\![P]\!]_i \cap \mathsf{WFW}).\, C, w, \eta, i, Q$ *safe*$_n$.

Differences from the CAP paper [6]. The original paper treated the meaning of interference syntactically in the model, that is, the equivalent of IEnv was a map from action to syntactic definition of the actions. This was done to avoid a cyclic definition in World. In this paper, we have factored out the semantics of interference to be a separate component. We thus impose the restriction that the interference environment cannot update the interference environment. Note, this kind of update was not allowed before, but was not explicitly forbidden in the model, just in the interpretation. This small refactoring of the semantics allows higher-order quantification.

We extend the model from the original CAP paper to additionally contain deny permissions [8]. This is a straightforward extension to the original paper.

Finally, we take an intuitionistic model for the permissions. This enables permissions to leak. The library we are considering in this paper requires garbage collection to collect signals when they are no longer accessible.

5. Verifying a More Complex Implementation

The module implementation given in §3 imposes a strong sequential order on calls to `grant`. A `wait` only checks its immediate predecessor, so a call to `grant` must ensure its predecessor is set before setting its own bit. In this section, we consider an alternative implementation that allows out-of-order bit setting. We prove that this implementation also implements our abstract specification.

The new implementation uses the same data-structure as the simple implementation. Bits can be set by calls to `grant` in arbitrary order, but as a consequence, each call to `wait` must examine *all* prior bits before exiting. As this implementation uses the same data-structure as the first one, the `split` and `newchan` operations are identical. The `grant` and `wait` operations are defined as follows:

```
grant(i) {
  ⟨i.bit := 1⟩;
}
```

```
wait(i) {
  while (i!=nil) {
    while(i.bit=0){ skip; }
    i := i.prev;
} }
```

As with the first implementation, each address has a `bit` field and a `prev` field. Calling `grant` sets the bit field for the current address from 0 to 1, then exits immediately. When `wait` is called, it blocks until every bit field earlier in the order is set. To do this, it chases `prev` fields, waiting for each `bit` field to go to 1 before accessing the preceding location. In this way, `wait` ensures that all previous threads have called `grant`.

The predicate definitions (given in Fig. 13) are similar to those for the simple implementation. The main difference is in the definition of the fut predicate. When the shared bit is set, the resource that is available to the thread may include a preceding fut predicate. So, if the current thread expects resource P, it may instead get a resource satisfying $(P' \mathrel{-\!\!*} P) * \mathsf{fut}(i', P')$, where i' is the immediately preceding location in the logical order.

The thread can then recover P' by checking the bit for i', which may include a fut predicate for the preceding location i''. Only when the thread has checked all the bits earlier in the order can it be confident it holds the full resource. In this way, our predicate definitions reflect the fact that the thread does not know exactly which threads have supplied a resource, and which have simply renounced access to it.

$$\mathsf{fut}(i,P) \triangleq [\textsc{get}]_1^{r'} * \left(\boxed{i.\mathrm{bit} \mapsto 0}^{r}_{I(i)} * \forall^* i'.\,[\textsc{set}(i')]^r_{\mathbf{d}\pi}\right) \vee \left(\begin{array}{l}\boxed{\begin{array}{l} i.\mathrm{bit} \mapsto 0 \vee \\ i.\mathrm{bit} \mapsto 1 * i.\mathrm{prev} \mapsto i' * \mathsf{pa}(i')\end{array}}^{r}_{I(r)} \\ * P \vee \exists P', i'.\,(\mathsf{fut}(i',P') * (P' \mathrel{-\!\!*} P))\end{array}\right)^{r'}_{J(i,P,\pi,r)}$$

$$\mathsf{req}(i,i',P) \triangleq \boxed{i.\mathrm{bit} \mapsto 0}^{r}_{I(i)} * i.\mathrm{prev} \mapsto i' * \mathsf{pa}(i') * \mathsf{box}(i,P,1)$$

$$\mathsf{pa}(i) \triangleq i = \mathsf{nil} \vee \boxed{\begin{array}{l} i.\mathrm{bit} \mapsto 0 \vee \\ i.\mathrm{bit} \mapsto 1 * i.\mathrm{prev} \mapsto i' * \mathsf{pa}(i')\end{array}}^{r}_{I(i)}$$

$$\mathsf{box}(i,P,\pi) \triangleq \boxed{\begin{array}{l} i.\mathrm{bit} \mapsto 0\ {}^{r}_{I(i)} * \forall^* i.\,[\textsc{set}(i)]^r_{\mathbf{d}\pi} \vee \\ \mathsf{box}(i,P_1,\pi_1) * \mathsf{box}(i,P_2,\pi_2) * \\ \quad P \mathrel{-\!\!*} (P_1 * P_2) \wedge \pi_1 + \pi_2 = \pi\end{array}}^{r'}_{J(i,P,\pi,r)} * [\textsc{put}]_1^{r'}$$

Figure 13. Predicate definitions for out-of-order bit setting.

```
1  {fut(i, P)}
2  wait(i) {
3    while (i!=nil) {
4      {i ≠ nil ∧ (pa(i) * P ∨ ∃P'. fut(i, P') * (P' -* P))}
5      while(i.bit=0) skip;
6      {i.bit ↦ _ * i.prev ↦ i' * pa(i')^r *
         (P ∨ ∃P'. fut(i', P') * (P' -* P))}
7      i := i.prev;
8    }
9    {i = nil ∧ ((pa(i) * P) ∨ (∃P'. fut(i, P') * (P' -* P)))}
10 } // fut(i, _) is false if i = nil, so...
11 {P}
```

Figure 14. Proof for `wait`.

The req predicate is defined similarly to the first proof, with a recursive box predicate controlling access to bit-setting.

The interference environment for the shared bit, $I(i)$, is:

$$\textsc{set}(i'):\quad i.\mathrm{bit} \mapsto 0 \quad \rightsquigarrow \quad i.\mathrm{bit} \mapsto 1 * i.\mathrm{prev} \mapsto i' * \mathsf{pa}(i')$$

The environment $J(i,P,\pi,r)$ for resource-holding regions is:

$$\textsc{put}: \begin{cases} [\textsc{set}(i')]^r_{\mathbf{d}\pi} & \rightsquigarrow \quad P \vee (\mathsf{fut}(i',P') * P' \mathrel{-\!\!*} P) \\ \left(\begin{array}{l}\mathsf{box}(i,P_1,\pi_1) * \mathsf{box}(i,P_2,\pi_2) \\ * P \mathrel{-\!\!*} (P_1 * P_2) \wedge \pi_1 + \pi_2 = \pi\end{array}\right) & \rightsquigarrow \quad \mathsf{emp} \end{cases}$$

$$\textsc{get}: \begin{cases} P & \rightsquigarrow \quad \mathsf{emp} \\ (\forall^* i'.\,[\textsc{set}(i')]^r_{\mathbf{d}\pi}) & \rightsquigarrow \left(\begin{array}{l}\mathsf{box}(i,P_1,\pi_1) * \mathsf{box}(i,P_2,\pi_2) * \\ P \mathrel{-\!\!*} (P_1 * P_2) \wedge \pi_1 + \pi_2 = \pi\end{array}\right) \end{cases}$$

(Here the symbol $\forall^*$ is the iterated version of $*$.)

A proof for `wait` is given in Fig. 14. The most interesting step is line 5, where the resource is recovered from the shared region. We justify this step by the following proof. The other case, where a `pa` rather than `fut` is present, is trivial.

$$\left\{\mathsf{fut}(\mathtt{i}, P') * (P' \mathrel{-\!\!*} P)\right\}$$

// Unfold definitions.

$$\left\{\begin{array}{l}(P' \mathrel{-\!\!*} P) * [\mathrm{GET}]_1^{r'} * \\ \boxed{\mathtt{i.bit} \mapsto 0}^{r}_{I(\mathtt{i})} * \forall^* i'.\, [\mathrm{SET}(i')]^{r}_{\mathbf{d}\pi} \vee \\ \boxed{\mathtt{i.bit} \mapsto \_ * \mathtt{i.prev} \mapsto i' * \mathsf{pa}(i')}^{r}_{I(\mathtt{i})} \\ * (P' \vee \exists P''.\, \mathsf{fut}(i', P'') * (P'' \mathrel{-\!\!*} P'))\end{array}\right\}^{r'}_{J(\mathtt{i},P',\pi,r)}$$

```
while(i.bit = 0) { skip; }
```

$$\left\{\begin{array}{l}(P' \mathrel{-\!\!*} P) * [\mathrm{GET}]_1^{r'} * \\ \boxed{\mathtt{i.bit} \mapsto \_ * \mathtt{i.prev} \mapsto i' * \mathsf{pa}(i')}^{r}_{I(\mathtt{i})} \\ * (P' \vee \exists P''.\, \mathsf{fut}(i', P'') * (P'' \mathrel{-\!\!*} P'))\end{array}\right\}^{r'}_{J(\mathtt{i},P',\pi,r)}$$

// Pull into local state and GC GET.

$$\left\{\begin{array}{l}\boxed{\mathtt{i.bit} \mapsto \_ * \mathtt{i.prev} \mapsto i' * \mathsf{pa}(i')}^{r}_{I(\mathtt{i})} * \\ (P' \vee \exists P''.\, \mathsf{fut}(i', P'') * (P'' \mathrel{-\!\!*} P')) * (P' \mathrel{-\!\!*} P)\end{array}\right\}$$

// Transitivity of $\mathrel{-\!\!*}$.

$$\left\{\begin{array}{l}\boxed{\mathtt{i.bit} \mapsto \_ * \mathtt{i.prev} \mapsto i' * \mathsf{pa}(i')}^{r}_{I(\mathtt{i})} * \\ \quad (P \vee \exists P''.\, \mathsf{fut}(i', P'') * (P'' \mathrel{-\!\!*} P))\end{array}\right\}$$

The proof for `grant`, `newchan` and `split` are similar to the proofs for the single-bit case. Once again, the proof of `grant` depends on the fact that `fut` can be split according to the resource held by it.

6. Related Work and Conclusions

Most work on combining separation logic with concurrency constructs has considered them as primitive in the logic. This begins with O'Hearn's work on concurrent separation logic [20], which takes statically allocated locks as a primitive. CSL has been extended to deal with dynamically-allocated locks [11, 14, 15] and re-entrant locks [12]. Others have extended separation logic or similar logics with primitive channels [13, 1, 24, 18], and event driven programs [17].

Concurrent abstract predicates [6] combine the explicit treatment of concurrent interference from rely-guarantee [16, 10, 23] and abstraction through abstract predicates [21], with a concurrent fiction of disjointness [7] supported by capabilities [8]. In this paper we have combined concurrent abstract predicates with higher-order separation logic [3]. We used our higher-order logic to define and verify a specification for barriers that enforce complex data and control dependencies in concurrent programs.

Although we have focussed in this paper on barrier constructs used for deterministic parallelism [25, 2, 4, 19], our logic is intended as a general approach to specifying concurrency constructs. Our syntactic approach has the advantage that concurrency constructs of different kinds combine transparently. For example, lock predicates defined in [6] can be transferred through our channel predicates without changing the semantics or proofs of correctness for either module. In addition, we can verify that concrete implementations of constructs satisfy their specification.

Acknowledgements Thanks to the anonymous referees, Richard Bornat, Matko Botinčan, Thomas Dinsdale-Young, Philippa Gardner, Neel Krishnaswami, Daiva Naudžiūnienė, Viktor Vafeiadis and John Wickerson.

References

[1] C. J. Bell, A. Appel, and D. Walker. Concurrent separation logic for pipelined parallelization. In *SAS*, 2009.

[2] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, 2010.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *TOPLAS*, 29(5), 2007.

[4] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA '09*, pages 97–116. ACM, 2009.

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

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

[7] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, 2010.

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

[9] M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. Computer laboratory technical report, University of Cambridge, 2010.

[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] C. A. R. Hoare and P. W. O'Hearn. Separation logic semantics for communicating processes. *ENTCS*, 212:3–25, 2008.

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

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

[16] C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.

[17] N. R. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI*, 2010.

[18] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP*, 2010.

[19] A. Navabi, X. Zhang, and S. Jagannathan. Quasi-static Scheduling for Safe Futures. In *PPoPP*, pages 23–32. ACM, 2008.

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

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

[22] M. C. Rinard and M. S. Lam. Semantic Foundations of Jade. In *POPL*, pages 105–118. ACM, 1992.

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

[24] J. Villard, É. Lozes, and C. Calcagno. Tracking heaps that hop with heap-hop. In *TACAS*, pages 275–279, 2010.

[25] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *OOPSLA*, pages 439–435, 2005.