Verifying Concurrent Programs with VST
William Mansky

1 Introduction

VST 2.1 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 malloced 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 #included 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 sev-
eral 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 pro-
gram 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 ≜

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 lock functions.

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.
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 ≜ R∗R ⊨ 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:

\[
{!!\text{readable share } sh \land \text{lock inv } sh \ell R} \text{ acquire}(\ell) \{R \ast \text{lock inv } sh \ell R\}
\]
\[
{!!(\text{readable share } sh \land \text{exclusive } R) \ast R \ast \text{lock inv } sh \ell R} \text{ release}(\ell) \{\text{lock inv } sh \ell R\}
\]

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 \( \text{ctr} \), its value is still \( z \). The return value of the function is that same \( z \), and the proof is complete. The proof of body\( \text{incr} \) 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 \texttt{spawn} function. The \texttt{spawn} function takes two arguments: a pointer to a function that the new thread should execute, and a \texttt{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. Currently, the starting function for a thread must take a single argument of type \texttt{void} and return a value of type \texttt{void}; the value returned is ignored completely, so it will usually be \texttt{NULL}.

The separation logic rule for \texttt{spawn} is:

\[
\{ P(y) \ast f : x. \{ P(x) \}{\text{emp}} \} \text{spawn}(f,y) \}
\]

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 \texttt{emp} reflects the fact that any resources held by the thread when it returns will be lost forever.

If we want to \texttt{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 \texttt{recursive} lock, one whose invariant includes a share of the lock itself. We can make such an invariant with the \texttt{selflock} function, as we can see in the definition of \texttt{thread\_lock\_inv}, and use it with the lemma \texttt{selflock\_eq}:

\[
\forall Q \, sh \, p, \text{selflock} \ Q \, sh \, p = Q \ast \text{lock\_inv} \ sh \, p \ (\text{selflock} \ Q \, sh \, p).
\]

In Verifiable C, functions that will be passed to \texttt{spawn} must have specifications of a certain form, as in \texttt{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);
```
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:

\[
\{!\text{writable\_share } sh \land \ell \quad \overleftarrow{sh} \mapsto \_\} \text{makelock}(\ell) \{\text{lock\_inv } sh \ell \quad R\}
\]

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\_lock ι`.

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 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 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:

\[
\{ \neg \forall \text{writable} sh \land \text{exclusive} R \} \cdot R \cdot \text{lock inv} \cdot \text{sh} \cdot \ell \cdot R \} \cdot \text{freelock}(\ell) \cdot \{ R \cdot \ell \rightarrow \text{sh} \}
\]

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

\[
\text{EX } z : Z, \text{data at uint} (\text{Vint} (\text{Int repr } z)) \cdot \text{ctr}
\]

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

\[
\text{EX } z : Z, \text{data at uint} (\text{Vint} (\text{Int repr } z)) \cdot \text{ctr} \cdot
\]

\[
\text{EX } x : Z, \text{EX } y : Z, \neg \forall (z = x + y) \land \text{ghost_var} gsh1 x g1 \land \text{ghost_var} gsh1 y g2
\]
The thread that holds the other half of \( g_1 \) or \( g_2 \) can thus record its contribution to \( \text{ctr} \), 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 \( \text{incr} \) now needs to know which ghost variable the caller wants to increment, so it takes a boolean \( \text{left} \) telling it whether we are looking at the left \( (g_1) \) or right \( (g_2) \) ghost variable. (We can imagine generalizing this to allow the caller to pass any \( \text{gname} \) from a list.)

\[
\text{DECLARE _incr}
\]
\[
\text{WITH ctr : val, sh : share, lock : val,}
\]
\[
g_1 : \text{gname}, g_2 : \text{gname}, \text{left} : \text{bool}, n : \text{Z}
\]
\[
\text{PRE [ ]}
\]
\[
\text{PROP (readable_share sh)}
\]
\[
\text{LOCAL (gvar _ctr ctr; gvar _ctr_lock lock)}
\]
\[
\text{SEP (lock_inv sh lock (cptr_lock_inv g1 g2 ctr);}
\]
\[
\text{ghost_var gsh2 n (if left then g1 else g2))}
\]
\[
\text{POST [ tvoid ]}
\]
\[
\text{PROP ()}
\]
\[
\text{LOCAL ()}
\]
\[
\text{SEP (lock_inv sh lock (cptr_lock_inv g1 g2 ctr);}
\]
\[
\text{ghost_var gsh2 (n+1) (if left then g1 else g2)).}
\]

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

\[
\text{DECLARE _read}
\]
\[
\text{WITH ctr : val, sh : share, lock : val,}
\]
\[
g_1 : \text{gname}, g_2 : \text{gname}, n_1 : \text{Z}, n_2 : \text{Z}
\]
\[
\text{PRE [ ]}
\]
\[
\text{PROP (readable_share sh)}
\]
\[
\text{LOCAL (gvar _ctr ctr; gvar _ctr_lock lock)}
\]
\[
\text{SEP (lock_inv sh lock (cptr_lock_inv g1 g2 ctr);}
\]
\[
\text{ghost_var gsh2 n_1 g1; ghost_var gsh2 n_2 g2)}
\]
\[
\text{POST [ tuint ]}
\]
\[
\text{PROP ()}
\]
\[
\text{LOCAL (temp ret_temp (Vint (Int.repr (n_1 + n_2)))))}
\]
\[
\text{SEP (lock_inv sh lock (cptr_lock_inv g1 g2 ctr);}
\]
\[
\text{ghost_var gsh2 n_1 g1; ghost_var gsh2 n_2 g2).}
\]

Finally, we add ownership of ghost variable \( g_1 \) to the resources passed to \( \text{thread_func} \) (and collected by its lock when it terminates):

\[
\text{DECLARE _thread_func}
\]
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 $P \Rightarrow Q$ (written as $P \; \Rightarrow \; Q$ 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. Of particular interest here is lemma
ghost_var_update, which says that $\text{ghost_var Tsh v p} \Rightarrow \text{ghost_var Tsh v' p}$. 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 upd_frame_r to frame out the unused ghost variable from the
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 normal 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.

5 Defining Custom 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 $\text{option}(\text{share} \ast A)$, where valid elements have nonempty shares. An element $\text{Some}(sh, a)$ represents a share $sh$ of value $a$, and $\text{None}$ 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. For instance, ghost_var $sh$ $v$ $g$ is defined as own $g$ $(\text{Some}(sh, v))$ 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.) The own predicate is governed by a few simple rules:

\[
\begin{align*}
\text{own_alloc} & : \forall g, a, pp. \text{valid } a \Rightarrow \text{emp } \Rightarrow \text{EX } g : \text{gname}, \text{own } g \ a \ pp \\
\text{join} & : \forall a1, a2, a3. \text{own } g \ a3 \ pp = \text{own } g \ a1 \ pp \ast \text{own } g \ a2 \ pp
\end{align*}
\]
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_NPC 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, \(\text{fp_update } a b \equiv \forall c, (\exists d, \text{join } a c d \land \text{valid } d) \rightarrow (\exists d, \text{join } b c d \land \text{valid } d)\).

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 \texttt{ghosts.v}, but we are always free to define new \texttt{Ghost} instances for more complicated patterns of sharing and recording.

6 The Rules of Concurrent Separation Logic

6.1 Lock and Thread Functions

These specifications can be found in \texttt{concurrency/semaxconc.v}.

\[
\begin{align*}
\text{makelock}(\ell) & \{\text{lock_inv } sh \ell R\} \\
\text{acquire}(\ell) & \{R \ast \text{lock_inv } sh \ell R\} \\
\text{release}(\ell) & \{\text{lock_inv } sh \ell R\} \\
\text{freelock}(\ell) & \{R \ast \ell \rightarrow \_\} \\
\text{spawn}(f, y) & \{\} \\
\end{align*}
\]
6.2 Ghost Operations

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

\[
\begin{align*}
\text{own_alloc} & \quad \text{valid } a \\
\text{emp} \Rightarrow & \quad \text{EX } g : \text{gname, own } g \ a \ pp
\end{align*}
\]

\[
\begin{align*}
\text{join a1 a2 a3} \\
\text{own g a3 pp = own g a1 pp } \ast \text{own g a2 pp}
\end{align*}
\]

\[
\begin{align*}
\text{own_valid_2} \\
\text{own g a1 pp } \ast \text{own g a2 pp } \Rightarrow & \quad (!!(\exists a3, \text{join a1 a2 a3 } \land \text{valid a3})
\end{align*}
\]

\[
\begin{align*}
\text{fp_update_ND } a \ B \\
\text{own g a pp } \Rightarrow & \quad \text{EX } b, !!(B b) \land \text{own g b pp}
\end{align*}
\]

\[
\begin{align*}
\text{fp_update a b} \\
\text{own g a pp} \Rightarrow & \quad \text{own g b pp}
\end{align*}
\]

\[
\begin{align*}
\text{own_dealloc} \\
\text{own g a pp} \Rightarrow & \quad \text{emp}
\end{align*}
\]