Verifying Concurrent Programs with VST

William Mansky

July 17, 2022

Abstract

The latest version of VST includes integration of most of the features of Iris [5], allowing us to verify concurrent programs with user-defined ghost state, global invariants, and logically atomic specifications. This document describes how to use Verifiable C to prove the correctness of concurrent programs, using examples that ship with VST. We will assume familiarity with the basics of VST, as described in the VST manual. The final sections also assume some familiarity with Iris: if you haven't used it before, Elizabeth Dietrich's guide [3] is a good resource for beginners.

1 Verifying a Concurrent Program with Locks

A concurrent C program is a sequential C program with a few additional features. It may create new *threads* of execution, which execute functions from the program in parallel, but with a single shared memory: any data on the heap (including global variables and `malloc`ed memory) can potentially be accessed by every thread. Threads can thus communicate by passing values to each other through memory locations, and threads may also *synchronize*, blocking each other's control flow to ensure that operations happen in a certain order. VST supports two kinds of synchronization by default: locks, and sequentially consistent atomic operations (which we will discuss in a later section). A *lock* data structure supports functions `acquire` and `release`, and ensures that it can be held by at most one thread at a time. When a thread tries to *acquire* a lock that is not currently available, it pauses its execution ("blocks") until the lock becomes available[1]. Locks can be used to enforce *mutual exclusion*, ensuring that a memory location is only accessed by one thread at a time. VST's locks are declared in the C header file `concurrency/threads.h`, which should be `#include`d in any Verifiable C concurrent program.

The file `progs64/incr.c`[2] contains a simple concurrent C program. It declares a global struct `c` of type `counter` with two fields: an integer `ctr`, and a `lock` that coordinates access to `ctr`. The struct has two accessor functions: `incr`, which increases the value of `ctr` by one, and `read`, which reads the current value of `ctr`. By acquiring the lock before using `ctr`, these functions ensure that their operations are well synchronized: without the lock, one thread might access `ctr` while another thread writes to it, causing a *data race*, which is undefined behavior in C. This is reflected in Verifiable C by the use of *shares*, as described in section 44 of the manual. A thread can only write to a memory location if it holds a sufficiently large share of the location that no other thread can possibly read from it. We allow `ctr` to be modified by multiple threads by moving its ownership share between threads via locks, as described below. A consequence of this share discipline is that if we prove any pre- and postcondition in Verifiable C for a program, we also know that as long as the precondition is met, the program does not

[1] Note that the thread that acquires a lock need not be the one that releases it: a locked lock can be passed through another synchronization mechanism to another thread, which then releases it (the "daring" concurrency of O'Hearn [9]), as demonstrated by the join lock in Section 2. Some authors use "lock" to refer specifically to a mutex that must be released by the thread that acquires it.

[2] We assume that readers are using VST in 64-bit mode, but these examples also appear in the 32-bit folder `progs`.

have any data races (just as proving correctness of a sequential program also implies that it has no null-pointer dereferences). In this section, we will focus on the verification of the `incr` and `read` functions, and demonstrate how to prove correctness of programs with locks.

The proof of correctness for `incr.c` is in `progs/verif_incr_simple.v`. It has several elements that do not appear in sequential Verifiable C proofs. First, its imports include `VST.concurrency.lock_specs`, which defines generic specifications for locks, and `VST.atomics.verif_lock`, which provides a verified implementation of the lock specifications. It then associates the `spawn` function with its standard specification. We will go through the specifications in detail in Section 5.1.

The first thing we need to do to verify the counter functions is to define a *lock invariant*, a predicate describing the resources protected by the `lock` field. A lock invariant can be any Verifiable C assertion (i.e., `mpred`), subject to an exclusivity condition described later. In this case, the lock protects the data in the `ctr` field. We want to know specifically that `ctr` always contains an unsigned integer value, so we use the lock invariant

$$\mathsf{cptr\_lock\_inv} \triangleq \mathsf{EX}\ z : \mathsf{Z}, \mathsf{field\_at\ Ews\ t\_counter\ [StructField\ \_ctr]\ (Vint(Int.repr}\ z))\ \mathtt{c}$$

We use the lock_inv predicate to assert that a lock exists in memory with a given invariant: lock_inv sh h R means that the current thread owns share sh of a lock with handle h with invariant R. The lock handle can contain various information, but always at least contains the location of the lock in memory, accessed as ptr_of h. Shares of a lock can be combined and split in the same way as shares of data_at, and any nonempty share is enough to acquire or release the lock[3].

Now we can give specifications to the functions that manipulate counters.

```
DECLARE _incr
 WITH gv : globals, sh1 : share, sh : share, h : lock_handle
 PRE [ ]
   PROP  (readable_share sh1; sh <> Share.bot)
   PARAMS() GLOBALS(gv)
   SEP   (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
          lock_inv sh h (cptr_lock_inv (gv _c)))
 POST [ tvoid ]
   PROP ()
   RETURN ()
   SEP (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
          lock_inv sh h (cptr_lock_inv (gv _c)))

DECLARE _read
 WITH gv : globals, sh1 : share, sh : share, h : lock_handle
 PRE [ ]
   PROP  (readable_share sh1; sh <> Share.bot)
   PARAMS() GLOBALS(gv)
   SEP   (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
            lock_inv sh h (cptr_lock_inv (gv _c)))
 POST [ tuint ]
   EX z : Z,
   PROP ()
   RETURN (Vint (Int.repr z))
```

[3]This contrasts with ordinary data_at, in which we need a writable share to write to a location; multiple threads can try to acquire a lock at the same time, and the lock's built-in synchronization will prevent any race conditions.

```
    SEP (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
         lock_inv sh h (cptr_lock_inv (gv _c))).
```

These are surprisingly boring specifications! The `read` function needs to know that we have access to `lock` and that it has the appropriate invariant, and returns some number about which we know nothing. The `incr` function does even less, just returning its resources as is. These are enough to prove *safety* of the program, to show that it has no data races, but not enough to learn much about what the program actually computes. This is a result of our invariant: when a thread acquires the lock, the *only* thing it knows about the memory it gains access to is that it satisfies the invariant. This is a well-known limitation of basic concurrent separation logic, and it is generally solved using *ghost state*, which we describe in Section 3. For now, we will describe how to prove safety for this program; later we will see how the proof of correctness builds on the safety proof.

There is one more important step before we can prove that the counter functions satisfy their specifications. In order to use a resource invariant, we need to show that it is *exclusive*, i.e., that it can only hold once in any given state. This is represented in VST by a property exclusive_mpred $R \triangleq R * R \vdash \mathsf{FF}$. This allows us to know that if the current thread holds the invariant, it also holds the lock. Fortunately, most common assertions (e.g., data_at for a non-empty type) are exclusive, so we can fairly easily prove the desired property ctr_inv_exclusive. It is useful to add this lemma to auto's hint database via Hint Resolve, so that the related proof obligations can be discharged automatically[4].

Now we can verify the bodies of `read` and `incr`, using the same Verifiable C tactics that we would use for a sequential program. The only new element is the use of the `acquire` and `release` functions, which allow threads to interact with locks and transfer ownership of resource invariants. We interact with these functions using the ordinary forward_call tactic. Their witnesses take three arguments: the location ℓ of the lock, the share *sh* of the lock it owned by the caller, and the lock invariant R. Their pre- and postconditions are as follows[5]:

$$\{!!sh \neq \mathsf{Share.bot} \wedge \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \mathtt{acquire}(\mathsf{ptr\_of}\ \ell)\ \{R * \mathsf{lock\_inv}\ sh\ \ell\ R\}$$

$$\{!!(sh \neq \mathsf{Share.bot}) * (\mathsf{exclusive}\ R) * R * \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \mathtt{release}(\mathsf{ptr\_of}\ \ell)\ \{\mathsf{lock\_inv}\ sh\ \ell\ R\}$$

When we acquire the lock, we also gain access to the invariant; when we release the lock, we must re-establish the invariant.

Consider the proof of body_read: we begin with the usual invocation of start_function. We then use forward_call to process the `acquire` call, adding cptr_lock_inv to the SEP clause. Unfolding its definition tells us that we now have access to `ctr`, and the integer stored in it, which we introduce as z. We assign z to the local variable `t`, and then release the lock. We use the lock_props tactic to discharge the exclusive obligation of `release` automatically, so that we need only prove that the invariant holds again. In this case, since have not changed the value of `ctr`, its value is still z. The return value of the function is that same z, and the proof is complete. The proof of body_incr is almost identical, except that at the call to `release` the invariant now holds at $z + 1$.

2 Thread Creation and Joining

Every C program starts its execution as a single-threaded program. It becomes concurrent when it spawns a new thread, for instance by calling Verifiable C's `spawn` function. The `spawn`

[4]In fact, the actual lock functions require a property called weak_exclusive_mpred that follows from exclusive_mpred but avoids universe inconsistencies; the lock_props tactic automatically makes the necessary transformations.

[5]In fact, this spec for `release` is a special case of a more general spec, as indicated by the `release_simple` argument to `forward_call`. We describe the lock specs in detail in Section 5.1.

function takes two arguments: a pointer to a function that the new thread should execute, and a `void*` that will be passed as an argument to that function. The new thread begins execution at the start of the indicated function, and continues to execute until it returns from that function; until then, it can assign to local variables, perform memory operations, and call other functions just as a single-threaded program would. Each thread has its own local variables, but memory is shared between all threads in a program. In the current version of VST, the starting function for a thread must take a single argument of type `void*` and return a value of type `void*`; the value returned is ignored completely, so it will usually be `NULL`.

The separation logic rule for `spawn` is:

$$\{P(y) * (f : x.\ \{P(x)\}\{\mathsf{emp}\})\}\ \texttt{spawn}(f, y)\ \{\}$$

where $f : x.\ \{P(x)\}\{\mathsf{emp}\}$ means that the function f takes a parameter x, and has precondition $P(x)$ and postcondition emp. From the parent thread's perspective, we give away resources satisfying the precondition of the spawned function f and get nothing back. Those resources now belong to the child thread, whose behavior is invisible to all other threads; the postcondition of emp reflects the fact that any resources held by the thread when it returns will be lost forever.

If we want to *join* with a spawned thread once it finishes, retrieving its resources and learning the results of any computations it performed, we can do so with a lock, which we can either pass as the argument to f or provide as a global variable. In order to recover *all* the resources the thread owned, including the share of the lock that we use for joining, we need to use a *recursive* lock, one whose invariant includes a share of the lock itself. We can make such an invariant with the $\mathsf{selflock}$ function, as we can see in the definition of $\mathsf{thread\_lock\_inv}$. Intuitively, $\mathsf{selflock}\ R\ sh\ \ell = R * \mathsf{lock\_inv}\ sh\ \ell\ (\mathsf{selflock}\ R\ sh\ \ell)$.

In Verifiable C, functions that will be passed to `spawn` must have specifications of a certain form, as in $\mathsf{thread\_func\_spec}$:

```
DECLARE _thread_func
 WITH y : val, x : share * share * lock_handle * lock_handle * globals
 PRE [ tptr tvoid ]
        let '(sh1, sh, h, ht, gv) := x in
        PROP  (readable_share sh1; sh <> Share.bot; ptr_of ht = y)
        PARAMS (y) GLOBALS (gv)
        SEP   (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
               lock_inv sh h (cptr_lock_inv (gv _c));
               lock_inv sh ht (thread_lock_inv sh1 sh (gv _c) h ht))
 POST [ tint ]
        PROP ()
        RETURN (Vint Int.zero)
        SEP ().
```

The WITH clause must have exactly two elements: one of type val that holds the argument passed to the function, and another that holds the entire rest of the witness, usually as a tuple. We can then destruct the tuple inside the precondition to access the rest of the witness. In the precondition, PARAMS must contain only the argument. The PROP and SEP clauses are unrestricted. The postcondition must return 0, and otherwise be empty. In this example, the thread function takes a share of the `lock` field and a share of the corresponding $\mathsf{lock\_inv}$ assertion, along with the recursive lock for joining, whose location is passed in the argument. The interesting part of the proof of this specification is the call to `release`, where we use a different subspecification ($\mathsf{release\_self}$, for recursive locks) to transfer all of the thread's resources (including its share of the lock) into the lock invariant.

Verifying the `main` function, which spawns the thread, is more complicated. First, we create the locks used by `ctr` and `thread_func`, using the `makelock` function:

$$\{\mathsf{mem\_mgr}\ gv\}\ \texttt{makelock}()\ \{\ell.\ \mathsf{lock\_inv}\ \mathsf{Tsh}\ \ell\ (R\ \ell)\}$$

We can make a lock with any invariant R at any time, and R can refer to the lock itself. We specifically do *not* need to know that R holds when we call `makelock`; we create the lock in the locked state, and only need to provide R when we release it. This is particularly convenient for making join-style locks, which are only released once (when the associated thread finishes its computation).

Next, we spawn the child thread using the forward_spawn tactic, a `spawn`-specific wrapper around forward_call. Its general form is forward_spawn *id arg w*, where *id* is the identifier of the function to be spawned, *arg* is the value of the provided argument, and w is the rest of the witness for the spawned function. The tactic automatically discharges the proof obligations of the spawn rule, leaving us to prove only the precondition of the spawned function. In this example, we split off a share of the `lock` field and each of the lock_inv assertions and provide them to the spawned thread to satisfy the precondition of `thread_func`, while retaining the other shares so that `main` can invoke the `incr` function in parallel with the spawned thread.

Finally, we join with the spawned thread by acquiring its lock. Because the lock is recursive, acquiring it allows us to retrieve the other half of both the thread lock and the counter lock, regaining full ownership. This allows us to deallocate the locks with calls to `freelock` (which like `release` has recursive and nonrecursive subspecs). We must hold a lock in order to free it, as seen in the nonrecursive `freelock` rule:

$$\{(\mathsf{exclusive}\ R) * R * \mathsf{lock\_inv}\ \mathsf{Tsh}\ \ell\ R\}\ \texttt{freelock}(\ell)\ \{R\}$$

Once we have freed both locks, the program is finished.

3 Using Ghost State

In the previous section, we proved that `incr.c` is safe, but not that the counter is 2 after being incremented twice. To prove that, our threads need to be able to record information about the actions they have performed on the shared state, instead of sealing all knowledge of the value of the `ctr` field inside the lock invariant. We can accomplish this with *ghost variables*, a simple form of auxiliary state.

In `verif_incr.v`, we augment the proof of the previous section with ghost variables and prove that the program computes the value 2. To do so, we use the new ghost_var assertion: ghost_var *sh a g* asserts that g is a *ghost name* (gname in Coq) associated with the value a, which may be of any type. We can split and join shares of ghost variables in the same way as memory locations, but they are not modified by program instructions. Instead, they can change by *view shifts*, which can be introduced at any point in the proof of a program. Whenever a thread holds full ownership (Tsh) of a ghost variable, it can change the value of the variable arbitrarily. For `incr.c`, we will add two ghost variables, each tracking the contribution of one thread to the value of the `ctr` field. We will divide ownership of each ghost variable between the lock invariant and one of the threads. By maintaining the invariant that `ctr` is the sum of the two contributions, we will be able to conclude that after two increments, the value of `ctr` is 2.

3.1 Extending the Specifications

Previously, the lock invariant for the `ctr` lock was

$$\mathsf{EX}\ z:\mathsf{Z}, \mathsf{field\_at}\ \mathsf{Ews}\ \mathsf{t\_counter}\ [\mathsf{StructField}\ \_\mathsf{ctr}]\ (\mathsf{Vint}(\mathsf{Int.repr}\ z))\ \mathsf{c}$$

Now, we augment it with shares of two ghost variables:

$$\begin{aligned}&\mathsf{EX}\ z:\mathsf{Z}, \mathsf{field\_at}\ \mathsf{Ews}\ \mathsf{t\_counter}\ [\mathsf{StructField}\ \_\mathsf{ctr}]\ (\mathsf{Vint}(\mathsf{Int.repr}\ z))\ \mathsf{c}\ *\\ &\quad \mathsf{EX}\ x:\mathsf{Z}, \mathsf{EX}\ y:\mathsf{Z}, !!(z = x + y)\ \&\&\ \mathsf{ghost\_var}\ \mathsf{gsh1}\ x\ g1 * \mathsf{ghost\_var}\ \mathsf{gsh1}\ y\ g2\end{aligned}$$

The thread that holds the other half of $g1$ or $g2$ can thus record its contribution to the `ctr`, but can only change that contribution while holding the lock, and only while maintaining the invariant that $z = x + y$.

Next, we modify each specification to take the ghost variables into account. Our specification for `incr` now needs to know which ghost variable the caller wants to increment, so it takes a boolean left telling it whether we are looking at the left ($g1$) or right ($g2$) ghost variable.

```
DECLARE _incr
 WITH sh1 : share, sh : share, h : lock_handle, g1 : gname, g2 : gname,
      left : bool, n : Z, gv: globals
 PRE [ ]
   PROP  (readable_share sh1)
   PARAMS () GLOBALS (gv)
   SEP   (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
          lock_inv sh h (cptr_lock_inv g1 g2 (gv _c));
          ghost_var gsh2 n (if left then g1 else g2))
 POST [ tvoid ]
   PROP ()
   LOCAL ()
   SEP (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
        lock_inv sh h (cptr_lock_inv g1 g2 (gv _c));
        ghost_var gsh2 (n+1) (if left then g1 else g2)).
```

Holding one of the ghost variables is not enough to guarantee anything about the value returned by `read`, but if we hold both of them, we should be able to predict the result.

```
DECLARE _read
 WITH sh1 : share, sh : share, h : lock_handle, g1 : gname, g2 : gname,
      n1 : Z, n2 : Z, gv: globals
 PRE [ ]
   PROP  (readable_share sh1)
   PARAMS () GLOBALS (gv)
   SEP   (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
          lock_inv sh h (cptr_lock_inv g1 g2 (gv _c));
          ghost_var gsh2 n1 g1; ghost_var gsh2 n2 g2)
 POST [ tuint ]
   PROP ()
   RETURN (Vint (Int.repr (n1 + n2)))
   SEP (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
        lock_inv sh h (cptr_lock_inv g1 g2 (gv _c));
        ghost_var gsh2 n1 g1; ghost_var gsh2 n2 g2).
```

Finally, we add ownership of ghost variable $g1$ to the resources passed to `thread_func` (and collected by its lock when it terminates):

```
DECLARE _thread_func
 WITH y : val, x : share * share * lock_handle * lock_handle * gname * gname * globals
 PRE [ tptr tvoid ]
   let '(sh1, sh, h, ht, g1, g2, gv) := x in
   PROP  (readable_share sh1; ptr_of ht = y)
   PARAMS (y) GLOBALS (gv)
   SEP   (field_at sh1 t_counter [StructField _lock] (ptr_of h) (gv _c);
          lock_inv sh h (cptr_lock_inv g1 g2 (gv _c));
          ghost_var gsh2 0 g1;
          lock_inv sh ht (thread_lock_inv sh1 sh h g1 g2 (gv _c) ht))
 POST [ tint ]
   PROP ()
   RETURN (Vint Int.zero)
   SEP ().
```

The value of $g1$ starts at 0, and should be 1 by the time the thread terminates, as reflected in thread_lock_R.

3.2 Proving with Ghost State

The proof for incr begins in the same way as before: we acquire the lock, unfold the invariant, and introduce the variables x, y, and z. This also gains us gsh1 shares of both ghost variables. The code that reads and increments ctr proceeds in the same way as before; even though the value of ctr has increased, the ghost variables are not yet updated. We do the update after the increment, but in fact we are free to do it anytime between acquiring and releasing the lock: the relationship between the values of the ghost variables and the real value in memory is part of the lock invariant, so we can break it freely while the lock is held, as long as we restore it before calling release.

When we are ready, we gather together all the shares of ghost variables that we hold, and use the viewshift_SEP tactic to update the ghost variables. This tactic is analogous to replace_SEP, but instead of proving $P \vdash Q$, we instead prove $P \Rrightarrow Q$ (written as `P |-- |==> Q` in ASCII). This *view shift* relation includes the ordinary derives relation, but also has a number of special rules that allow us to modify ghost state[6]. Of particular interest here is lemma ghost_var_update, which says that ghost_var Tsh v $p \Rrightarrow$ ghost_var Tsh v' p. As long as we have total ownership of a ghost variable, we can change its value to anything of the same type.

The ghost state logic performed in the viewshift_SEP is captured in the lemma ghost_var_incr. We begin by using the lemma bupd_frame_r to frame out the unused ghost variable. Then we use the lemma ghost_var_update′ to update the value of both halves of the remaining ghost variable. If one half had value x and the other had value n, then the update 1) tells us that $x = n$ and 2) changes the value of both halves to $n + 1$. Once this operation is complete in the body of incr, we have reestablished the lock invariant: the value of ctr has been changed from z to $z + 1$, and exactly one of x and y has been incremented to match. Because the frame depends on whether we passed in $g1$ or $g2$, we instantiate it before doing the case analysis on left; other than that, the proof is straightforward.

The proof of correctness of read is similar, but we do not need to do a view shift: instead, we use an ordinary assert_PROP to extract the values of both x and y, so that we can compute z. The only change we need to make to the proof for thread_func is to pass the extra arguments to incr, telling it that we are the thread holding $g1$ and its starting value is 0. The remaining interesting change is in the proof of main, where we need to create the ghost variables that we use in the rest of the program. We do this using a ghost_alloc tactic that takes the ghost assertion we want to allocate, minus its gname; the tactic allocates a new gname at which the assertion holds, which we can then introduce as usual with Intro. Once we allocate the two ghost variables with starting value 0, we can incorporate them into the lock invariants when we call makelock, and the rest of the proof proceeds as before. When we spawn the child thread, we pass it the gsh2 share of ghost variable $g1$ along with the shares of the locks, as its precondition now requires. When we reclaim its share of the ghost variable and call read, we can now use our half-shares of both ghost variables with value 1 to conclude that the value of t is 2.

[6]Formally, the view shift operator allows us to perform any *frame-preserving update* on ghost state, i.e., any change that could not invalidate any other thread's ghost state. We will discuss this idea further in Section 4.

4 Defining Custom Ghost State

4.1 The Structure of Ghost State

The ghost variables of the previous section are a special case of a much more general *ghost state* mechanism. With ghost variables, every thread that holds a share knows the exact value of the variable, but there are many other sharing patterns that may be used in concurrent programs. The key to defining a new pattern is to describe what happens when two elements are joined together, by creating an instance of the Ghost typeclass. Many useful instances can be found in `concurrency/ghosts.v`. An instance of the Ghost typeclass is a *separation algebra* with a join relation, plus a valid predicate marking those elements of the algebra that can be used in assertions. For instance, ghost variables of type A are drawn from a separation algebra over the type option $(share * A)$, where valid elements have nonempty shares. An element Some (sh, a) represents a share sh of value a, and None represents no ownership or knowledge of the variable. Two Some elements join by combining their shares, but only if they agree on the value; a None element joins with any other element and is the identity.

Every ghost state assertion is a wrapper around the predicate own g a pp, where g is a gname, a is an element of a Ghost instance, and pp is a separation logic predicate[7]. For instance, ghost_var sh v g is defined as own g (Some (sh, v)) NoneP. (For most kinds of ghost state, pp will be the empty predicate NoneP, but its inclusion also allows us to create *higher-order ghost state* [4].) The own predicate is governed by a few simple rules:

$$\text{own\_alloc}\,\frac{\mathsf{valid}\ a}{\mathsf{emp} \Rrightarrow \mathsf{EX}\ g : \mathsf{gname}, \mathsf{own}\ g\ a\ pp}$$

$$\text{own\_op}\,\frac{\mathsf{join}\ a1\ a2\ a3}{\mathsf{own}\ g\ a3\ pp = \mathsf{own}\ g\ a1\ pp * \mathsf{own}\ g\ a2\ pp}$$

$$\text{own\_valid\_2}\,\frac{}{\mathsf{own}\ g\ a1\ pp * \mathsf{own}\ g\ a2\ pp \Rrightarrow\ !!(\exists a3, \mathsf{join}\ a1\ a2\ a3 \wedge \mathsf{valid}\ a3)}$$

$$\text{own\_update}\,\frac{\mathsf{fp\_update}\ a\ b}{\mathsf{own}\ g\ a\ pp \Rrightarrow \mathsf{own}\ g\ b\ pp}$$

$$\text{own\_dealloc}\,\frac{}{\mathsf{own}\ g\ a\ pp \vdash \mathsf{emp}}$$

Of these rules, own_alloc and own_dealloc let us create and destroy ghost state, own_op lets us split and combine it according to its join relation, own_valid_2 tells us that any two pieces of ghost state that we hold at the same gname are consistent with each other, and own_update_ND lets us do *frame-preserving updates* to our ghost state: we can change its value arbitrarily as long as this does not invalidate any other piece of the same ghost state that might be held by another thread. Formally, fp_update $a\ b \triangleq \forall c, (\exists d, \mathsf{join}\ a\ c\ d \wedge \mathsf{valid}\ d) \rightarrow (\exists d, \mathsf{join}\ b\ c\ d \wedge \mathsf{valid}\ d)$.

The frame-preserving updates allowed by the join relation of each kind of ghost state determines what the ghost state can be used for. For instance, two pieces of a ghost variable only join if they have the same value; thus we can only change the value of a ghost variable when we have all its shares, because then we know that no other thread is restricting its value. Some ghost constructions allow smaller or older values to join with larger or newer ones, so that we can change a value without needing to update the records of all parties; others have extremely restrictive joins that ensure that a piece of ghost state belongs to only one thread at a time. Most concurrent programs can be verified with some combination of the types of ghost state defined in `ghosts.v`, but we are always free to define new Ghost instances for more complicated patterns of sharing and recording.

[7]More accurately, pp is of type preds, a dependent pair of a type signature (possibly including mpred) and a value of that type. This construction is used to embed predicates inside ghost state (as well as function pointers, lock invariants, etc.), which in turn can be the subject of predicates, without circular reference issues.

4.2 Example: `incr` with Unbounded Threads

We can put custom ghost state to use in further generalizing the `incr` example. The program `incrN.c` uses the same counter data structure, but creates more than two threads that access it simultaneously. We could give each one its own ghost variable, but we can write a simpler proof by recognizing that the value of the counter has nothing to do with which threads accessed it—each call to `incr` increments its value by 1, regardless of which thread calls `incr` or how many other threads have access to it. In other words, we should be able to track the counter's value with a single piece of ghost state that simply accumulates the number of calls to `incr`. In `verif_incr_gen.v`, we define custom ghost state to do exactly that, following the lead of Ley-Wild and Nanevski [7].

We begin by declaring an instance of the Ghost typeclass. A Ghost instance has three fields: a carrier type G, a predicate valid on G, and a join relation Join_G. It also has three proof obligations: it must be a separation algebra and a permission algebra, and validity of an element must imply validity of its sub-elements according to Join_G. To be a permission algebra, the join predicate must be functional, associative, commutative, and non-decreasing; to be a separation algebra, it must support a function core : $G \rightarrow G$ that, for each element, gives a unit for that element. These are not fundamental requirements for ghost state in general, but VST expects them to hold of the heap, and so it is convenient to impose them on ghost state as well.

For our example, we want to count the number of `incr` calls in two places. First, each time a thread calls `incr`, it should record that it has made a call. Second, the counter's lock invariant should record the total number of calls made, since that should also be the value of the counter. If we omit the latter record, then our ghost state will count the number of calls made, but there will be nothing to connect this number to the value of `ctr`. This is a common pattern for ghost state, which we call the *reference* pattern: each thread holds partial information describing its contribution to the shared state, and the shared resource holds a "reference" copy that records all of the contributions. We provide a function ref_PCM that makes such a reference structure for any Ghost instance. An element of ref_PCM is a pair of an optional contribution element ghost_part *sh* a and an optional reference element ghost_reference r, where a and r are drawn from the underlying Ghost instance, and *sh* is a nonempty share. To join two elements, we combine the shares and values of the contributions (if any), and require that the elements contain at most one reference between them (ensuring the uniqueness of the reference value). When a contribution element has the full share Tsh, it is guaranteed to be equal to the reference element, since this means we have collected all of the contributions. In general, we start by creating an initial contribution element and reference element, store the reference in the invariant of the shared data, and divide the contribution element into shares that we distribute to each thread. The contribution elements then record all the contributions of every thread, and when we rejoin them at the end of the program we learn exactly what all threads have done collectively and can deduce the state of the shared data. We will work this out in detail in the rest of the example; see `concurrency/ghosts.v` for the full list of lemmas about the ghost_part and ghost_reference assertions.

The underlying Ghost instance for the increment program is simply a nat recording the number of calls to `incr`. The join operation for the ghost is addition, and all numbers are valid. This sum_ghost instance is then passed to ref_PCM to make the reference ghost state we need. We also write some local definitions for the kinds of ghost state we expect to use: partial contributions (ghost_part), reference state (ghost_ref), and the combination of both (ghost_part_ref). (Using these definitions, which specialize the parametric definitions from `ghosts.v` to the sum_ghost instance, avoids relying on Coq to find the right Ghost instance for our ghost assertions.)

We are now ready to write the specifications for our functions. First, we note that every client of the counter should have both a share of the lock_inv assertion for the counter lock, and a share of the ghost_part assertion to record the number of increments this thread has observed.

We can make the proofs simpler by bundling up both these resources in a ctr_handle assertion. In more complex examples, clients may need many more resources to call data structure operations, and we can use this pattern to encapsulate details of the data structure's implementation. Now we can say that the `init_ctr` function creates a handle with full ownership Tsh and starting value 0, `dest_ctr` deallocates a full handle and guarantees that its value is the current value of the `ctr` field, and `incr` increments the value in a handle by 1, giving us a very neat summary of the operations on the counter data structure. We can also show that we can combine two ctr_handles by adding together their shares and their increment counts.

Proving the correctness of these specs involves correctly manipulating our new kind of ghost state. We allocate the ghost state in `init_ctr`, as a combination of total information $(\mathsf{Tsh}, 0)$ and reference element 0. This time, ghost_alloc leaves us with a subgoal: we need to show that our initial element is valid. For a ref_PCM instance, this means that the share of the thread contributions is nonempty (which Tsh is) and the contributions are *completable* to the reference element—i.e., there exists some remaining contribution that could join with the existing contributions to make the reference element. When the share is total and the two elements are equal, this is easy to prove. Now, when we release the counter lock, we establish its invariant by separating the reference copy from the contributions and giving it to the lock.

Conversely, in `dest_ctr`, we must relate the total contributions to the value of the counter. The calling thread passes in a total contribution element $\mathsf{ghost\_part}(\mathsf{Tsh}, v)$, and from the lock invariant we receive $\mathsf{ghost\_ref}(z)$, where z is also the value of `ctr`. Given these two pieces, we can use the lemma ref_sub (which is derived from the validity rule own_valid_2 of general ghost state) to conclude that $z = v$, exactly as desired.

In `incr`, after incrementing the `ctr` field, we want to simultaneously add 1 to the caller's contribution and the reference ghost state. To do this, we need to show that this addition is a frame-preserving update. Fortunately, ref_PCM comes with a lemma ref_add for doing just this kind of update: we can add any piece of ghost state to both the contribution and the reference. In general, when we define a new kind of ghost state, we will prove lemmas describing its common forms of frame-preserving update; in the absence of these lemmas, we can use the generic own_update rule and work with the definition of frame-preserving update directly.

The proof of `thread_func` is very similar to the previous versions, but `main` is slightly more complicated than before, illustrating common patterns for reasoning about programs that spawn several threads performing the same operations. We begin by calling `init_ctr` to make the counter lock and the ghost variables. Then, because each of the N threads will need a share of the ctr_handle and the `lock` field, we use the split_shares lemma to divide both Tsh and Ews into $N + 1$ pieces, one for each spawned thread and one retained by the parent. In the first loop, we give each thread its resources: a share of the `lock` field, a share of the counter handle with initial value 0, and half of a thread lock for joining. We store each of the locks in the `thread_lock` array, so that we can join with each of the spawned threads later. In the second loop, we reverse the process, joining with each thread and reclaiming its shares—but since each thread we join with has completed its body, each reclaimed counter handle now has a value of 1 instead of 0. Adding these together, we get full ownership of a ctr_handle with value N, so it is easy to show that after we call `dest_ctr` the value of the counter is N.

5 Basic Rules of Concurrent Separation Logic

5.1 Lock Specifications

These specifications can be found in `concurrency/lock_specs.v`, except for the recursive self variants, which are in `atomics/verif_lock.v`.

$$\{\mathsf{mem\_mgr}\ gv\}\ \texttt{makelock()}\ \{\ell.\ \mathsf{lock\_inv}\ \mathsf{Tsh}\ \ell\ (R\ \ell)\}$$

Note that the R passed to `makelock` is a function from the lock handle to the invariant, so that the invariant can reference the lock's own name (as in $\mathsf{selflock}$).

$$\{!!sh \neq \mathsf{Share.bot} \wedge \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \mathtt{acquire}(\mathsf{ptr\_of}\ \ell)\ \{R * \mathsf{lock\_inv}\ sh\ \ell\ R\}$$

The specs for `release` and `freelock` have several variants, to allow for things like recursive locks that contain parts of their own $\mathsf{lock\_inv}$ assertions. Their top-level specs are written in a confusing but flexible style that implies both normal and recursive subspecifications:

$$\{!!sh \neq \mathsf{Share.bot} \wedge (\mathsf{exclusive}\ R) * \triangleright \mathsf{lock\_inv}\ sh\ \ell\ R * P * (\mathsf{lock\_inv}\ sh\ \ell\ R * P \mathrel{-\!\!*} Q * R)\}\ \mathtt{release}(\mathsf{ptr\_of}\ \ell)\ \{Q\}$$

$$\{\mathsf{lock\_inv}\ \mathsf{Tsh}\ \ell\ R * P * ((P * R \vdash \mathsf{FF}) \wedge \mathsf{emp})\}\ \mathtt{freelock}(\mathsf{ptr\_of}\ \ell)\ \{P\}$$

In nonrecursive use, the "unknown precondition" P is simply the invariant R, yielding the familiar specs with subspec names $\mathsf{release\_simple}$ and $\mathsf{freelock\_simple}$:

$$\{!!sh \neq \mathsf{Share.bot} \wedge (\mathsf{exclusive}\ R) * R * \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \mathtt{release}(\mathsf{ptr\_of}\ \ell)\ \{\mathsf{lock\_inv}\ sh\ \ell\ R\}$$

$$\{(\mathsf{exclusive}\ R) * R * \mathsf{lock\_inv}\ \mathsf{Tsh}\ \ell\ R\}\ \mathtt{freelock}(\mathsf{ptr\_of}\ \ell)\ \{R\}$$

In recursive use, it is instead the nonrecursive part of the invariant, allowing us to give up the $\mathsf{lock\_inv}$ assertion into the invariant (subspec names $\mathsf{release\_self}$ and $\mathsf{freelock\_self}$):

$$\{R * \mathsf{lock\_inv}\ sh\ \ell\ (\mathsf{selflock}\ R\ sh\ \ell))\}\ \mathtt{release}(\mathsf{ptr\_of}\ \ell)\ \{\mathsf{emp}\}$$

$$\{!!(sh1 + sh2 = sh) \wedge (\mathsf{exclusive}\ R) * \mathsf{selflock}\ R\ sh2\ \ell * \mathsf{lock\_inv}\ sh1\ \ell\ (\mathsf{selflock}\ R\ sh2\ \ell)\}\ \mathtt{freelock}(\mathsf{ptr\_of}\ \ell)\ \{R\}$$

5.2 Spawn

This specification can be found in `concurrency/semax_conc.v`.

$$\{P(y) * (f : x.\ \{P(x)\}\{\mathsf{emp}\})\}\ \mathtt{spawn}(f, y)\ \{\}$$

5.3 Ghost Operations

These rules can be found in `msl/ghost_seplog.v`.

$$\mathsf{own\_alloc}\ \frac{\mathsf{valid}\ a}{\mathsf{emp} \Rrightarrow \mathsf{EX}\ g : \mathsf{gname}, \mathsf{own}\ g\ a\ pp}$$

$$\mathsf{own\_op}\ \frac{\mathsf{join}\ a1\ a2\ a3}{\mathsf{own}\ g\ a3\ pp = \mathsf{own}\ g\ a1\ pp * \mathsf{own}\ g\ a2\ pp}$$

$$\mathsf{own\_valid\_2}\ \frac{}{\mathsf{own}\ g\ a1\ pp * \mathsf{own}\ g\ a2\ pp \Rrightarrow !!(\exists a3, \mathsf{join}\ a1\ a2\ a3 \wedge \mathsf{valid}\ a3)}$$

$$\mathsf{own\_update\_ND}\ \frac{\mathsf{fp\_update\_ND}\ a\ B}{\mathsf{own}\ g\ a\ pp \Rrightarrow \mathsf{EX}\ b, !!(B\ b)\ \&\&\ \mathsf{own}\ g\ b\ pp}$$

$$\mathsf{own\_update}\ \frac{\mathsf{fp\_update}\ a\ b}{\mathsf{own}\ g\ a\ pp \Rrightarrow \mathsf{own}\ g\ b\ pp}$$

$$\mathsf{own\_dealloc}\ \frac{}{\mathsf{own}\ g\ a\ pp \Rrightarrow \mathsf{emp}}$$

6 Global Invariants and Atomic Operations

6.1 Invariants and Fancy Updates

One of the most powerful applications of ghost state is in defining "global invariants", which are similar to lock invariants but are not associated with any particular memory location. Instead, a global invariant is true before and after every step of a program, acting as a publicly accessible resource. We have already seen how to use the viewshift_SEP tactic to change the value of ghost state; we can also use it to interact with global invariants. When we call viewshift_SEP with current assertion P and target assertion P', we get a goal of the form $P \vdash {\models\!\!\Rrightarrow_\top} P'$, where ${\models\!\!\Rrightarrow_\top}$ is a "fancy update" operator parameterized by a set of enabled invariants (the full set $\top$ by default). This allows us to create, open, and close invariants in order to prove P', as long as we never open the same invariant twice and always end with all invariants closed. The primary rules for manipulating invariants are:

$$\text{inv\_alloc}\ \frac{}{\triangleright P \vdash {\models\!\!\Rrightarrow_E}\ \mathsf{EX}\ i : \mathsf{iname}, \boxed{P}^{i}}$$

$$\text{inv\_dup}\ \frac{}{\boxed{P}^{i} = \boxed{P}^{i} * \boxed{P}^{i}}$$

$$\text{inv\_open}\ \frac{i \in E}{\boxed{P}^{i} \vdash {}^{E}{\models\!\!\Rrightarrow}^{E\setminus i}\ \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{E\setminus i}{\models\!\!\Rrightarrow}^{E}\ \mathsf{emp})}$$

where $\boxed{P}^{i}$ is written in Coq as invariant i P. We can put any resources we currently own into an invariant, which can then be freely duplicated and shared between threads—but can only be accessed through view shifts (e.g., between program steps). During a view shift, we can open any enabled invariant, but must close it again before taking any steps of execution (except for those that are explicitly marked as atomic; more on this in Section 6.2). A global invariant effectively turns its contents into a public resource, freely accessible by all threads as long as they maintain it in its current state at all times (except instantaneously during view shifts).

6.2 Atomic Operations

A program instruction can use the contents of a global invariant if it can guarantee that no one will ever see an intermediate state in which the invariant does not hold. This means that *atomic* operations can freely access invariants: for instance, if a global invariant holds full ownership of a memory location `x`, then an atomic operation can read or write the value of `x`. Theoretically, an atomic operation is any operation that is guaranteed by the language to not result in any visible intermediate states. In C (as of the C11 standard), there is a set of functions that explicitly perform atomic memory operations: `atomic_load`, `atomic_store`, etc. We can give separation logic rules for these operations, building on the idea that they are like the corresponding nonatomic operations but can also access invariants, and then use those rules to prove correctness of C programs that use them (i.e., lock-free concurrent programs).

The generic proof rule for atomic operations in Iris is:

$$\text{atomic consequence}\ \frac{P\ {}^{E}{\Rrightarrow}^{E\setminus E'}\ P' \qquad \{P'\}\ e\ \{Q'\} \qquad Q'\ {}^{E\setminus E'}{\Rrightarrow}^{E}\ Q \qquad e \text{ atomic}}{\{P\}\ e\ \{Q\}}$$

In other words, we can open a set of invariants E' (using the rule inv_open from section 6.1), prove a triple for the atomic operation e using the resources from the invariants, and then restore them. In C, we have a small fixed set of atomic operations, so instead of adding an atomic predicate to the definition of the language, we can just instantiate this rule for each atomic operation. For instance, we know that the rule for a load is $\{x \mapsto v\}$ `*x` $\{v.\ x \mapsto v\}$. Plugging this into the Hoare triple in the consequence rule gives us:

$$\text{atomic load}\ \frac{P\ {}^{E}{\Rrightarrow}^{E\setminus E'}\ x \mapsto_{\mathrm{a}} v * R \qquad x \mapsto_{\mathrm{a}} v * R\ {}^{E\setminus E'}{\Rrightarrow}^{E}\ Q}{\{P\}\ \texttt{atomic\_load}(x)\ \{v.\ Q\}}$$

where $\mapsto_{\mathrm{a}}$ is a special atomic points-to assertion (written as $\mathsf{atomic\_int\_at}$ or $\mathsf{atomic\_ptr\_at}$ in VST), and we use the frame R to store everything that isn't the memory location being accessed. We can get around the need to supply the frame by using magic wand instead:

$$\text{atomic load}\ \frac{P\ {}^{E}{\Rrightarrow}^{E\setminus E'}\ x \mapsto_{\mathrm{a}} v * (x \mapsto_{\mathrm{a}} v \mathrel{-\!\!*} {}^{E\setminus E'}{|\!\!\Rrightarrow}^{E}\ Q)}{\{P\}\ y = \texttt{atomic\_load}(x)\ \{v.\ Q\}}$$

$$\text{atomic store}\ \frac{P\ {}^{E}{\Rrightarrow}^{E\setminus E'}\ x \mapsto_{\mathrm{a}} \_ * (x \mapsto_{\mathrm{a}} v \mathrel{-\!\!*} {}^{E\setminus E'}{|\!\!\Rrightarrow}^{E}\ Q)}{\{P\}\ \texttt{atomic\_store}(x, v)\ \{Q\}}$$

Now we can see that if an invariant I contains a points-to assertion $\texttt{x} \mapsto_{\mathrm{a}} v$, we can access and modify the value of `x` using atomic operations (and *only* atomic operations). The VST specifications corresponding to these rules can be found in `atomics/SC_atomics_base.v`.

Note that, as the name `SC_atomics` suggests, these are rules for *sequentially consistent* atomic operations: weaker memory orders have more complicated rules, since a thread is not guaranteed to see an objective "current value" held in the invariant.

6.3 Cancelable Invariants

The invariants of Section 6.1 have a major disadvantage in a memory-managed language like C: they can never be deallocated. If we put a points-to assertion into an invariant, it must be in the invariant for the entire rest of the program, and there is no way to retrieve and deallocate it. However, there is a simple workaround: instead of putting a memory assertion P into an invariant, we can instead put $P \vee G$, where G is an exclusive ghost state token. After we allocate the invariant, we hold G, so whenever we access the invariant we can prove that it contains P. Once we are done with the invariant, we can put G into it and take P out, so that while $P \vee G$ is still true, it only holds ghost resources G and we are free to deallocate the memory resources P. Of course, we want our invariant to be accessed by multiple threads simultaneously, so it should be possible to divide G into shares and distribute it among threads; only once we have recollected all its pieces can we trade it for P and stop using the invariant. This is the *cancelable invariant* trick [2] that makes invariants usable in malloc-free languages.

Cancelable invariants are defined in VST in `concurrency/cancelable_invariants.v`. The ghost resource G is defined as $\mathsf{cinv\_own}\ g\ \mathsf{Tsh}$, where $\mathsf{cinv\_own}$ is custom ghost state whose carrier type is a share and whose join operation is the ordinary share join; a cancelable invariant is defined as $\mathsf{cinvariant}\ i\ g\ P \triangleq \mathsf{invariant}\ i\ (P \vee \mathsf{cinv\_own}\ g\ \mathsf{Tsh})$. The relevant rules are:

$$\mathsf{cinv\_alloc}\ \frac{}{\triangleright P \vdash\ {|\!\!\Rrightarrow}_{E}\ \mathsf{EX}\ i\ g, \mathsf{cinvariant}\ i\ g\ P * \mathsf{cinv\_own}\ g\ \mathsf{Tsh}}$$

$$\mathsf{cinv\_open}\ \frac{sh \neq \mathsf{Share.bot} \wedge i \in E}{\mathsf{cinvariant}\ i\ g\ P * \mathsf{cinv\_own}\ g\ sh \vdash {}^{E}{|\!\!\Rrightarrow}^{E\setminus i}\ \triangleright P * \mathsf{cinv\_own}\ g\ sh * (\triangleright P \mathrel{-\!\!*} {}^{E\setminus i}{|\!\!\Rrightarrow}^{E}\ \mathsf{emp})}$$

$$\mathsf{cinv\_cancel}\ \frac{i \in E}{\mathsf{cinvariant}\ i\ g\ P * \mathsf{cinv\_own}\ g\ \mathsf{Tsh} \vdash\ {|\!\!\Rrightarrow}_{E}\ \triangleright P}$$

The operations on cancelable invariants closely correspond to those of lock invariants: we can allocate them, split them into shares, pass them to different threads, recollect them, and deallocate them. The difference is that cancelable invariants can only be accessed instantaneously between program steps and by atomic operations, while lock invariants can be accessed in the space between `acquire` and `release` calls. In fact, we can use cancelable invariants to implement lock invariants, and this is exactly what we do in `atomics/verif_lock.v`. For a lock at location ℓ that protects resources R, its cancelable invariant is

$$\mathsf{inv\_for\_lock}\ \ell\ R \triangleq \exists b.\ \ell \mapsto_{\mathrm{a}} b * \text{if } b \text{ then } \mathsf{emp} \text{ else } R$$

When the lock is held (i.e., $\ell \mapsto_{\mathrm{a}} \mathsf{true}$) the thread that holds it also owns R; when the lock is not held ($\ell \mapsto_{\mathrm{a}} \mathsf{true}$) then R must be in the invariant. In `verif_lock.v`, we use inv_for_lock to prove that a simple atomic-operation-based spinlock implements the lock specifications of Section 5.1. This is the standard lock implementation for concurrent VST programs.

7 Iris in VST

The ghost state, invariants, and view shifts described in this manual are heavily influenced by Iris, a language-independent separation logic framework. In fact, VST's logic is provably an instance of the Iris framework. Doing this instance proof lets us use many Iris definitions and tactics directly in VST, and makes it much easier to deal with complicated view shifts and derived concepts like logical atomicity. This section assumes basic familiarity with the features of Iris; if you have not used it before and are interested in writing proofs of correctness for complicated concurrent programs, we recommend working through Elizabeth Dietrich's beginner's guide [3] first. Also note that we only explain how to *use* the Iris features in VST: for implementation details, please see the technical report on arXiv [8].

User beware: Iris and VST are both large and complicated Coq developments, and combining them can lead to notational oddities, universe inconsistencies, and other unpredictable issues. Please report any issues encountered by creating an issue on GitHub or emailing `mansky1@uic.edu`. In particular, Iris is built on top of std++, a variant standard library for Coq, and importing any of its files will cause significant changes to your proof environment (new definitions of standard list functions, unicode symbols for $\forall$ and $\exists$, λ-notation for anonymous functions, and many others). You do not need to write your own definitions in this style, and you may find it easier to read once you get used to it, but it is definitely a change from vanilla Coq!

7.1 Iris Proof Mode

One of the best features of Iris is Iris Proof Mode (also called MoSeL), a set of tactics for doing separation logic proofs in the same style as standard Coq proofs [6]. The tactics are described at `https://gitlab.mpi-sws.org/iris/iris/blob/master/docs/proof_mode.md`, and tutorials and examples are available on the Iris Project website (`https://iris-project.org`). Fortunately, the proof mode is defined in a very generic way, and can be used for any separation logic that is proved to be an instance of Iris's formulation of the logic of bunched implications (BI). The file `veric/bi.v` contains a proof that Verifiable C is such a logic, allowing us to use IPM/Mosel in VST proofs.

Iris has its own framework for language semantics and associated symbolic execution engine, but it is most naturally suited to functional languages and generally less automatic than VST's forward. On the other hand, the Iris tactics for reasoning about separation logic implications give a much higher degree of control than cancel and entailer, and are more fully featured than sep_apply. IPM also has a smooth treatment of "modalities" such as the basic and fancy updates; while in VST we might need to explicitly apply lemmas about the monotonicity, transitivity, and frame properties of $\Rrightarrow$, in IPM we can often eliminate them automatically. Importing `VST.veric.bi` (or any of the other files mentioned in this section, all of which export `bi`) gives access to IPM for any separation logic entailment: simply use the iIntros tactic, and the goal will immediately be converted into an Iris proof state where all Iris tactics can be applied. There is also an iVST tactic for switching back to VST's style of entailment if needed.

7.2 Invariants and Namespaces

Invariants in Iris have slightly different proof rules from the ones in Section 6.1: in inv_alloc, instead of obtaining the new invariant at some existentially quantified name i, the user chooses a *namespace* of the form nroot .@ *name* (where *name* is a string) and obtains the invariant at that namespace. Namespaces can also be further qualified (nroot .@ name_1 .@ name_2 .@ ...). This makes it easier to open invariants at the same time: instead of tracking that two arbitrary i's are different from each other, we can simply observe that two concrete strings are not equal. VST supports this mechanism as well, in `concurrency/invariants.v`, giving exactly the same interface to invariants as in Iris.

VST's invariants also satisfy the typeclasses necessary to use Iris's tactics for invariants. Invariants are *persistent*, and can be introduced/destructed with the `#` pattern to put them in the persistent context, so that they are automatically replicated as needed (at least during the IPM section of a proof). The iInv tactic can be used to open invariants: instead of explicitly invoking inv_open, users can (and should) write tactics like iInv N as `"H" "Hclose"` to open an invariant with name N (either namespace or hypothesis name), call its contents `"H"`, and call the closing viewshift `"Hclose"`.

7.3 Logical Atomicity

In Iris, global invariants and fancy updates are used to implement the *logically atomic triples* introduced in the TaDA logic [1]. Logically atomic triples are a strong specification for concurrent data structures, expressing the idea that an operation appears to take effect instantaneously at a linearization point, and no intermediate states are visible. Logically atomic operations *appear to execute atomically*, and so they can access the contents of invariants, exactly as if they were atomic operations in the sense of Section 6.2.

The general form of an atomic triple is

$$\forall a.\ \langle P_l \mid P_p(a)\rangle\ c\ \langle Q_l \mid Q_p(a)\rangle$$

where P_l and Q_l are *private* or *local* pre- and postconditions, and P_p and Q_p are *public* pre- and postconditions, parameterized by an abstract value a. The private pre- and postconditions work in the same way as in an ordinary Hoare triple, but the public ones are different: P_p must be true at *every point* from the beginning of the function until the linearization point, for some value of a that is not known to the function and may change without notice, and Q_p must become true at the linearization point (for the value of a in force at that point), but may not still be true by the end of the function. More concisely, P_p is true until the linearization point, at which point the function atomically transitions from P_p to Q_p, and then continues to execute (without changing the data in the public pre/postcondition) until it returns.

In general, the public pre- and postcondition will hold the abstract state of a data structure, which may change as it is accessed by other threads; the private pre- and postcondition will hold the local information that a thread needs to access the structure. For instance, an atomic specification for a map insert function might look like

$$\forall m.\ \langle \mathsf{is\_map}(p) \mid \mathsf{map\_state}(p, m)\rangle\ \mathtt{insert}(p, x, v)\ \langle \mathsf{is\_map}(p) \mid \mathsf{map\_state}(p, m[x \mapsto v])\rangle$$

We are not guaranteed that the map state at the end of the insertion will be $m[x \mapsto v]$ where m is the map state when `insert` is called: rather, we know that p always holds *some* map during the function's execution, and at some point the function will take that map m, add $x \mapsto v$, and then eventually return (and in the meantime other threads may have further modified the map). If we provide a similar specification for `lookup`, then we will have specified a linearizable concurrent map, whose behavior in an execution always matches that of some linear sequencing of the lookups and inserts performed during the execution.

In atomics/general_atomics.v, we use Iris's definition of atomic updates to build atomic specifications in VST. We provide an atomic variant of the funspec notation:

```
ATOMIC TYPE W OBJ a INVS E
WITH ...
PRE [ ... ]
  PROP (...)
  LOCAL (...)
  SEP (P_l) | (P_p)
POST [ ... ]
  PROP ()
  LOCAL ()
  SEP (Q_l) | (Q_p)
```

where W is the TypeTree representing the type of the WITH clause; a is the quantified abstract state for the triple; E is the mask (set of invariants) needed to implement the triple (often empty), and the public and private pre- and postconditions are as explained above. Atomic triples will always need to be declared with Program Definition, and will have a number of obligations related to nonexpansiveness, but in the current version of VST these should almost always be solved automatically. The notation for atomic specs is slightly brittle, and reasonable-looking specs will sometimes fail to parse: if you encounter this, please create a GitHub issue or email mansky1@uic.edu.

When **proving** that a function implements an atomic specification, the precondition will contain an atomic_shift assertion with the public pre- and postcondition inside it. This atomic shift is a wrapper around Iris's atomic_update (AU), and can be opened with the iMod tactic. It acts similarly to an invariant, except that it always has a choice of **two** closing view shifts: an *aborting* view shift that can be used when the public precondition is maintained, and restores the atomic shift; and a *committing* view shift that can be used when the public postcondition is satisfied, and returns a black-box assertion Q that is required in order to prove the function's postcondition. We can abort the atomic shift any number of times, treating the public precondition like an invariant, but in order to complete the proof of the function's atomic specification, we must always do a commit to obtain Q, after which we lose the atomic shift and can no longer access the public precondition.

We can **use** an atomic triple with the usual forward_call tactic. The witness to forward_call needs to have one additional element: the desired postcondition Q. As usual, forward_call will generate an obligation to prove the precondition of the spec: in this case, the precondition includes both the private precondition P_l and the atomic shift itself. Usually the client will have an invariant containing the abstract state of the data structure (i.e., the public precondition) as well as additional information (e.g., the history in the case of linearizability); proving the atomic shift will then involve proving that the contents of the invariant imply the public precondition, and that the public postcondition can be used to re-establish the invariant while also proving Q. Iris's iAuIntro tactic can be used to open atomic shift proof obligations, turning them into a conjunction of committing and aborting view shifts.

One of the simplest use cases for atomic specifications is locks. Instead of the usual lock-invariant-based specs for locks, atomic triples let us state much simpler specs:

$$\forall b.\ \langle \mathsf{lock\_state}(\ell, b)\rangle\ \mathtt{acquire}(\ell)\ \langle b = \mathsf{false} \wedge \mathsf{lock\_state}(\ell, \mathsf{true})\rangle$$

$$\langle \mathsf{lock\_state}(\ell, \mathsf{true})\rangle\ \mathtt{release}(\ell)\ \langle \mathsf{lock\_state}(\ell, \mathsf{false})\rangle$$

where $\mathsf{lock\_state}(\ell, \mathsf{true})$ means that ℓ is held by some thread, and $\mathsf{lock\_state}(\ell, \mathsf{false})$ means that it is not. An acquire takes a lock in an unknown state, and at some point when it is not held, atomically sets it to held; a release atomically releases a held lock. These specifications say nothing about who owns ℓ or what resources it protects, giving us more flexibility in our reasoning: we can put the lock assertion into a global invariant instead of passing out shares among threads, and we can create a lock in one place and attach an invariant to it in another. These specs also imply the ordinary lock specs. In `atomics/verif_lock_atomic.v`, we re-verify our lock implementation against these specs (with $\mathsf{lock\_state}(\ell, b)$ implemented by an atomic points-to $\ell \mapsto_{\mathrm{a}} b$), and provide a range of subspecifications for using the locks with invariants, atomically or nonatomically.

We use the atomic lock specifications to prove atomic specifications for the counter example as well. In `progs64/verif_incr_atomic.v`, we use atomic specifications to avoid the need to choose a kind of ghost state in advance when verifying the `incr` and `read` functions: instead, we prove triples that look like

$$\langle \mathsf{c.lock} \mapsto \ell \mid \mathsf{ctr\_state}\ gv\ \ell\ n\ g \rangle\ \texttt{incr()}\ \langle \mathsf{c.lock} \mapsto \ell \mid \mathsf{ctr\_state}\ gv\ \ell\ (n+1)\ g \rangle$$

that precisely capture the behavior of the function (`incr` atomically adds 1 to the value of the counter). When we acquire and release the counter's lock in `incr` and `read`, we use the atomic invariant-based subspecs $\mathsf{acquire\_inv}$ and $\mathsf{release\_inv}$, which allow us to find the lock invariant in the public state (i.e., the precondition of the $\mathsf{atomic\_shift}$) instead of requiring the calling thread to own it. In the client, we create an invariant

$$\boxed{\mathsf{EX}\ x\ y,\ \mathsf{ghost\_var\ gsh1}\ x\ g_1 * \mathsf{ghost\_var\ gsh1}\ y\ g_2 * \mathsf{ctr\_val}\ g\ (x+y)}$$

and use it to call the atomic specifications. For instance, when thread 1 performs an increment, it starts with the invariant and its own ghost variable $\mathsf{ghost\_var\ gsh2}\ 0\ g_1$, and proves that the atomic spec provided by `incr` leads to a postcondition $\mathsf{ghost\_var\ gsh2}\ 1\ g_1$ (while re-establishing the invariant). Different clients with different invariants and ghost state could call the same specs for `incr` and `read`, as long as their invariants included the $\mathsf{ctr\_val}$ assertion (the abstract state of the counter) used in the public pre- and postconditions. We believe that this is the strongest possible specification for the increment data structure (such as it is): rather than tying it to a specific kind of ghost state based on the expected use, we simply show that `incr` increases a value by 1 and `read` reads a value, atomically, and then allow clients to build protocols on top of this functionality as they see fit.

A larger and more compelling use case for atomic specifications is for *fine-grained* concurrent data structures, which use multiple locks and/or atomic pointers for different sections of the data structure, allowing multiple threads to perform unrelated operations simultaneously. There is a full example of this approach in `atomics/verif_hashtable_atomic.v`, a verification of a simple lock-free hashtable implemented with SC atomic operations. We prove atomic triples of the form

$$\langle H.\ h \mapsto \vec{p} \mid k \neq 0 \land \mathsf{hashtable}\ H\ \vec{p}\ \vec{g} \rangle\ \texttt{set\_item}(h,k,v)\ \langle h \mapsto \vec{p} \mid \mathsf{hashtable}\ H[k \mapsto v]\ \vec{p}\ \vec{g} \rangle$$

$$\langle H.\ h \mapsto \vec{p} \mid k \neq 0 \land \mathsf{hashtable}\ H\ \vec{p}\ \vec{g} \rangle\ \texttt{get\_item}(h,k)\ \langle v.\ h \mapsto \vec{p} \mid H(k) = v \land \mathsf{hashtable}\ H\ \vec{p}\ \vec{g} \rangle$$

showing that the hashtable functions atomically perform the desired operations on the global state of the hashtable, and then attach an invariant asserting that the current state of the hashtable can be reconstructed from a ghost-state history of all operations performed, effectively proving *linearizability* of the hashtable.

References

[1] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 207–231, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[2] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proc. ACM Program. Lang.*, 4(POPL):34:1–34:29, 2020.

[3] Elizabeth Dietrich. A beginner guide to Iris, Coq and separation logic. *CoRR*, abs/2105.12077, 2021.

[4] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 256–269, New York, NY, USA, 2016. Association for Computing Machinery.

[5] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 637–650, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 205–217, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, page 561–574, New York, NY, USA, 2013. Association for Computing Machinery.

[8] William Mansky. Bringing Iris into the Verified Software Toolchain. *CoRR*, abs/2207.06574, 2022.

[9] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271 – 307, 2007. Festschrift for John C. Reynolds's 70th birthday.