Verifying Concurrent Programs with VST

William Mansky

1 Introduction

As of version 2.1, VST includes support for verifying concurrent programs with user-defined ghost state, as in modern concurrent separation logics. This document describes how to use Verifiable C to prove the correctness of concurrent programs, using examples from the `progs` folder that ships with VST. We will assume familiarity with the basics of VST, as described in the VST manual.

2 Verifying a Concurrent Program with Locks

A concurrent C program 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. In Verifiable C, synchronization is provided by a *lock* data structure, which supports functions `acquire` and `release`. Each lock in a program 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. Locks can be used to enforce *mutual exclusion*, ensuring that a memory location is only accessed by one thread at a time. VST ships with a C header file `threads.h` that declares the concurrency primitives (locks and thread creation), and should be `#include`d in any Verifiable C concurrent program.

The file `progs/incr.c` contains a simple concurrent C program. It has a global integer variable `ctr` that is used as shared data, with two accessor functions: `incr`, which increases the value of `ctr` by one, and `read`, which reads the current value of `ctr`. These functions use the lock `ctr_lock` to synchronize access to `ctr`. This synchronization is necessary because `incr` changes the value of `ctr`: if a thread tries to access a memory location while another thread writes to that location, a *data race* occurs, leading to unpredictable (formally, *undefined*) behavior. This is considered an error in C. This is reflected in Verifiable C by the existence 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. If we want to have a memory location that can be modified by multiple threads, we must move shares between threads via locks, as described below. A consequence of this 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 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, it imports `VST.progs.conclib`, a library of lemmas and tactics that are useful for concurrent program verification. It then declares specifications for the built-in concurrency primitives of VST. Their specifications are already defined in `concurrency/semax_conc.v`, so we only need to associate them with their function identifiers. We will go through the specifications of each concurrency primitive in the following sections; the concurrent separation logic rules are summarized in Section 6.

The first thing we need to do to verify functions on `ctr` is to define a *lock invariant*, a predicate describing the resources protected by the lock `ctr_lock`. A lock invariant can be any Verifiable C assertion (i.e., `mpred`), subject to a condition described later. In this case, the lock protects the data in `ctr`. We want to know specifically that `ctr` always contains an unsigned integer value, so we use the lock invariant cptr_lock_inv $\triangleq$ EX z : Z, data_at tuint (Vint(Int.repr z)) `ctr`. We use the lock_inv predicate to assert that a lock exists in memory with a given invariant: lock_inv sh p R means that the current thread owns share sh of a lock at location p with invariant R. Shares of a lock can be combined and split in the same way as shares of data_at, and any readable share is enough to acquire or release the lock[1].

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

```
DECLARE _incr
 WITH ctr : val, sh : share, lock : val
 PRE [ ]
   PROP (readable_share sh)
   LOCAL (gvar _ctr ctr; gvar _ctr_lock lock)
   SEP (lock_inv sh lock (cptr_lock_inv ctr))
 POST [ tvoid ]
   PROP () LOCAL () SEP (lock_inv sh lock (cptr_lock_inv ctr))

DECLARE _read
 WITH ctr : val, sh : share, lock : val
 PRE [ ]
   PROP (readable_share sh)
   LOCAL (gvar _ctr ctr; gvar _ctr_lock lock)
```

[1] 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 (lock_inv sh lock (cptr_lock_inv ctr))
  POST [ tuint ]
   EX z : Z,
    PROP ()
    LOCAL (temp ret_temp (Vint (Int.repr z)))
    SEP (lock_inv sh lock (cptr_lock_inv ctr))
```

These are surprisingly boring specifications! The `read` function needs to know that the lock exists, and returns some number, about which we know nothing; the `incr` function does even less, taking the lock_inv assertion and returning it as is. These are enough to prove *safety* of the program, to show that it is a valid C program, but not enough to learn much about what the program actually computes. This is a product 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 4. 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.

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:

$$\{!!\mathsf{readable\_share}\ sh \land \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \mathtt{acquire}(\ell)\ \{R * \mathsf{lock\_inv}\ sh\ \ell\ R\}$$

$$\{!!(\mathsf{readable\_share}\ sh \land \mathsf{exclusive}\ R) * R * \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \mathtt{release}(\ell)\ \{\mathsf{lock\_inv}\ sh\ \ell\ R\}$$

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 $z + 1$.

3 Thread Creation and Joining

Every C program starts its execution as a single-threaded program. It becomes concurrent when it calls an external function that spawns a new thread, such as with Verifiable C's spawn function. The spawn 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:

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

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 selflock function, as we can see in the definition of thread_lock_inv, and use it with the lemma selflock_eq: $\forall Q\ sh\ p, \mathsf{selflock}\ Q\ sh\ p = Q * \triangleright\mathsf{lock\_inv}\ sh\ p\ (\mathsf{selflock}\ Q\ sh\ p)$.

In Verifiable C, functions that will be passed to spawn must have specifications of a certain form, as in thread_func_spec:

```
DECLARE _thread_func
 WITH y : val, x : val * share * val * val
 PRE [ _args OF (tptr tvoid) ]
        let '(ctr, sh, lock, lockt) := x in
        PROP  (readable_share sh)
        LOCAL (temp _args y; gvar _ctr ctr; gvar _ctr_lock lock;
               gvar _thread_lock lockt)
        SEP   (lock_inv sh lock (cptr_lock_inv ctr);
               lock_inv sh lockt (thread_lock_inv sh ctr lock lockt))
 POST [ tptr tvoid ]
        PROP () LOCAL () 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, LOCAL must hold a temp for the argument followed by zero or more global variables. The PROP and SEP clauses are unrestricted. The postcondition must be completely empty, reflecting the separation logic rule for `spawn`. In this example, the thread function takes a readable share of the lock protecting `ctr`, along with the same share of a recursive lock for joining; the pointers to the locks and to `ctr` are taken from global variables, while the argument to the function is ignored entirely. The proof of this specification is straightforward until we reach the last line, where the spawned thread releases its lock (using the `release2` function, which is specialized to recursive locks). At that point, we unroll the definition of selflock and show that the invariant is satisfied by precisely all the resources held by the thread.

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:

$$\{!!\mathsf{writable\_share}\ sh \land \ell \overset{sh}{\longmapsto} \_\}\ \texttt{makelock}(\ell)\ \{\mathsf{lock\_inv}\ sh\ \ell\ R\}$$

To make a location into a lock, all we need is a writable share of the location. We do *not* need to know that the invariant R holds; 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). In Verifiable C, the location to be converted into a lock needs to point to a memory block of the appropriate size, so the caller must provide the predicate data_at_ sh tlock ℓ.

Next, we divide the locks into shares: one for the spawned function, and one retained by `main`. The file `progs/conclib.v` includes lemmas that for splitting shares into readable pieces: the key ones are split_readable_share (of which split_Ews is a special case) and split_shares (which produces a list of shares of any desired length).

Next, we spawn the child thread using the forward_spawn tactic, a `spawn`-specific wrapper around forward_call. Its general form is forward_spawn *idarg* *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 each of the locks and provide it to the spawned thread to satisfy the precondition of `thread_func`, while retaining the other share 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 locks, regaining full ownership. This allows us to deallocate the locks with calls to `freelock` (for the non-recursive `ctr` lock) and `freelock2` (for the recursive thread lock). We must hold a lock in order to free it, as seen in the `freelock` rule:

$$\{!!(\mathsf{writable\_share}\ sh \wedge \mathsf{exclusive}\ R) * R * \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \texttt{freelock}(\ell)\ \{R * \ell \xmapsto{sh} \_\}$$

The freed lock converts back into an ordinary memory location, and we can store data in it or convert it into a lock with a different invariant. In this example, we simply end the program instead.

4 Using Ghost State

In the previous section, we proved that `progs/incr.c` is safe, but not that `ctr` 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 `ctr` inside the lock invariant. We can accomplish this with *ghost variables*, a simple form of auxiliary state.

In `progs/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 `ctr`. We will divide ownership of each ghost variable between the lock invariant and the related thread. 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.

4.1 Extending the Specifications

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

$$\mathsf{EX}\ z : \mathsf{Z}, \mathsf{data\_at}\ \mathsf{tuint}\ (\mathsf{Vint}(\mathsf{Int.repr}\ z))\ \texttt{ctr}$$

Now, we want to augment it with shares of two ghost variables. For our convenience, `conclib.v` defines shares gsh1 and gsh2 that are readable halves of the total share Tsh. So our new invariant will be

$$\begin{array}{l}\mathsf{EX}\ z : \mathsf{Z}, \mathsf{data\_at}\ \mathsf{tuint}\ (\mathsf{Vint}(\mathsf{Int.repr}\ z))\ \texttt{ctr}\ * \\ \quad \mathsf{EX}\ x : Z, \mathsf{EX}\ y : Z, !!(z = x + y)\ \&\&\ \mathsf{ghost\_var}\ \mathsf{gsh1}\ x\ g1 * \mathsf{ghost\_var}\ \mathsf{gsh1}\ y\ g2\end{array}$$

The thread that holds the other half of $g1$ or $g2$ can thus record its contribution to `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. (In Section 4.3, we will generalize this to allow the caller to pass any gname from a list.)

```
DECLARE _incr
 WITH ctr : val, sh : share, lock : val,
      g1 : gname, g2 : gname, left : bool, n : Z
 PRE [ ]
        PROP  (readable_share sh)
        LOCAL (gvar _ctr ctr; gvar _ctr_lock lock)
        SEP   (lock_inv sh lock (cptr_lock_inv g1 g2 ctr);
               ghost_var gsh2 n (if left then g1 else g2))
 POST [ tvoid ]
        PROP ()
        LOCAL ()
        SEP (lock_inv sh lock (cptr_lock_inv g1 g2 ctr);
             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 ctr : val, sh : share, lock : val,
      g1 : gname, g2 : gname, n1 : Z, n2 : Z
 PRE [ ]
        PROP  (readable_share sh)
        LOCAL (gvar _ctr ctr; gvar _ctr_lock lock)
        SEP   (lock_inv sh lock (cptr_lock_inv g1 g2 ctr);
               ghost_var gsh2 n1 g1; ghost_var gsh2 n2 g2)
 POST [ tuint ]
        PROP ()
        LOCAL (temp ret_temp (Vint (Int.repr (n1 + n2))))
        SEP (lock_inv sh lock (cptr_lock_inv g1 g2 ctr);
             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 : val * share * val * val * gname * gname
PRE [ _args OF (tptr tvoid) ]
       let '(ctr, sh, lock, lockt, g1, g2) := x in
       PROP  (readable_share sh)
       LOCAL (temp _args y; gvar _ctr ctr; gvar _ctr_lock lock;
              gvar _thread_lock lockt)
       SEP   (lock_inv sh lock (cptr_lock_inv g1 g2 ctr);
              ghost_var gsh2 0 g1;
              lock_inv sh lockt (thread_lock_inv sh g1 g2 ctr lock lockt))
POST [ tptr tvoid ]
       PROP () LOCAL () SEP ().
```

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

4.2 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 new viewshift_SEP tactic to update the ghost variables. This tactic is analogous to replace_SEP, but it adds a modifier that we have not seen before: 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[2]. 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 call to viewshift_SEP here does several things at once. First, we pick out the other half of the ghost variable that was passed to the function (i.e., if left then $g1$ else $g2$) and join them together. We use the lemma ghost_var_share_join′, which tells us that we can join two ghost_var assertions with compatible shares, and learn that they agree on the value of the variable in the process: if we are updating $g1$ then $x = n$, and if we are updating $g2$ then $y = n$, where n is the value of the ghost variable provided by the caller. We then use the lemma bupd_frame_r to frame out the unused ghost variable from the

[2] 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 5.

view shift, and finally apply ghost_var_update to change the value of our ghost variable from n to $n + 1$.

Once this operation is complete, 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 join the shares of both ghost variables, so that we know exactly the values of both x and y (and thus 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 without 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 then 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.

4.3 Generalizing to N Threads

The structure of the ghost state in the previous program limited the number of threads accessing the counter to two. We could pass the ghost variables between threads to enable more than two threads to call incr, but no more than two threads could hold ghost variables at a time, since there were only two ghost variables. In this section, we show how we can make the counter agnostic to the number of ghost variables, extending this limit to an arbitrary N.

The code in incrN.c makes a few additions to incr.c. The counter has initialization and destruction functions init_ctr and dest_ctr, making the counter more of an independent data structure. (We could now move ctr, ctr_lock, init_ctr, and dest_ctr to a separate file from thread_func and main.) The main function now spawns N threads, each with its own lock, each executing thread_func to increment the counter by 1. Once main has joined with all the threads, it reads the final value of ctr, which we expect to be equal to N. Note that none of the counter functions take N as an argument: the number of threads will be a parameter to their specifications, but does not affect the computations they perform, so we could use the counter in multiple programs with different numbers of threads.

To adapt our specifications to N threads, we first generalize the counter lock's invariant to take a list of ghost variables lg. The counter value z will then be the sum of the values of all the ghost variables:

$$\begin{array}{l}\mathsf{EX}\ z : \mathsf{Z}, \mathsf{data\_at\ tuint}\ (\mathsf{Vint}(\mathsf{Int.repr}\ z))\ \texttt{ctr} \ast \\ \quad \mathsf{EX}\ lv : \mathrm{list}\ Z, !!(z = \mathrm{sum}\ lv)\ \&\&\ \circledast_{g \in lg, v \in lv}\ \mathsf{ghost\_var\ gsh1}\ v\ g\end{array}$$

(In Coq, we can write iterated separating conjunction over a list with iter_sepcon, or over two lists with iter_sepcon.) The specifications for the previously existing functions are modified accordingly, taking an index i into the list of ghost variables to indicate which variable will be used to record the calling thread's operations (and `thread_func` now takes as its argument the lock it should use to join). The specification for the new function `init_ctr` takes the number N of simultaneous threads to support; although N does not appear in the body of the function, in the specification we need to know how many ghost variables to create.

```
DECLARE _init_ctr
 WITH N : Z, ctr : val, lock : val
 PRE [ ]
        PROP  (0 <= N)
        LOCAL (gvar _ctr ctr; gvar _ctr_lock lock)
        SEP   (data_at_ Ews tuint ctr;
               data_at_ Ews tlock lock)
 POST [ tvoid ]
   EX lg : list gname,
        PROP (Zlength lg = N)
        LOCAL ()
        SEP (lock_inv Ews lock (cptr_lock_inv lg ctr);
             iter_sepcon (ghost_var gsh2 0) lg).
```

After the call, `ctr` is protected by its lock with the invariant, and N ghost variables have been initialized to 0 (as, by implication, has `ctr` itself). Destructing the counter does the same thing in reverse:

```
DECLARE _dest_ctr
 WITH lg : list gname, lv : list Z, ctr : val, lock : val
 PRE [ ]
        PROP  ()
        LOCAL (gvar _ctr ctr; gvar _ctr_lock lock)
        SEP   (lock_inv Ews (gv lock) (cptr_lock_inv lg ctr);
               iter_sepcon2 (fun g v => ghost_var gsh2 v g) lg lv)
 POST [ tvoid ]
        PROP ()
        LOCAL ()
        SEP (data_at Ews tuint (vint (sum lv)) ctr;
             data_at_ Ews tlock lock).
```

The dest_ctr function retrieves the free shares of all N ghost variables ($N =$ length lg, so it does not need to be passed explicitly), and frees the lock, guaranteeing that the current value of ctr is the sum of the values of the ghost variables.

The proofs for incr and thread_func are almost unchanged from the previous version. The proof of init_ctr is similar to that of the beginning of main, allocating the ghost variables (we use the ghosts_alloc tactic to make a list of N ghost variables) and showing that the lock invariant holds in the initial state. In dest_ctr, we acquire and free the lock and then use the same sort of ghost variable reasoning as in read to show that the list of values associated with the ghost variables inside the lock invariant is the same as the list of values passed in by the caller (and therefore the value of ctr is equal to the sum of that list). At the end of the function, we deallocate the ghost variables: because they are not connected to real memory, we can eliminate them at any time with a view shift.

The proof of correctness of the modified 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. Because each thread needs to know about the counter lock, we use the split_shares lemma to divide 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 counter lock, a ghost variable, and half of a thread lock for joining. In doing so, we gradually use up the data in the thread_lock array by converting it into lock_inv assertions. We use sublist i N to describe the list of remaining shares/ghost variables at the ith iteration; by the end of the loop, $i = N$ and all shares and ghost variables have been given away. In the second loop, we reverse the process, joining with each thread and reclaiming shares and ghost variables—but since each thread we join with has completed its body, each ghost variable now has a value of 1 instead of 0. So when we call dest_ctr, we know that the final value of ctr is the sum of a list of N 1's, which simple arithmetic tells us is equal to N.

5 Defining Custom Ghost State

5.1 The Structure of Ghost State

The ghost variables of the previous section are a special case of a much more general *ghost state* mechanism. In fact, any Coq type can be used as ghost state, as long as we can describe what happens when two elements of that type are joined together. To do so, we create an instance of the Ghost typeclass. A number of instances can be found in progs/ghosts.v. An instance of the Ghost typeclass is a *separation algebra* with associated join relation, with an additional valid predicate marking those elements of the algebra that can be used in assertions. For instance, ghost variables of type A are drawn from the 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[3]. 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*, in the style of Iris [2].) The own predicate is governed by a few simple rules:

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

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

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

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

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

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.

[3] 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.

5.2 Example: incr with Unbounded Threads

We can put custom ghost state to use in generalizing the incr example still further. In Section 4.3, each time we initialized the counter, we chose a bound N on the number of threads that could access the counter simultaneously, and made N ghost variables for that purpose. But in fact, 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. 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 this section, we will define custom ghost state to do exactly that.

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 : $G \to G$ 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.

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 part-reference ghost state we need. We also define some local definitions for the kinds of ghost state we expect to use: partial contributions, reference state, and the combination of both. (Using these definitions, which specialize the parametric definitions in ghosts.v to the sum_ghost instance, allows us to avoid relying on Coq to find the right Ghost instance for our ghost assertions.)

Now that we no longer have a list of ghost variables, the specifications for most functions are simpler. init_ctr gives us a contribution ghost with full share and value 0, and dest_ctr guarantees that the value of ctr is exactly the value of the total contributions of all threads. The incr and thread_func functions no longer need indices; they simply take any arbitrary ghost part and increase its value by 1. Since all the parts will be summed to determine the value of the counter, this precisely reflects the fact that incr increases the counter by 1.

Proving the correctness of our new 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. (Note how much simpler this proof is than that of the previous section, in which we needed to prove that each ghost variable in the list had the same value in the thread as it did in the lock invariant.)

The last major change in the proofs is in incr, in which we want to simultaneously add 1 to a contribution element and the reference element. 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 on any piece of ghost state to both the contribution and the reference, as long as it is safe to add to any element between the contribution and the reference. In sum_ghost, it is always safe to add 1 to any element, so this is easy to prove. 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 remaining proofs (of thread_func and main) are very similar to those in the previous section. In main, we need to split off shares of the ghost contribution for each thread instead of giving away a ghost variable, and rejoin the shares when we join with the threads. Thanks to our simpler ghost state, we can easily track the fact that we have seen total contributions of i after joining with i threads, and quickly conclude that after joining with all N threads the value of `t` is N.

6 The Rules of Concurrent Separation Logic

6.1 Lock and Thread Functions

These specifications can be found in `concurrency/semax_conc.v`.

$$\{!!\mathsf{writable\_share}\ sh \land \ell \overset{sh}{\longmapsto} \_\}\ \texttt{makelock}(\ell)\ \{\mathsf{lock\_inv}\ sh\ \ell\ R\}$$

$$\{!!\mathsf{readable\_share}\ sh \land \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \texttt{acquire}(\ell)\ \{R * \mathsf{lock\_inv}\ sh\ \ell\ R\}$$

$$\{!!(\mathsf{readable\_share}\ sh \land \mathsf{exclusive}\ R) * R * \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \texttt{release}(\ell)\ \{\mathsf{lock\_inv}\ sh\ \ell\ R\}$$

$$\{!!(\mathsf{writable\_share}\ sh \land \mathsf{exclusive}\ R) * R * \mathsf{lock\_inv}\ sh\ \ell\ R\}\ \texttt{freelock}(\ell)\ \{R * \ell \overset{sh}{\longmapsto} \_\}$$

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

6.2 Ghost Operations

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

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

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

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

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

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

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

7 Implementation of Ghost State and Advanced Concurrency Reasoning

The implementation of ghost state and related constructs in VST recapitulates the corresponding constructions from Iris, with some slight modifications. Some features are reimplemented on top of VST's separation logic (Verifiable C), some are imported directly from Iris, and some are definitions from Iris added as axioms. In this section, we will give an overview of all the advanced concurrency features of VST and their implementation.

7.1 Ghost State in the Model

Verifiable C assertions are predicates on rmaps, step-indexed maps from memory locations to annotated values/predicates. Ghost state is implemented as another component of the rmap: every heap is a step-indexed pair of a resource map and a collection of named pieces of ghost state. Each piece is a dependent tuple of a ghost algebra (an instance of the Ghost typeclass), an element of that algebra, a proof that that element is valid, and a predicate (which allows for simple forms of higher-order ghost state). Every Verifiable C predicate is evaluated on the combination of resource map and ghost state; common assertions like data_at refer to empty pieces of ghost state, while own assertions refer to empty resource maps.

This approach is notably different from that of Iris, in which the subject of separation logic predicates is a single piece of ghost state: even points-to assertions are ghost state that is separately linked to assertions about a monolithic physical state, and all the different kinds of ghost state in the program are joined into a single dependent map which itself satisfies the properties of a ghost algebra. VST's approach allowed us to reuse the proofs about points-to assertions with minimal changes, but is not quite as flexible when it comes to higher-order ghost state. On the other hand, VST's approach has a surprising advantage: while in Iris the type of the top-level ghost map is a parameter to the type of predicates, and all reasoning is done in the context of a typeclass asserting that certain kinds of ghost state are present in the map, VST can add new ghost state of any kind at any time during a proof, by adding a new dependent tuple for the desired ghost algebra to the collection.

The own predicate and view shift operator are defined by recapitulating the constructions from Iris: own $g\ a\ pp$ asserts that the ghost state of the rmap is a singleton collection with name g, element a, and predicate pp. Likewise, the view shift is derived from a basic update operator $\mathrel{|\!\dot{\Rrightarrow}}$ of type mpred $\rightarrow$ mpred (written |==> in Coq), where $\mathrel{|\!\dot{\Rrightarrow}} P$ asserts that P is true on an rmap whose ghost state is a frame-preserving update away from the current state. These definitions allow us to prove the rules of section 6.2, giving us a "separation logic with ghost state" in the style of Iris but independent of the Coq development of Iris.

It still remains to eliminate the ghost update operators: after all, we do not want to prove at the end of the increment program that $\mathrel{|\!\dot{\Rrightarrow}} (\mathtt{x} = 2)$, but rather that $\mathtt{x} = 2$. This is accomplished by augmenting the semantics of the "juicy machine", the operational semantics used by Verifiable C (later erased to the actual CompCert semantics of C). After every step of the juicy machine, the machine can make a frame-preserving update to its ghost state. This leads to an extended rule of consequence:

$$\textsf{view\_shift\_conseq}\ \frac{P \Rrightarrow P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \Rrightarrow Q}{\{P\}\ c\ \{Q\}}$$

This rule underlies the viewshift_SEP tactic, which allows us to prove that $P \Rrightarrow P'$ and then replace P with P' in our precondition.

The constructions of this section are sufficient for the examples described above, but they only capture a small piece of the functionality of modern concurrent separation logics. In the following sections, we describe how more features of Iris have been integrated into VST.

7.2 Invariants

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. Although Iris was originally published with the slogan “ghost state and invariants are all you need” [4], it was later shown that in fact even invariants could be defined as ghost state [3]. VST uses a slight variant of Jung et al.’s construction, which is explained well in the reference and will not be detailed here. The end result is a “fancy update” operator ${}^{E_1}{\mid\!\Rrightarrow}^{E_2}$ that allows for the allocation, opening, and closing of invariants. The “masks” E_1 and E_2 are sets of invariant names, indicating which invariants are enabled (not currently open) at this point in the proof. The primary rules for manipulating invariants are:

$$\textsf{inv\_alloc}\ \frac{}{\triangleright P \vdash {\mid\!\Rrightarrow}_E\ \mathsf{EX}\ i : \mathsf{iname}, \boxed{P}^{i}}$$

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

where $\boxed{P}^{i}$ is written in Coq as invariant i P. 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 the following sections). This is all just a particular instance of the general ghost state of the previous section, and so can be used in any VST proof with no additional effort. However, it should be noted that unlike the basic update, the fancy update is not currently part of the juicy machine, which makes it difficult to eliminate fancy updates from our proofs. The fancy update reduces to a basic update in the presence of a “world satisfaction” (wsat) assertion that holds all the resources for enabled invariants. In the semantics of Iris, this wsat is assumed to exist alongside the program at every step. In VST, on the other hand, we need to provide it explicitly:

$$\textsf{make\_wsat}\ \frac{}{\mathsf{emp} \vdash \dot{\mid\!\Rrightarrow}\ \mathsf{EX}\ \mathit{inv\_names} : \mathsf{invG}, \mathsf{wsat}}$$

where invG is a typeclass containing the ghost names needed for the construction of invariants. When we have wsat, we can derive rules for invariants without the fancy update:

$$\textsf{inv\_alloc'}\ \frac{}{\mathsf{wsat} * \triangleright P \vdash \dot{\mid\!\Rrightarrow}\ \mathsf{EX}\ i : \mathsf{iname}, \triangleright(\mathsf{wsat} * \boxed{P}^{i})}$$

But since wsat is not duplicable or splittable, so this is not sufficient to allow multiple threads to access invariants simultaneously (which, after all, is the point of invariants). Until the semantics are modified appropriately, we could add the extended rule of consequence as an axiom, or we could take a slightly more conservative approach and only add axioms for specific uses of the fancy update. For the time being, we have done the latter; two specific uses are described in the following sections.

7.3 Iris Proof Mode

One of the best features of the Iris Coq development 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. 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 effective 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 very 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 $\overset{\cdot}{\models\!\!\Rrightarrow}$, in IPM we can often eliminate them automatically. Once you have imported `VST.veric.bi`, you can switch into IPM for any separation logic entailment using the iStartProof or iIntros tactic, and then use any and all Iris tactics to prove the goal. There is also an iVST tactic for switching back to VST's style of entailment, but it has not been extensively tested—please report any issues by creating an issue on GitHub or emailing `mansky1@uic.edu`. Be warned: 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!

7.4 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 "physically atomic" operations can freely access invariants: for instance, if a global invariant holds full ownership of a memory location `x`, then a physically atomic operation can read or write the value of `x`. In Iris, normal memory operations are usually considered to be atomic, but this is expressly not the case in C: unless we know something about the target architecture, we must assume that intermediate states of a write are visible. On the other hand, there are explicitly atomic memory operations in the C standard: `atomic_load`, `atomic_store`, and related operations. 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 Iris proof rule for atomic operations 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 7.2), 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 $\{x \mapsto v\}$ `y = *x` $\{!!(y = x)\ \&\&\ x \mapsto v\}$. Plugging this into the Hoare triple in the consequence rule gives us:

$$\text{atomic load}\,\frac{P\ \ {}^{E}\!\Rrightarrow^{E\setminus E'}\ x \mapsto v * R \qquad !!(y = x)\ \&\&\ x \mapsto v * R\ \ {}^{E\setminus E'}\!\Rrightarrow^{E}\ Q}{\{P\}\ y = \texttt{atomic\_load}(x)\ \{Q\}}$$

where 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:

$$\text{atomic load}\,\frac{P\ \ {}^{E}\!\Rrightarrow^{E\setminus E'}\ x \mapsto v * (!!(y = x)\ \&\&\ x \mapsto v \mathrel{-\!\!*}\ {}^{E\setminus E'}\!\models\!\!\!\Rrightarrow^{E} Q)}{\{P\}\ y = \texttt{atomic\_load}(x)\ \{Q\}}$$

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

Now we can see that if an invariant I contains a points-to assertion `x` $\mapsto v$, we can access and modify the value of `x` using atomic loads and stores (and *only* atomic loads and stores, until we add atomic rules for other operations). The VST specifications corresponding to these rules can be found in `SC_atomics.v`. They take an additional argument *inv_names* : invG, the typeclass used in the invariant construction, but otherwise are simply the translation of the rules above into VST specs.

Note that, as the name `SC_atomics` suggests, these are rules for *sequentially consistent* atomic operations: it would not be sound to access a points-to assertion in an invariant using weak-memory atomics, since a thread is not guaranteed to see the objective "current value" held in the invariant. A rough axiomatization of the rules of iGPS [5] (a weak-memory logic build on top of Iris) can be found in `acq_rel_atomics` and `acq_rel_SW` (for single-writer protocols); interested parties are encouraged to refer to the cited paper.

7.5 Logical Atomicity

Global invariants are closely related to the traditional correctness criterion of *linearizability.* Suppose we had a data structure with an invariant that said "there exists a linear history of operations performed on the data structure, stored in some ghost state, and the current state of the data structure is the one that would result from applying each of those operations in order." Then this would directly imply that the data structure is linearizable, and we could use the invariant to reason accordingly: we could spawn multiple threads, have each perform some operations (and collect some ghost state representing the operations performed), and then join them and learn that the operations must have executed as if by some interleaving of the threads. *Logical atomicity* is an extension of this idea: by proving that a data structure operation *appears to execute atomically* at some linearization point, we can then allow that operation to access the contents of invariants, exactly as if it were one of the atomic operations of the previous section (even if it is a function that performs multiple memory operations).

Logical atomicity was introduced in the TaDA logic [1], in the form of *atomic Hoare triples.* 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

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

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.

The original Iris paper [4] showed how to encode atomic triples using ghost state and invariants (and called them "logically" atomic triples, to distinguish them from the "physically" atomic low-level operations). The encoding immediately implies that we can access any invariant when calling a function with an atomic triple, in the same way that we can access an invariant using atomic loads and stores. This gives us a mechanism for proving very strong correctness properties for concurrent data structures. The key idea is that during the proof of an atomic triple, we have a resource P that is entirely unknown, except that we can access it with an *atomic shift* that extracts from it the public precondition P_p. Once we have done so, we must atomically either restore P_p (and then regain P) or establish Q_p (and then gain a resource Q indicating that we have reached the linearization point and accomplished the goal of the function). A (rather messy) VST implementation of this encoding can be found in `atomics/general_atomics.v`, which defines notation for an atomic variable of VST funspecs:

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

`W` is the TypeTree representing the type of the WITH clause; `a` is the quantified abstract state for the triple; `Ei` and `Eo` are the masks (sets of enabled invariant names) inside and outside the triple, 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 always be solved automatically. Please note that this notation is extremely hacky; errors and complaints can be reported by creating an issue on GitHub or emailing `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 can be accessed through atomic *commits* and *rollbacks*:

$$\textsf{atomic\_commit}\ \frac{\forall a, R * P_p\ a \Rrightarrow \textsf{EX}\ y,\ Q_p\ a\ y * R'\ y}{\textsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * R \Rrightarrow \textsf{EX}\ y,\ Q\ y * R'\ y}$$

$$\textsf{atomic\_rollback}\ \frac{\forall a, R * P_p\ a \Rrightarrow P_p\ a * R'}{\textsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * R \Rrightarrow \textsf{atomic\_shift}(P_p, E_i, E_o, Q_p, Q) * R'}$$

In a commit, we must provide resources R that, in combination with the public precondition, allow us to prove the public postcondition; we then gain access to an assertion Q that will be required by the postcondition of the function (as well as any leftover resources R'). In a rollback, we provide resources R that, in combination with the public precondition, allow us to restablish the public precondition; we then regain the atomic shift, as well as leftover resources R'. The latter can be useful, for instance, if we have several pieces of information (e.g. ghost state) about the data structure and want to learn some relationship between them from the well-formedness properties of the data structure: we might have snapshots of two successive elements in a list, for example, and want to extract the information that the latter is larger than the former, where the fact that the list is always sorted is present in the public precondition. In order to complete the proof of the function's atomic specification, we must always do a commit to obtain Q; we may do any number of rollbacks before that point, but after that point 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 two additional elements: the true postcondition Q, and the invG instance supporting the invariant construction. 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. Note that both commits/rollbacks and atomic shift proofs involve multiple view shifts and separating implications, and so are much easier to prove in Iris Proof Mode than with VST's tactics.

An example of using atomic specifications can be found in `progs/verif_incr_atomic.v`, which re-proves the increment example using atomic triples. Using this approach, we no longer need to choose a kind of ghost state in advance when verifying the `incr` and `read` functions: instead, we prove triples that look like

$$\langle \mathsf{ctr\_lock} \mid \mathsf{ctr\_val}\ n\rangle\ \texttt{incr()}\ \langle \mathsf{ctr\_lock} \mid \mathsf{ctr\_val}\ (n+1)\rangle$$

that precisely capture the behavior of the function (`incr` atomically adds 1 to the value of the counter). In the client, we create an invariant

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

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 ghost_var gsh2 0 g_1, and proves that the atomic spec provided by `incr` leads to a postcondition 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 ctr_val assertion (the abstract state of the counter) used in the public pre- and post-conditions. 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.

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] 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.

[3] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[4] 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.

[5] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[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.