USSL: A Universe-Stratified Concurrent Separation Logic for Intrinsic Proofs of Dependently Typed Programs

ANONYMOUS AUTHOR(S)*

Modern separation logics supporting features such as dynamically allocated locks or invariants resort to step-indexed constructions to break an inherent circularity in the semantics of storable heap predicates. We present a new, simpler approach by observing that the universe hierarchy of a dependently typed language can be used to stratify heaps and heap predicates, without requiring step indexes. The resulting construction is predicative and less expressive than fully impredicative models, though it can be applied directly to intrinsic proofs of shared-memory, concurrent dependently typed programs, for both partial and total correctness.

We develop our construction as a new library in F*, called USSL, with many features of a modern separation logic, including dynamically allocated invariants and forms of higher-order ghost state, based on partial commutative monoids, all backed by a mechanized, foundational proof of soundness. We also design libraries that use refinement subtyping to simplify the use of stratified proposition. We evaluate USSL by developing various concurrent programs, including synchronization primitives such as dynamically allocated locks and barriers. We also report on our experience using it to verify an implementation of an industry-standard secure boot protocol.

Note: This paper describes an earlier, predicative version of the PulseCore program logic. PulseCore has since been generalized to support impredicative invariants using a different construction than what is described here. Please see https://doi.org/10.1145/3729311 if you are interested in learning about PulseCore. If you are interested in how one can use universes to build a predicative separation logic with dynamically allocated invariants, then read on ahead.

1 INTRODUCTION

Dependently typed programming is a powerful technique for provably correct functional programming. We seek to broaden the scope of dependently typed programming to apply also to concurrent programs, including those that use shared memory, with concurrent separation logic as a foundation for specifications and proofs in an intrinsic style, i.e., the type of a program directly expresses its correctness specification. Such a system would provide a language-integrated logical notion of ownership similar to but more flexible than Rust's, and with the expressive power to specify and prove the correctness of programs.

There are several lines of prior research to consider. Iris (Jung et al. 2018) is a highly expressive program logic formalized in Coq within which one can prove the correctness of concurrent programs. However, Iris is not itself integrated within a dependently typed language and is not typically used to reason about Coq programs itself. VST (Appel 2012) is another foundational approach to separation logic in Coq, but similar to Iris, it is applied to proofs of deeply embedded C programs, rather than to Coq directly. Hoare Type Theory & FCSL (Nanevski et al. 2008, 2014, 2019), and CFML (Charguéraud 2011) in contrast, shallowly embed a CSL in Coq and for Coq programs, and are closer to our goal. However, they lack support for dynamically allocated invariants and higher-order ghost state, e.g., one cannot program a dynamically allocated mutex in HTT, FCSL, or CFML (which focuses primarily on sequential programs). The closest prior work to ours is SteelCore (Swamy et al. 2020), a CSL shallowly embedded in F* (Swamy et al. 2016), with support for dynamically allocated invariants. However, SteelCore's model relies on a non-standard axiom for monotonic state (Ahman et al. 2018) which, as we discuss in detail in §??, is unsound when combined with certain commonly used classical axioms.

1.1 USSL: A Universe-Stratified Separation Logic

A key difficulty in giving a model to a modern CSL is to handle *invariants*, a feature that requires "storing" a heap predicate in the heap, which is inherently circular. Prior approaches to addressing this problem involve the use of step-indexing, a technique that breaks the circularity by approximating predicates by an index tied (loosely) to the number of steps of computation that have been taken. In Iris, the construction involves interpreting separation logic propositions in a custom category, different from the default category of Coq's Type, making it hard to apply Iris to Coq itself. VST's model also uses step-indexing, based on its underlying Indirection Theory (Hobor et al. 2010), and does not rely on advanced categorical constructions, though it models only dynamically allocated locks rather than invariants, it lacks other forms of higher-order ghost state, and the use of step-indexing limits its use to only partial correctness proofs.

Our main observation is that there is another way to break the circularity, without requiring custom categories and step indexing, while obtaining a separation logic that supports dynamically allocated invariants and higher-order ghost state. Rather than approximating predicates by step indexes, we stratify the heap into several layers, each corresponding to a universe level. Heap predicates that speak about a given layer of a heap cannot be stored in that layer of the heap (that would be circular), but they can be stored in the next layer (or above).

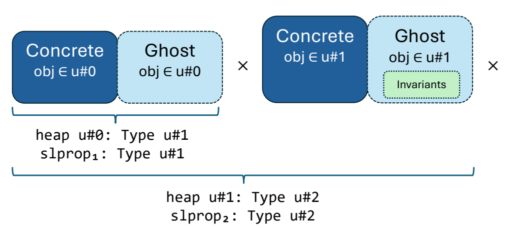

Fig. 1. USSL's heap: base case and one extension

Developing this idea, we present USSL, a new dependently typed, concurrent separation logic shallowly embedded in F*. The core logic of F* is an (extensional) type theory with a countably infinite hierarchy of predicative universes, with universe polymorphism—such features are present in other dependently typed languages, and we believe the core elements of our model could be developed in other tools too. Our construction works by identifying an abstract signature slsig u#α, for a separation logic at a given universe level u#α. This signature includes, among other things, the type of heaps, predicates, and invariants and various laws that they are expected to satisfy. The central part of the construction shows how to extend a signature from a given level u#α to the next level u#(α+1). The diagram alongside is a sketch of the construction, focusing on heaps, predicates, and invariants, showing the base case and one step of extension. Each heap layer is composed of a concrete and a ghost compartment—the latter is used to store "ghost" objects that are for specification and proof purposes only. Among the ghost objects are *invariants*, predicates about the heap that are maintained by every step of program execution. For the base case, invariants are degenerate. A heap layer is indexed by a universe level. For example, heap u#α can store objects that reside in Type u#α. Separation logic propositions on heap u#α, written slprop$_{\alpha+1}$, reside in universe Type u#(α+1)—so they cannot be stored in heap u#α. However, the next heap layer, heap u#(α+1), is a product of heap u#α with concrete and ghost compartments at the next universe level—so, a proposition slprop$_{\alpha+1}$ can be stored in a heap u#(α+1). We show that our construction generalizes to an arbitrary number of layers by proving that any heap-like structure can be extended with a layer in the next universe, deriving a logic over the extended product of heaps. The separation logic propositions of this derived logic can be stored in its (combined) heap, except for those at the topmost level.

Some pros and cons. The main advantage of our approach is that it yields an "elementary" model of a modern CSL with a simple construction that is usable for intrinsic proofs. The model supports dynamically allocated invariants, in the sense that any storable proposition p:slprop$_\alpha$ can be stored in a heap_α as an invariant and described using the assertion $\mathsf{inv}_\alpha\ \mathsf{i}\ \mathsf{p} : \mathsf{slprop}_{\alpha+1}$, where i is the invariant's name. As usual, invariants can be "opened" (i.e., used and restored) in computations consisting of no more than a single, physically atomic step. The main downside of our approach is that the heap extension construction applies only up to some bound, so invariants are not impredicative. That is, the $\mathsf{inv}_\alpha\ \mathsf{i}\ \mathsf{p}$ proposition is not itself storable in a heap_α, so one cannot construct a resource of the form $\mathsf{inv}_\alpha\ \mathsf{i}\ (\mathsf{inv}_\alpha\ \mathsf{j}\ \mathsf{p})$, only $\mathsf{inv}_{\alpha+1}\ \mathsf{i}\ (\mathsf{inv}_\alpha\ \mathsf{j}\ \mathsf{p})$, allocating an invariant at layer $\alpha+1$ that mentions an invariant at layer α. This is a significant limitation, and indeed some higher-order examples such as the event loop studied by Svendsen and Birkedal (2014) appear to not be expressible in USSL. Nevertheless, many non-trivial examples can be expressed, including examples that require a mixture of dynamically allocated invariants and higher-order ghost state (e.g., a barrier studied by Jung et al. (2016)), as well as a wide range of case studies on which we say more, below.

1.2 USSL at Work

USSL is intended as the semantic foundation for USSLang, a surface language for programming and proving in CSL implemented as an F⋆ compiler plugin. The design and implementation of the USSLang plugin is beyond the scope of this paper, though our examples and libraries use USSLang to demonstrate the usefulness of the USSL program logic.

To streamline working with stratified propositions, we show how to structure them using refinement types that define isomorphisms between propositions at different layers. The result is that although the logic is stratified, many common uses can ignore the stratification, and where stratification is relevant, SMT-based proof automation shields the user from a lot of the complexity.

We illustrate the logic at work by using it to program libraries that combine features such as invariants and higher-order ghost state. We also use USSL to verify an implementation of the DICE Protection Environment (DPE) (Trusted Computing Group 2023), an industry standard secure boot protocol service. DPE uses USSLang in an end-to-end concurrency-capable application, providing a real-world test of the expressiveness and usability of the logic. In total, we report on more than 20,000 lines of code that have been verified using USSLang.

1.3 Summary of Contributions

In summary, we offer the following contributions:

- USSL: A new stratified heap construction based on universe levels suitable as a semantics for a dependently typed concurrent separation logic with support for dynamically allocated invariants and storable propositions in a predicative setting. Our model is fully formalized in about 15,000 lines of pure F⋆.
- The design of proof-oriented libraries that structure and ease the use of stratified propositions using refinement types, typeclasses, and dependently typed generic programming.
- An evaluation of USSL demonstrating its expressiveness and limits, on examples such as dynamic locks and barriers, that have previously been proven in impredicative logics.
- An implementation of DPE in USSLang, including a formalization of its top-level API, with a proof that it respects a state machine specification while serving multiple concurrent sessions.

All our code is open source and mechanically checked by F⋆, and available in the anonymous supplement.

2 USSL: FROM A USER'S PERSPECTIVE

We present USSL using USSLang code to illustrate its main features, aiming to give the reader a flavor of the program logic before we formalize it in the next section. We only present what is needed to verify a simple spin lock, deferring more advanced features until later. We present background on F⋆ as we go.

At its core, F⋆ is a purely functional language, though we aim to write shared memory concurrent programs in it. We encode concurrency in F⋆ by representing a program as a forest of infinitely branching trees, one tree for each thread, and where the root of a thread's tree represents an atomic step of computation to be executed next. The dynamic semantics is given by an interpreter of these action-tree forests that non-deterministically interleaves atomic actions from each thread (§3.7). Since we aim to write potentially non-terminating programs, this interpreter uses F⋆'s support for general recursion, encapsulated in an effect of divergence. These semantics provide a formal basis on which to reason about concurrent programs, though for efficient execution, USSLang programs are extracted to OCaml, C, or Rust rather than interpreted as trees in F⋆.

USSL provides the corresponding static semantics to reason about the (partial) correctness of programs executing in our interleaving-of-atomic-actions semantics. As in other concurrent separation logics, all specifications are interpreted in the context of a program fragment executing concurrently with other (type-correct) threads. Programs operating on shared resources can try do so in a lock-free manner with atomic operations; or, they may use various libraries for synchronization built using the underlying primitive atomic operations. The simplest of such libraries is a spin lock, which enforces mutual exclusion by busy-waiting while trying to atomically compare-and-set a mutable memory location shared between threads.

2.1 An Interface for a Spin Lock

Consider the following interface to a library implementing a spin lock. This interface uses the concrete syntax of F⋆, which is similar to the syntax of OCaml (e.g., val for signatures, let for definitions, etc.), but with dependent refinement types. We adopt a convention where free variables in a definition are implicitly bound at the top of the definition, except when we think explicit binders help with clarity.

```
val lock : Type u#0
val protects (l:lock) (p:slprop) : slprop
```

This interface offers an abstract type lock : Type u#0, a type in the lowest universe. The universe is important: referring back to our stratified heap diagram from Figure 1, universe-0 terms can be stored in the heap without a problem. So, our locks will storable in the heap.

The interface also offers an abstract predicate protects l p, associating a lock l with a proposition p. The type slprop, the main type of separation logic propositions in USSLang, corresponds to slprop_4 from the Introduction, and is an affine predicate over the entire extended heap—recall that USSLang instantiates USSL with four heap layers, so slprop resides in Type u#4.

The basic slprop connectives include emp, the trivial proposition valid in all heaps, and p ** q, the separating conjunction,[1] valid when a heap can be split into two disjoint fragments that validate p and q. As usual, ** is associative and commutative, where emp is both its left and right unit.

The are four main operations on our simple locks: create, dup, acquire, and release. For this simple example, we do not support de-allocation of a lock, nor do we prevent double-releases, though our actual libraries do.

Create & storable propositions. The syntax below introduces the signature of a stateful function create, with a single parameter p, a *storable* proposition. The precondition of create is p; a named return value l:lock; and a postcondition protects l p.

```
fn create (p:storable) requires p returns l:lock ensures protects l p
```

[1] We use ** for separating conjunction instead of *, to avoid clashes with other uses of *, e.g., integer multiplication.

In other words, the create p allocates a lock l, takes ownership of the proposition p, and associates it with the lock by returning protects l p. Importantly, p is a storable proposition—as we'll see, under the covers, implementing the lock allocates p as an *invariant*, and invariants can only be allocated for storable propositions.

The definition of the storable type follows the structure of our stratified heap construction. In particular, when extending a heap, we derive a pair of functions $\uparrow\alpha : \mathsf{slprop}_\alpha \to \mathsf{slprop}_{\alpha+1}$ and $\downarrow\alpha : \mathsf{slprop}_{\alpha+1} \to \mathsf{slprop}_\alpha$, to convert between the propositions of adjacent heap layers. The type storable is the restriction of slprop to those elements that are isomorphic to propositions in slprop_3, i.e., storable = p:slprop{ is_storable p } where is_storable = $\uparrow_3$ ($\downarrow_3$ p) == p. We also define refinements such as p:slprop{ $\uparrow_3$ ($\uparrow_2$ ($\downarrow_2$ ($\downarrow_3$ p))) == p}, for propositions isomorphic to slprop_2, etc.

The proposition protects l p is not itself storable; as such, this interface does not support allocating locks that protect other locks, e.g., although protects l1 (protects l0 p) is a well-typed slprop, there is no way to actually create an instance of this proposition, since create expects a storable argument.

Ghost functions: Duplicating permission on a lock. In this case at least, locks that protect other locks are not particularly useful, since protects is duplicable, as shown by the signature of dup below.

```
ghost fn dup (l:lock) requires protects l p ensures protects l p ** protects l p
```

The postcondition of dup *duplicates* the precondition, protects l p. The return type of dup is just unit: USSLang lets us omit it, instead of having to write returns _:unit.

The ghost keyword indicates that dup l is a *ghost function*, a key feature of USSL: ghost functions differ from regular stateful functions in that, 1. they always terminate; 2. they do not read or write any concrete state, though they may read and write ghost fragments of the heap; 3. they can be used in any context of a USSLang computation, so long as that context does not depend on the return value. F⋆ already provides a notion of ghost terms as an *effect*, guaranteeing to erase terms with ghost effect when compiling a program, while enforcing that compiled programs cannot depend on the values of ghost terms. Under the covers, USSL ghost functions are just a particular instantiation of F⋆ ghost functions, which means they are erased as well.

Acquire & Release. Finally, we have functions to acquire and release locks: acquire l blocks until the lock becomes available and then returns p; while release l gives up ownership of p to the lock.

```
fn acquire (l:lock) requires protects l p   ensures protects l p ** p
fn release (l:lock) requires protects l p ** p   ensures protects l p
```

Unlike ghost functions, regular functions like acquire may loop forever—USSL is a program logic of *partial correctness*, where potentially non-terminating computations are modeled using F⋆'s effect of divergence.

2.2 Representing a Lock: Invariants

Invariants are named propositions that are valid before and after every step of computation. In particular, inv i p asserts that p:slprop is an invariant whose name is i, which has the type of invariant names, iname. The type iname is an *erasable* type in F⋆, meaning that invariant names are only needed for specification and proof purposes—they are irrelevant at runtime.

A lock is just a record containing a mutable reference to a machine word r:ref U32.t and an invariant name i:iname. The invariant of the lock (lock_inv) states that when the flag r is cleared, the lock owns p; otherwise it owns nothing—the proposition r $\mapsto$ v asserts that the reference r *points-to* a heap cell containing the value v, and $\exists_*$ is an existential quantifier for slprop. The protects predicate states that the invariant name l.i is associated with lock_inv l.r p, whose type states that it is storable if p is storable—we'll see why this is important, next.

```
fn create (p:storable) requires p ensures protects l p {
  let r = alloc 0ul;
  let i = new_invariant (lock_inv r p);
  let l = {r;i}; rewrite each r as l.r, i as l.i; l }

fn release l requires p ** protects l p
ensures protects l p { with_invariants l.i
 { drop (if _ then _ else _); l.r := 0ul }}
```

```
fn rec acquire l requires protects l p
ensures p ** protects l p {
 let retry = with_invariants l.i
 returns retry:bool
 ensures lock_inv l.r p ** (if retry then emp else p)
 { let b = cas l.r 0ul 1ul;
   if b { false } else { true }};
 if retry { acquire l }}
```

Fig. 2. Implementing and proving create, acquire, and release

```
type lock = { r:ref U32.t; i:iname }
let lock_inv r p : v:slprop { is_storable p ⟹ is_storable v } = ∃* v. r ↦ v ** (if v=0ul then p else emp)
let protects l p = inv l.i (lock_inv l.r p)
```

This type of lock_inv illustrates the integration of F⋆'s refinement types with USSL's logic. We prove that the ↑ and ↓ maps preserve the structure of propositions, and provide lemmas that enable the Z3 SMT solver (de Moura and Bjørner 2008) integrated with F⋆'s typechecker to automatically prove, for example, that is_storable p ∧ is_storable q ⟹ is_storable (p ** q), and (∀ x. is_storable (p x)) ⟹ is_storable etc.

2.3 Creating a Lock: new_invariant

The ghost operation new_invariant p dynamically allocates an invariant, requiring the caller to give up ownership of p, and gaining instead an invariant inv i p. The inv predicate is qualified to a given heap layer—USSLang defines inv i p to be inv_4 i p.

```
ghost fn new_invariant (p:storable) requires p returns i:iname ensures inv i p
```

A key restriction is that the proposition p is *storable*. Intuitively, referring back to Figure 1, this means that p can only describe properties of parts of the heap that exclude the highest universe level. In other words, the p:storable restriction on new_invariant forces a form of predicativity—propositions that can be turned into invariants exclude invariants themselves.

Allocating a lock. Using new_invariant, the code in Figure 2 shows and implementation and proof of create in USSLang. Program proofs like this are implemented interactively in USSLang, where the user queries a VSCode-based development environment for the proof state at each point, adding the appropriate annotations and advancing through the proof a line of code at a time—a "live" interactive proof experience similar to recent work by Gruetter et al. (2024). USSLang provides some proof automation, primarily related to automatic framing, together with support for user-provided hints for rewriting and folding and unfolding predicates, integrated with F⋆'s SMT based automation. USSLang makes different proof-engineering tradeoffs than F⋆ itself, favoring control over automation, and producing smaller, targeted SMT queries. This requires the user to be more explicit about some proof steps such as equational rewriting, that are usually implicit in default F⋆.

The code is relatively straightforward: we allocate a new ref cell r. Then allocate an invariant, at which point we need to prove that lock_inv r p is storable, but since p:storable, the refinement type of lock_inv suffices for the SMT solver to complete the proof. Finally, after a rewrite to change inv i (lock_inv r p) to inv l.i (lock_inv l.r p), we return a record representing the lock.

2.4 Invariants are Duplicable

Invariants inv i p are duplicable, in the sense that inv i p can be converted to inv i p ** inv i p, as shown by dup_invariant below. This is important since it allows invariants to be shared among multiple threads. Since protects l p is essentially just an invariant, duplicating it is easy:

```
ghost fn dup_invariant (i:iname) (p:slprop) requires inv i p ensures inv i p ** inv i p
ghost fn dup l requires protects l p ensures protects l p ** protects l p
{ unfold protects; dup_invariant _ _; fold protects; fold protects; }
```

2.5 Using Invariants in acquire & release: with_invariant

Once an invariant inv i p is allocated, USSL enforces that p is valid before and after every step of computation in all threads throughout the remainder of the program. For this purpose, USSL distinguishes a class of *atomic* steps, instructions that execute in a single physical step on a given computer. For examples, most platforms support several forms of atomic operations, e.g., atomic reads, writes, or compare-and-swap (CAS) instructions. When proving a program that is intended to execute on a given platform, one reflects its atomic instructions in USSL as primitive operations with types such as the following—note the atomic keyword.

```
atomic fn cas (r:ref U32.t) (old v:U32.t)
requires r ↦ u   returns b:bool ensures (if b then r ↦ v ** pure (old == u) else r ↦ u)
```

The signature above states that cas r old v is an atomic operation on a 32-bit integer reference r, requiring permission to the reference, r ↦ u. The atomic operation cas returns a boolean b indicating whether or not it succeeded: if so, r is updated to contain the new value v while proving that its previous value u is equal to old; otherwise, r is unchanged. Note, the proposition pure (p:prop) : storable lifts pure propositions to (storable) propositions.

Atomic computations can be preceded or followed by any number of ghost steps—the resulting computation is still considered atomic. A key rule of USSL, expressed by its with_invariant combinator, is that an invariant can be *opened*, i.e., assumed and restored, for a single atomic step, or zero or more ghost steps. That is, having proved inv i p ** pre, an atomic computation e can assume p ** pre, return a value x:a, and ensure p ** post x, so long as e only opens invariants whose names are distinct from i.

Implementing acquire. To explain more, let's look at with_invariants in action in the implementation of acquire, the tail-recursive function shown at the right of Figure 2. One can also implement it using USSLang's support for while-loops, though this tail-recursive implementation is shorter. The key point to note here is that invariants can be opened for at most one atomic operation, and can assume lock_inv l.r p in the body of the with_invariants l.i block. So, we do a single cas, and if the cas succeeded, we know the old value of l.r must have been 0ul; so we have p and can return retry=false; otherwise, we return retry=true from the block. After the block, if retry is set, we recurse; otherwise, we have p and we can return. Of course, acquire can loop forever, in case the thread holding the lock never releases it. Recall, non-atomic, non-ghost computations in USSL are only proven partially correct—they are allowed to loop indefinitely.

Implementing release. Finally, the implementation of release, shown at the bottom left of Figure 2 is simpler. Our goal is to return ownership of the proposition p to the lock, and clear the underlying flag stored in the lock's ref cell, using and restoring the invariant in a single atomic step. We open the invariant and have lock_inv l.r p ** p in the block; we unconditionally clear the flag and restore the lock_inv having returned p to the lock. A quirk is that this simple interface to locks allows a thread to release a lock even when it is not held, so long as they can prove the proposition p.

So, the drop operation explicitly drops any permission already held by the lock—this is allowed since USSL is an *affine* separation logic. However, to avoid unintended resource leaks, USSLang will not implicitly drop resources, and the explicit drop is needed. Our supplement shows a more sophisticated implementation of locks that forbids such resource leaks.

We hope this conveys a flavor of the USSL logic, focusing primarily on its support for dynamically allocated invariants, storable propositions, and three kinds of computations. USSL provides many other features, including a memory representation based on partial commutative monoids that allow users to develop custom memory layouts and sharing disciplines; various kinds of ghost state; and support for derived connectives, all integrated in F⋆'s dependent type system. In the next section, we build a formal model for USSL that supports all these features.

3 FORMALIZING USSL

We now explain our formal model of USSL by presenting first, an abstract universe-polymorphic signature of heaps and heap predicates. We then show how to inhabit that signature in two ways. First, we present a base heap at a chosen universe—this gives a simple separation logic, but with only a degenerate notion of invariant. Then, we present the essence of our stratified heap construction, which, for a heap at a given universe, constructs an extended heap at the next universe. The extension of a heap lifts predicates of the original heap to the extended heap, while making them storable and allocatable as invariants. Starting from a base heap and iterating heap extension, our soundness proof for USSL applies to an arbitrary number of levels. Finally, we show how to represent concurrent computations using a kind of indexed free monad, and reason about them using the USSL logic.

Note, our presentation here is slightly simplified from our actual mechanized model—we omit some technicalities that are overly specific to F⋆, focusing on the main ideas that should generalize to other settings. Also, for lack of space, we omit some features related to proving the injectivity of the association between invariants and their names or their freshness, though our model does have these features. We also use a slightly more mathematical notation than our actual F⋆ code. The formalization is about 15,000 lines of F⋆ code, which is checkable in about 5 minutes on a modern laptop using 8 threads, though we expect the proofs to become more compact over time.

3.1 An Abstract Signature of USSL Heaps and Heap Predicates

As in many standard models of separation logic (Jensen and Birkedal 2012), a USSL predicate is just an affine predicate over a partial commutative monoid (PCM), as defined by the pcm a type below from F⋆'s standard library.

```
type pcm a = { composable: symrel a;    op: (x:a → y:a{composable x y} → a);    one:a;
               laws: associative_commutative_unit op one;    refine: a → prop }
```

There's perhaps one unusual bit: a pcm a includes a predicate refine that allows one to define an invariant on the carrier type. PCMs are used in separation logics to describe *partial knowledge* of some shared state. For example, one could have PCM to represent partial knowledge of a pair a & b, with four values Neither, X x, Y y, and Both x y. A thread asserting knowledge of X x expresses ownership of the first component of the pair, only. However, X x is not a real value stored in memory representing a pair. The refine predicate allows the PCM designer to enforce that only certain cases of the PCM carrier can actually occur in memory at runtime, the other cases just being partial views of the in-memory structure. Others have used refined PCMs to construct non-trivial gadgets for spatial and temporal sharing (Arasu et al. 2023), so USSL retains this capability. Of course, one can simply disregard the refine field by setting it to $\lambda\ \_ \rightarrow \top$.

When x y : a have the type of a carrier of a PCM p:pcm a, we write x .? y for p.composable x y, and $x \odot y$ for p.op x y, and leave the specific PCM p implicit. Our model is defined for an abstract type heap with a pcm heap structure— for heaps, we write empty for p.one.

Affine predicates are slprops. Affine predicates over PCMs are defined as follows:

```
let is_affine (p:h → prop) = ∀ h₀ h₁. (p h₀ ∧ h₀ .? h₁) ⟹ p (h₀ ⊙ h₁)
let affine_p = p:(h → prop) { is_affine p }
```

The type slprop for a given heap h is defined as affine_p h.[2] One can define the trivial proposition emp and the separating conjunction (p ** q) in a standard way, as shown below, and prove that they form a commutative monoid over slprop.

```
let pure p = λ h → p     let emp = pure ⊤     let ( ** ) p q h = ∃ h₀ h₁. h₀ .? h₁ ∧ h == h₀ ⊙ h₁ ∧ p h₀ ∧ q h₁
```

Memories: Heaps with metadata. Its useful to think of heaps as a map from abstract memory addresses to heap cells. In addition to this map, modeling the memory of a program may require some additional metadata—in our instantiation of the model, the metadata contains freshness counters. We call this type mem where the class memory mem associates with a memory a type of separable heaps and their predicates, together with a function heap_of : mem → heap to project a heap from a memory.

```
class memory (mem:Type u#a) = { heap: Type u#a;     sep: pcm heap;     heap_of: mem → heap; }
```

For a m:memory mem, we write slprop m to mean affine predicates over m.heap. When it is clear from the context, we simply write slprop. We also write interpret p m to mean p (heap_of m). For a m:memory mem, we define rmem m, for refined memory, as r:mem{ m.sep.refine (heap_of r) }, i.e., memories whose heap satisfies the PCM refinement.

So far, all of what we've shown is relatively standard and corresponds to an abstract basis for a variety of separation logics. We now get to our model of invariants, storable propositions, and erasure—three central notions in USSL.

Invariants. Recall from the previous section that invariants in USSLang are named propositions, inv i p, where an abstract name i:iname is associated with proposition p:slprop. The class below provides the iname type (in universe u#0, so invariant names can be stored in the heap), and the invariant former inv. The predicate iname_ok i m is a weaker form of inv i p—it states only that the name of an invariant is valid, but not anything about the proposition associated with it.

```
class invariants (mem:Type u#a) (m:memory mem) = {
  iname:Type u#0;  inv : iname → slprop m → slprop m;
  iname_ok: iname → mem → prop;
  inv_iname_ok : (i:iname → p:slprop m → h:mem → Lemma (interpret (inv i p) h ⟹ iname_ok i h));
  dup_inv : (i:iname → p:slprop m → Lemma (inv i p == inv i p ** inv i p));
  mem_owns : set iname → mem → slprop m;
  mem_owns_equiv : (i:iname → p:slprop m → h:mem → e:set iname →
    Lemma ((interpret (inv i p) h ∧ i ∉ e) ⟹ mem_owns e h == mem_owns (i ∪ e) h ** p));
}
```

[2] A technicality in F* is that slprop is actually defined as the refinement of affine_p h to functions for which the extensionality holds, i.e., ∀ h. (p h ⟺ q h) ⟹ p==q. This would be unnecessary in other type theories where the extensionality axiom is admissible for all functions.

Key properties of invariants. First, via dup_inv, invariants are duplicable. The main property of invariants is mem_owns e m—the predicate can depend on the entirety of m, since it typically involves scanning the entire heap for stored invariants from the freshness counter down. It states, roughly, that except for the invariants whose names are in the exclusion set e, all invariants in m are valid. More precisely, via mem_owns_equiv, in a heap that validates inv i p, and when i is not in the set of excluded invariant names e, the predicate mem_owns e m is equivalent to p and, separately, mem_owns (i ∪ e) m, which additionally excludes the invariant in question, i. In other words, mem_owns e m describes all the invariants in the heap that are not currently opened by some thread in the program.

Liftable propositions. For memories that are defined in universe u#(a + 1), we define a class containing a type prev in the previous universe u#a, with a pair of functions, up and down, to convert between prev and slprop m. An p:slprop is storable if it is isomorphic to prev, i.e., up (down p) == p.

```
class with_prev (mem:Type u#(a + 1)) (m: memory mem) = {
  prev : Type u#a;      up: prev → slprop m;      down: slprop m → prev;
  up_down: (p:prev → Lemma (down (up p) == p)); }
```

The next part of our signature involves capturing requirements to build a model for ghost state. This model is based on the notion of erasure in F⋆—we describe this briefly next.

A Primer on Erasure and Ghost Computations in F⋆. Erasure in F⋆ is important primarily for efficiency—a program p should be compiled while erasing all sub-computations that do not affect p's result. F⋆'s type-and-effect system allows encapsulating computationally irrelevant terms to ensure that the reduction of a pure term cannot depend on the values of encapsulated irrelevant terms. The encapsulation mechanism is based on a simple monadic dependence tracking scheme, inspired by other calculi for monadic information flow control, notably Abadi et al. (1999), and one could implement the same basic scheme as a foundation for erasure in other languages.

Very briefly, F⋆ provides an abstract, monadic type erased t, with a constructor, the return of the monad, hide: t → erased t used to mark certain terms as irrelevant, e.g., the term hide (1 + 1) has type erased nat but will be erased by the compiler to () rather than being reduced at runtime to, say, hide 2. One can work with erased values in an explicitly monadic style. However, F⋆'s effect system provides a more convenient syntax. In particular, an inverse function reveal satisfies reveal (hide x) == x, though reveal (x:erased t) : t is marked with a "Ghost" effect, which taints any pure computation that depends on the result of a reveal. This means that no pure computation can compose with it, except when t is a type that carries no information (e.g., it is a sub-singleton, or t = erased s, etc.), similar to the elimination rule for Prop in Coq. Non-informative types inhabit non_info t, the type of total functions that are equivalent to the ghost function reveal, i.e., x:erased t → y:t { y == reveal x }. F⋆ provides implicit coercions to insert reveal in ghost contexts, so one doesn't usually need to write it explicitly. The online F⋆ book provides more details about erasure.[3]

Erasability in USSL. Erasure in USSL is important not just for efficiency—as mentioned earlier, we want to be able to compose atomic steps with ghost steps and have them be able to open invariants. As such, ensuring that ghost steps have no observable effects and can indeed be erased is important also for correctness. Our goal is to build a notion of erasure for USSL's stateful, concurrent computations from a simpler foundation of erasure of pure computations provided by F⋆. Towards that end, the signature of USSL expects an instance of the erasability class below.

```
class erasability (mem:Type u#a) (m:memory mem) (i:invariants mem) = {
  equiv_ghost : mem → mem → prop;
```

[3] https://fstar-lang.org/tutorial/book/part4/part4_ghost.html#erasure-and-the-ghost-effect

```
  update_ghost : (m0:mem → m1:erased mem { equiv_ghost m0 m1 } → m:mem { m == reveal m1 });
  equiv_ghost_preorder : preorder equiv_ghost; non_info_iname: non_info i.iname; }
```

The predicate equiv_ghost m_0 m_1 is valid when m_0 and m_1 only differ on erased state. In particular, update_ghost, given a memory m_0 and an erased memory m_1, can reconstruct an updated memory m that is equal to m_1—since it can use m_0 to construct all the non-erased parts of m. The equiv_ghost relation only needs to be a preorder, though commonly, it would also be an equivalence relation.

Top-level signature. Finally, we reach the top-level signature of the USSL semantics of memories and propositions, slsig, combining the elements we've defined so far. This mem type is equipped with a m:memory structure; an invariants structure; a notion of raw propositions; and support for erasability. We have a mem and slprop m in at least universe 1—the base heap can store objects in universe 0, so its predicates are in universe 1.

```
type slsig : Type u#(a + 2) =
{ mem : Type u#(a + 1);   m:memory mem;   i:invariants mem;   s:with_prev mem;   e:erasability mem m i }
```

We write prev s for s.s.prev; slprop s for slprop s.m; iname s for s.i.iname; and so on. We can also define storable propositions for a given signature:

```
let is_storable s (p:slprop s) = s.s.up (s.s.down p) == p    let storable s (p:slprop s) = p:slprop s {is_storable s p}
```

3.2 Actions

The next step of our formalization is to give a semantics to computations, in particular to the atomic and ghost actions that are the basic building blocks of USSL programs. Specifications of actions are given using an indexed state monad (Nanevski et al. 2008), st a pre post, the type of pure functions from initial states s_0:s satisfying the precondition pre, to results (x, s_1) that satisfy the postcondition post s_0 x s_1.

```
let st s a (pre: s → prop) (post: s → a → s → prop) = s0:s { pre s0 } → res:(a & s) {  post s0 res.1 res.2 }
```

It is easy to derive standard Hoare-style rules in st for returning pure values; for sequential composition; for strengthening preconditions and weakening postconditions; and for operations to get and put the state.

Based on this, we define the type of actions in USSL, frame-preserving, state-passing functions, whose state is a refined memory, rmem s. The signature below describes a total function, which for any frame is an st computation over rmem s, with result type a, and carrying four indexes: the flag ghost describes whether or not the action has observable effects on the state; opens is a set of invariant names that may be opened by the computation; req is the precondition; and ens is the postcondition.

```
let action (ghost:bool) (s:slsig) (a:Type u#a) (opens:set (iname s)) (req:slprop s) (ens: a → slprop s)
= except:set (iname s) { except ∩ opens = ∅ } → frame:slprop s → st (rmem s) a
  (requires λ m0 → inames_ok except m0 ∧ interpret (req ** frame ** mem_owns except m0) m0)
  (ensures λ m0 x m1 → (ghost ⟹ equiv_ghost m0 m1) ∧ inames_ok except m1 ∧
    interpret (ens x ** frame ** mem_owns except m1) m1)
where interpret p m = p (heap_of m) and inames_ok i m = ∀ n ∈ i. iname_ok i m
```

We write act for action false and ghost_act for action true. Further, most actions open no invariants—so, we usually omit the opens : set iname argument, rather than writing ∅ explicitly. We also write ghost_act s p q for ghost_act s unit p (λ _ → q), and similarly for act s p q.

For any signature s:slsig, we can prove that an erased ghost action with a non-informative result is itself non-informative, in the following sense:

```
val ghost_act_non_info (_:non_info a) (_:erased (ghost_act s a opens p q)) : ghost_act s a opens p q
```

The proof involves ghost-evaluating the erased action on an erased initial memory to obtain an erased result and erased final memory, and using the equiv_ghost relation to apply the update_ghost function to reconstruct the final memory. In other words, an erased ghost action can indeed be erased at runtime, since an equivalent counterpart can always be constructed.

3.3 A Concrete Heap

We now show how to instantiate the abstract signature slsig. As a roadmap of the technical development, we start by defining a core universe polymorphic concrete heap core : Type u#(a + 1). Next, we define a ghost heap erased core—all actions on a ghost heap are ghost, while only actions that do not modify the concrete heap are ghost. Combining the two, we define a base heap base_heap = core & erased core, which supports both concrete and ghost actions. Base heaps have only a degenerate notion of invariants—as a final step in the construction, we define a notion of heap extension, which adds a non-trivial notion of storable propositions and invariants.

We start with the type of core, shown below:

```
type cell : Type u#(a + 1) = | Cell : meta:erased bool → a:Type u#a → p:pcm a → v:a → cell
let core_heap = addr → option cell where addr = nat
let core = { heap_of:core_heap; ctr:nat }
```

Each cell contains a value v of a given type a, where a is the carrier of some p:pcm a. Modeling each cell of a heap as a PCM provides a flexible basis on which to define various sharing disciplines. For example, should a user wish to model the data layout of a C like language with structures and unions, USSL allows each abstract memory address to model an allocation unit, where the value stored is product or sum, chosen from some appropriate PCM to model the struct or union in question, together with some sharing discipline, e.g., different threads may own different fields of a struct. We also store one bit of metadata in each cell, which we will use to encode invariants.

A PCM for core_heap. The first step in instantiating the signature is to give a PCM on core_heap, easily defined as the pointwise lifting of the PCM at each cell, where cells are composable if they agree on their types, their PCMs and metadata, and the values are composable in the cell's PCM. In other words, two core heaps h_0, h_1 are disjoint if on every address a in the domain of both h_0 and h_1, the cells h_0 a and h_1 a are composable. With a PCM instance for core_heap, showing that core inhabits the memory class is easy.

Invariants are degenerate. As mentioned before, for core the notion of invariants is trivial. We just define iname = unit; inv i p = pure (p == emp); and iname_ok _ _ = ⊤. This makes the lemmas inv_iname_ok and dup_inv trivial to prove. For mem_owns ex m we state that all addresses greater than the counter are unallocated, i.e., pure (∀ i. i ≥ m.ctr ⟹ heap_of m i == None), which makes the main invariant lemma, mem_owns_equiv, also easy to prove.

Only emp is storable. We define the type prev as the singleton type in the appropriate universe and define up _ = emp and down _ = ().

Trivial ghost actions. Finally, since concrete heaps have no ghost state, we define equiv_ghost m_0 m_1 as m_0==m_1, which makes it trivial to give an instance of the erasability class for core.

Points-to Assertions. On core heaps, we define a points-to assertion as shown below, lifting notions from a PCM at a given cell to core heaps, i.e., pts_to meta r v asserts partial knowledge v over the contents of a memory cell at address r.

```
let ( ⪯ ) (x y:a) = ∃ frame. x .? frame ∧ x ⊙ frame == y
```

```
let ref (#t:Type u#a) (p:pcm t) = addr
let pts_to (meta:bool) #t (#p:pcm t) (r:ref t p) (x:t) : slprop core = λ (h:core_heap) → match h r with
| Some (Cell m _ p' y) → m==meta ∧ p == p' ∧ x ⪯ y | _ → ⊥
```

A reference r:ref p is just an abstract memory address addr, though we write its type as ref p just to indicate that it is a reference to a cell of type p:pcm a. This allows us to write pts_to meta r x and F★ can infer the implicit arguments #t and #p.

One can also prove the following actions, starting with extend to allocate a new heap cell—notice that the initial value must satisfy the PCM refinement:

```
val extend (meta:bool) (p:pcm) (x:a{p.refine x}) : act core (ref p) emp (λ r → pts_to meta r x)
```

Next, one can lift a refinement-preserving, frame-preserving update on a PCM to a heap cell:

```
let fp_upd (p:pcm a) (x y:a) = v:a{ p.refine v ∧ x ⪯ v } →
  u:a{ p.refine u ∧ y ⪯ u ∧ (∀ (f:a{x .? f}). y .? f ∧ (x ⊙ f == v ⟹ y ⊙ f == u))}
val upd (r:ref p) (f:fp_upd p x y) : act core unit (pts_to meta r x) (λ _ → pts_to meta r y)
```

Dereferencing a reference returns a refined value (v:a{p.refine v}), and initial knowledge of the ref-cell pts_to r x can be refined to pts_to r y, where y is compatible with the initial knowledge and the returned value v.

```
let fcompat (p:pcm a) (x v y:a) = ∀ (frame:a). (x .? frame ∧ v == x ⊙ frame) ⟹ (y .? frame ∧ v == y ⊙ frame)
let upd_knows (p:pcm a) (x:a) = v:a{x ⪯ v} → y:a{y ⪯ v ∧ fcompat p x v y}
val read (r:ref p) (f:upd_knows p x) : act core (v:a{p.refine v}) (pts_to meta r x) (λ v → pts_to meta r (f v))
```

The following ghost actions allow combining and sharing partial knowledge of a heap cell.

```
val gather (r:ref p) (x y:_): ghost_act core (_:unit{x .? y}) (pts_to m r x ** pts_to m r y) (pts_to m r (x ⊙ y))
val share (r:ref p) (x y:_{x .? y}): ghost_act core unit (pts_to m r (x ⊙ y)) (pts_to m r x ** pts_to m r y)
```

3.4 A Base Heap: Product of Concrete and Ghost Heaps

With a core heap at hand, we can define the base heap as a product of base = core & erased core, where base_heap = core_heap & erased core_heap.

To define a PCM on base_heap, one can easily define pcm_erase (p:pcm a) : pcm (erased a), since, as explained earlier, a and erased a are isomorphic using hide and reveal. Further, one can take the product of PCMs, pcm_prod (p:pcm a) (q:pcm b) : pcm (a & b). Using both of these, we can define pcm_base : pcm base_heap as pcm_prod pcm_core (pcm_erase pcm_core).

For invariants on base, we chose just trivial invariants, as we did with core, and, similarly, we only define trivial storable propositions.

Defining erasability is more interesting. We define equiv_ghost b_0 b_1 == fst b_0 == fst b_1, allowing the erased heap to change arbitrarily, easily showing it to be a preorder. For update_ghost b_0 b_1 we define it as (fst b_0, hide (reveal (snd b_1))).

Affine predicates on the components of a PCM product can be lifted to affine predicates on the product itself. This lets us define a pts_to and ghost_pts_to predicates on base heaps, assertions about concrete and ghost state, respectively, simply by lifting the pts_to of each side. Note, we only use the meta field on the ghost heap, so the meta field of the points-to assertion on concrete references is always false.

```
val lift_fst: slprop h → slprop (pcm_prod h g)        val lift_snd: slprop g → slprop (pcm_prod h g)
let pts_to r x = lift_fst (pts_to false r x)
let ghost_pts_to meta (r:ghost_ref p) (x:a) = lift_snd (pts_to meta r x) where ghost_ref p = erased (ref p)
```

Similarly, actions on core lift to actions on base, while actions on erased core lift to ghost actions on base. For example, we have:

```
val upd (r:ref p) (f:fp_upd p x y) : act base (pts_to r x) (pts_to r y)
val ghost_upd (r:ghost_ref p) (f:fp_upd p x y) : ghost_act base (ghost_pts_to m r x) (ghost_pts_to m r y)
```

3.5 Heap Extension: The Essence of Stratification

Next, we show how to extend a signature s:slsig u#a to a signature extend s : slsig u#(a + 1), where the prev propositions of (extend s) are the slprops of s, yielding a construction that supports the dynamic allocation of non-trivial invariants. That is, our initial goal is to define:

```
val extend (s:slsig u#a) : t:slsig u#(a + 1) { t.s.prev == slprop s }
```

Extended memories and heaps. The key idea is to define a type of memories that is a product of s.mem and a base at the next universe level, i.e., ext_mem_t s = s.mem & base u#(a + 1), where heap_of m : ext_heap s is (heap_of (fst m), heap_of (snd m)). Obtaining a PCM on ext_heap s is straightforward, as it is just the product of the PCMs on s.m.heap and base.heap. We write ext_mem s for the memory built on ext_mem_t s. As before, products of PCMs also yield products of affine predicates, so we have lift_fst and lift_snd to lift predicates on s.m.heap and base.heap to ext_heap s.

Liftable predicates. We define the prev type on ext_mem s as slprop s, where converting between slprop s and slprop (ext_mem s) via up and down and proving down (up p) = p is straightforward.

```
let ext_prev (s:slsig u#a) : with_prev (ext_mem s) =
{ prev = slprop s;     up = (λ p h → p (fst h));     down = (λ p h → p (h, empty));     up_down = . . . }
```

Invariants. A dynamically allocated invariant inv i p is represented by a ghost cell at address i in the base u#(a + 1) heap, where the cell stores the proposition p from a PCM that supports duplicable knowledge of the cell's content. Ghost cells that store invariant propositions are distinguished from other ghost cells by setting their metadata bit. However, as we also wish to preserve invariants from the signature s:slsig u#a, we define invariants as follows:

```
let ext_iname s = erased (either (iname s) (ghost_ref (pcm_dup (slprop s))))
let ext_iname_ok s i (h0, h1) = match i with | Inl i → iname_ok s i h0 | Inr i → valid_ghost_addr h1 i
```

Invariant names iname are either names from s or ghost references to cells in pcm_dup (slprop s). In the PCM pcm_dup a, we have x .? y only if either x or y are the unit element or they are equal; and composition of non-unit elements x ⊙ y is just x (since x == y). An invariant name is valid (iname_ok) when either the iname_ok of s is valid, or that the ghost reference is valid in h_1.

Next, ext_inv s defines the main invariant former, where ext_inv s i p is either an invariant in s or asserts knowledge of the ghost cell containing down p. In both cases, p must be storable—in this definition up, down, and is_storable are with respect to ext_prev s, the extended signature being defined.

```
let ext_inv s i p = match i with
  | Inl i → up (inv i (down p)) ** pure (is_storable p)
  | Inr i → lift_snd (ghost_pts_to true i (Some (down p))) ** pure (is_storable p)
```

The main internal invariant, mem_owns is defined below, and has three components: the lifting of the mem_owns from s applied to the first component of the memory (where left_of ex is the projection of a set (either a b) to set a); the lifting of the mem_owns from base applied to the second component of the memory; and the main interesting part being the third conjunct, invs_except.

```
let ext_mem_owns s ex (m0,m1) = lift1 (mem_owns #s (left_of ex) m0) ** lift2 (mem_owns #base {} m1)
                        ** invs_except (ghost_ctr m1) ex m1
```

The invs_except predicate is an iterated separating conjunction of inv_of_cell for every address in the ghost heap of m_1: if the cell has its meta bit set then either the cell's address is in the exclusion set ex, or it's PCM is pcm_dup (slprop s) and it contains a proposition p : slprop s such that up p is valid. In other words, invs_except asserts ownership over all invariants stored in m_1 except those in the exclusion set ex.

Proving the lemmas inv_iname_ok and dup_inv is relatively easy; mem_owns_equiv, the main lemma, is more involved, but the proof follows from the corresponding lemmas from s and base, together with an induction on the iterated conjunction in invs_except.

Erasability. To complete our construction of extend s, we need to give an instance of the erasability class. Defining equiv_ghost just follows from the conjunction of equiv_ghost on s and base_heap, and the associated lemmas are easy to prove.

Other properties. Our construction satisfies several other useful properties, notably that up and down commute with the logical connectives of slprop, including up_inv, which proves that lifting invariants on s produces invariants on extend s.

```
val down_star s (p q:slprop (extend s)) : Lemma (down (p ** q) == down p ** down q)
val up_star s (p q:slprop s) : Lemma (up (p ** q) == up p ** up q)
val down_emp s : Lemma (down emp == emp)
val up_emp s : Lemma (up emp == emp)
val up_inv s i p : Lemma (up (inv s i p) == inv (extend s) (lift_iname i) (up p))
```

These lemmas allow one to also prove useful congruences on storable propositions, e.g., the conjunction of two storable propositions is storable. Such lemmas, in conjunction with F⋆'s refinement types and SMT based automation, make it easy to construct complex storable propositions from basic building blocks like the storable points-to assertion.

```
let storable_star (s:slsig) (p q:storable s) : Lemma (is_storable s (p ** q))
```

Actions on extended heaps. As with our product of core and erased core, actions and propositions on s and base lift to actions and propositions on extend s. For example, we gain two new points-to predicates, for storing concrete or ghost values in the extended heap:

```
let ext_pts_to (#a:Type u#(a + 1)) (#p:pcm a) (r:ref p) (x:a) = lift_snd (pts_to r x)
let ext_ghost_pts_to (#a:Type u#(a + 1)) (#p:pcm a) (r:ghost_ref p) (x:a) = lift_snd (ghost_pts_to false r x)
```

More significantly, we can define the main interface to invariants, starting with new_invariant, a ghost action that allows allocating any storable proposition as an invariant, writing ŝ for (extend s).

```
val new_invariant s (p:slprop ŝ { is_storable ŝ p }) : ghost_act ŝ iname p (λ i → inv ŝ i p)
```

We can also define storable invariants for doubly storable propositions, effectively lifting invariant creation on s to ŝ, where storable_iname i asserts that i:iname ŝ is just a lifting of an iname s.

```
val new_storable_invariant s (p:slprop ŝ) { is_storable s (down p) ∧ is_storable ŝ p })
: ghost_act ŝ (i:iname ŝ { storable_iname i }) p (λ i → inv ŝ i p)
val storable_inames_are_storable s (i:iname ŝ { storable_iname i }) (p:slprop ŝ) { is_storable p })
: Lemma (is_storable ŝ (inv ŝ i p))
```

Next, dup_inv is a ghost action that allows duplicating an invariant:

```
val dup_inv s i p : ghost_act ŝ unit (inv ŝ i p) (λ _ → inv ŝ i p ** inv ŝ i p)
```

And, finally, with_invariant allows opening an invariant i, running an action f, and restoring the invariant, so long as the f does not internally open i again—the entire step is ghost when f is ghost.

```
val with_invariant #s (i:_{ i ∉ opens }) (f:action is_ghost s a opens (p ** fp) (λ x → p ** fp' x))
: action is_ghost s a ({i} ∪ opens) (inv s i p ** fp) (λ x → inv s i p ** fp' x)
```

The proof outline of with_invariant below shows how mem_owns_equiv is the key lemma, where by assumption {i} ∪ opens is disjoint from the exclusion set ex, where the signature s implicit.

precondition: frame ** (inv i p ** fp ** mem_owns ex m_0) *by* mem_owns_equiv
frame ** (inv i p ** fp ** p ** mem_owns ({i} ∪ ex) m_0) *by* rearrange
(frame ** inv i p) ** (p ** fp ** mem_owns ({i} ∪ ex) m_0) *by run action* f *with* ({i}∪ex) ∩ opens = ∅
(frame ** inv i p) ** (p ** fp' x ** mem_owns ({i} ∪ ex) m_1) *by* mem_owns_equiv
frame ** (inv i p ** ** fp' x ** mem_owns ex m_1) *postcondition*

This concludes the core formalization of the logic. We have built a generic separation logic with concrete and ghost state, with storable propositions and actions that enable dynamically allocating invariants.

3.6 Derived Connectives

Our core logic provides only the basic connectives of separation logic: several variants of pts_to, **, and emp. In this section we show how to derive a variety of other connectives—we work with an implicit signature s.

3.6.1 Points-to with fractional permissions. The points to predicate of the core logic is generic in a PCM associated with a heap cell. This provides the flexibility needed to encode a variety of spatial and temporal sharing disciplines. For example, one can easily code up points-to predicates with fractional permissions (Boyland 2003); in §4, we look at other PCM-based ghost state constructions.

The PCM of fractions, pcm_frac a : pcm (option (a & r:real{r>0})), has unit element None, and non-unit elements are composable if they agree on their values and their real-valued permissions sum to at most 1; and composition sums permissions. A points-to predicate with fractional permissions pts_to_frac r p v is just an instance of the core points-to predicate, using the PCM of fractions.

```
let pts_to_frac (#a:Type) (x:ref a) (p:real{p>0}) (v:a) = pts_to #a #(pcm_frac a) x (Some (v, p))
```

The usual actions to share and gather fractional permissions follows directly, by instantiating the PCM-generic share and gather actions shown in §3.3—we write $x \mapsto v$ for pts_to_frac x 1 v and and $x \overset{p}{\mapsto} v$ for pts_to_frac x p v.

$$\text{val share : ghost\_act } (x \overset{p+q}{\mapsto} v)\ (x \overset{p}{\mapsto} v \ast\ast\ x \overset{q}{\mapsto} v)$$
$$\text{val gather : ghost\_act } (x \overset{p}{\mapsto} u \ast\ast\ x \overset{q}{\mapsto} v)\ (x \overset{p+q}{\mapsto} u \ast\ast \text{ pure } (u{=}{=}v \land p{+}q \le 1))$$

3.6.2 Iterated Conjunction. Being embedded in F*, USSL is naturally extensible with additional connectives. For instance, an iterated conjunction defined by recursion is shown below, including several ghost functions to manipulate on_range predicates:

```
let rec on_range (p: (nat → slprop)) (i j: nat) : Tot slprop (decreases (if j ≤ i then 0 else j – i))
= if j < i then pure ⊥ else if j = i then emp else p i ** on_range p (i + 1) j
val on_range_empty : ghost_act emp (on_range p i i)
val on_range_singleton_intro : ghost_act (p i) (on_range p i (i+1))
val on_range_singleton_elim : ghost_act (on_range p i (i+1)) (p i)
val on_range_join : ghost_act (on_range p i j ** on_range p j k) (on_range p i k)
val on_range_j : ghost_act (on_range p i j ** pure (i ≤ j ∧ j ≤ k)) (on_range p i j ** p j ** on_range p (j+1) k) . . .
```

3.6.3 Existential Quantification. We define existential quantification as an affine predicate, as shown below. This quantifier is impredicative, meaning that one can quantify over values in any universe a:Type u#a, relying on the impredicativity of the propositional existential quantifier.

```
let ( ∃* ) (#a:Type u#a) (p: a ⟶ slprop) = λ m ⟶ ∃ x. p x m
```

Ghost actions to introduce and eliminate existentials are shown below—the elimination is the lifting of the propositional axiom of indefinite description.

```
val intro_∃ : ghost_act (p x) (∃* x. p x)
val elim_∃ : ghost_action (erased x) (∃* x. p x) (λ x ⟶ p x)
```

One can also prove the following lemma, showing that an existentially quantified storable proposition is itself storable.

```
val ∃_storable (p:(a ⟶ slprop) { ∀ x. is_storable (p x) }) : Lemma (is_storable (∃* x. p x))
```

3.6.4 Trades & Universal Quantification. A magic wand is easy to define:

```
let ( -* ) (p q : slprop) = λ m0 ⟶ ∃ r. r m0 ∧ (∀ m1. m0 .? m1 ∧ p m1 ⟹ q (m0 ⊙ m1))
```

However, we define variations on the magic wand that encapsulate ghost actions and are more idiomatic for use in USSLang.

First, analogous to a view-shift in Iris, one can define the following abstraction of an invariant-opening ghost function that transforms assertions on the ghost state from p to q.

```
let shift opens p q = ∃* (f:ghost_act unit opens p (λ _ ⟶ q)). emp
```

The introduction rule for a shift is just the introduction for the corresponding existential quantifier. The elimination rule is a bit more interesting, and is implemented by eliminating the existential associated with the shift, then applying the ghost function to transform the p to q. As such, shifts provide a "logical" way to manipulate ghost functions.

```
val intro_shift (f:ghost_act opens p q) : ghost_act opens emp (shift opens p q)
val elim_shift : ghost_act opens (shift opens p q ** p) q
```

Shifts are both duplicable and storable, since they hold only the trivial proposition emp. We also define a connective called a "trade", which is a resource-holding analog of a shift. Trades have the same elimination principle as shifts, but they are not duplicable in general, since they may hold a non-trivial resource r. They are also not storable in general, though one can also define a variant, a storable_trade, that holds a storable resource r.

```
let trade opens p q = ∃* (r:slprop) (f:ghost_act opens (r ** p) q). r
val intro_trade (r:slprop) (f:ghost_act opens (r ** p) q) : ghost_act opens r (trade opens p q)
val elim_trade : ghost_act opens (trade opens p q ** p) q
val weaken_opens : ghost_act (trade o1 p q ** pure (o1 ⊆ o2)) (trade o2 q r)
val trans : ghost_act (trade o p q ** trade o q r) (trade o p r)
```

Trades are weaker than magic wands, in the sense that p -* q implies trade o p q; trades are also weaker than shifts. In a similar style, one can also define universal quantification as an existential abstraction of a ghost function.

```
let fa (#a:Type u#a) opens (p:a ⟶ slprop) = ∃* (r:slprop) (f:(x:a ⟶ ghost_act opens r (p x))). r
val intro_∀ (f:(x:a ⟶ ghost_act opens r (p x))) : ghost_act r (fa opens p)
val elim_∀ (x:a) : ghost_act opens (fa opens p) (p x)
```

Abstracting ghost functions into logical connectives yields a proof style that enables switching between the logical flavor of systems like Iris with the explicit ghost-function passing style of systems like Verifast (Jacobs and Piessens 2011).

3.7 Representing Concurrent Computations

The last step in our formalization is to extend the logic beyond just individual actions and ghost actions and apply it to general-purpose concurrent programs. This is relatively straightforward, by representing computations as trees of actions, one sub-tree for each thread. The semantics of an entire computation tree is given by an interpreter which repeatedly picks a thread (non-deterministically) and evaluates that thread's next action—in other words, the dynamic semantics of is an interleaving of atomic actions from threads modeling a sequentially consistent machine.

The type of computation trees is stt a p q, shown below. Intuitively, these trees represent the runtime configuration of a partially reduced program with precondition p, return type a, and postcondition q. The term Ret v represents a computation fully reduced to a value. Act f k is a computation that begins with the action f and then continues with k. Par m_0 m_1 k represents m_0 and m_1 executing concurrently, with the continuation k waiting until they have both completed.

```
type stt (s:slsig) : a:Type u#a → slprop s → (a → slprop s) → Type =
| Ret: x:a → stt s a (p x) p
| Act: f:act s o p q → k:(x:b → Dv (stt s a (q x) r)) → stt s a p r
| Par: m0:stt s (raise_t unit) p0 q0 → m1:stt s (raise_t unit) p1 q1 → k:stt s a (q0 ** q1) q → stt s a (p0 ** p1) q
```

In the type of Act f k, the continuation k is a function in F⋆'s effect of divergence. That is, the continuation is allowed to potentially loop forever—this is the essential bit that allows us to give a partial correctness semantics for USSL, enabling programming functions such as acquire to loop indefinitely until the lock becomes available.[4] Note also that we only have one kind of action node, Act, rather than a separate one for ghost actions. This is because, as discussed in §3.2, using ghost_act_non_info, an erased ghost action can be promoted to a ghost action with the same type, and then subsumed to an action. Another subtlety is in Par m_0 m_1 k, the sub-trees m_0 and m_1 return unit values raised to a given universe u#a—this is a technicality that allows the definition to be properly universe polymorphic in the result type. With Par, we model structured parallelism only, since in conjunction with synchronization libraries like locks, it can be used to encode other forms of concurrent control, though in the future one could imagine extending the representation of trees with other forms of concurrent composition.

To interpret computation trees, we work in an extension of the st monad that provides an additional action called flip that consumes a boolean from an infinite input tape of randomness. The signature of the interpreter is shown below, reflecting our main partial correctness theorem: given a source of randomness (t:tape) a computation tree f:stt s a pre post can be interpreted as a potentially divergent state-passing function from input states m_0 validating its precondition and the invariant, to results and final states m_1 validating the postcondition.

```
val run s (t:tape) (f:stt s a pre post) (m0:rmem s { interpret (pre ** mem_owns ∅ m0) m0 })
: Dv (res:(a & rmem s) { let x, m1 = res in interpret (post x ** mem_owns ∅ m1) m1 })
```

The implementation of run is relatively straightforward, especially for Ret and Act nodes. For Par nodes, it consumes a bit from the tape, and recurses into one of the subtrees accordingly.

---

[4]It is also possible to require k to be a total function, yielding a total correctness semantics for, say, non-blocking concurrent programs with structured parallelism. Alternatively, instead of the effect of divergence, one could also define a coinductive semantics in the style of interaction trees (Xia et al. 2019), and indeed a library for interaction trees exists in F⋆ (https://github.com/RemyCiterin/CoIndStar/tree/main). However, deriving a separation logic for interaction trees would require a significantly more work than the compact, intrinsically typed approach we use here.

One can also derive the following combinators.

val frame (fr:_) (f:stt s a p q) : Dv (stt s a (p ** fr) (λ x → q x ** fr))
val bind (f:stt s a p q) (g: (x:a → stt s b (q x) r)) : Dv (stt s b p r)
val par (m_0:stt s unit p0 q0) (m_1:stt s unit p1 q1) : Dv (stt s unit (p0 ** p1) (q0 ** q1))

While one can actually execute programs using run, in practice, we rely on run only to provide a semantic basis, and instead execute concurrent USSLang programs by extracting them to OCaml, C, or Rust using efficient, native concurrency primitives.

4 CASE STUDIES

Although the USSL logic and actions are generic in a signature, for USSLang, we fix a signature sig_3 where sig_0 = base u#0 and sig_{i+1} = extend sig_i, reflecting the naming convention shown in Figure 1. Fixing a signature makes the design of libraries relatively simple, though perhaps not as general as embracing signature-polymorphism throughout—we leave signature polymorphism in USSLang as a topic to explore in the future. We can now define:

let $slprop_4$: Type u#4 = slprop sig_3
let $slprop_3$: Type u#3 = slprop sig_2 let $\uparrow_3$ = sig_3.s.up let $\downarrow_3$ = sig_3.s.down
let $slprop_2$: Type u#2 = slprop sig_1 let $\uparrow_2$ = sig_2.s.up let $\downarrow_2$ = sig_2.s.down
let $slprop_1$: Type u#1 = slprop sig_0 let $\uparrow_1$ = sig_1.s.up let $\downarrow_1$ = sig_1.s.down
let slprop = $slprop_4$
let storable = p:slprop{ is_storable sig_3 p }
let $storable_2$ = p:slprop { is_storable sig_3 p ∧ is_storable sig_2 ($\downarrow_3$ p) }
let $storable_3$ = p:slprop { is_storable sig_3 p ∧ is_storable sig_2 ($\downarrow_3$ p) ∧ is_storable sig_1 ($\downarrow_2$ ($\downarrow_3$ p)) }

The types $storable_3$ ≤ $storable_2$ ≤ storable ≤ slprop, are related by refinement subtyping. This lets us use just one set of connectives, e.g., we have (**) : slprop → slprop → slprop, though with lemmas that allow us to prove that p ** q is storable if p and q are storable, and likewise for $storable_2$. This means that connectives like ** and ∃* are overloaded, e.g., (**) can also be typed at storable → storable → storable and $storable_2$ → $storable_2$ → $storable_2$, providing a flavor similar to intersection types.

We also have several kinds of PCM-generic points-to predicates, for universes 0–3. We also have the corresponding $ghost\_pts\_to_0$–$ghost\_pts\_to_3$ predicates.

val pts_to_0 (#a:Type u#0) (#p:pcm a) (r:ref p) (v:a) : $storable_3$
val pts_to_1 (#a:Type u#1) (#p:pcm a) (r:ref p) (v:a) : $storable_2$
val pts_to_2 (#a:Type u#2) (#p:pcm a) (r:ref p) (v:a) : storable
val pts_to_3 (#a:Type u#3) (#p:pcm a) (r:ref p) (v:a) : slprop

We also overload the invariant type inv i p, so that it can be storable when i is a storable iname, constructed using new_storable_invariant shown in §3.5. As such, with refinement subtyping, one can allocate invariants of the form inv i (inv j p), provided inv j p is storable, without needing to deal with multiple distinct variants of the inv type.

One may wonder why we chose four levels—pts_to_0 is obviously useful for the concrete heap, pts_to_1 is useful to store values of sigma types like (t:Type u#0 & f t), which is occasionally useful; pts_to_2 is for storing predicates about memory locations containing sigma types; and $slprop_4$ is useful for invariants over everything else. Of course, we could easily add more levels in the future.

The following provides a mapping between USSLang and USSL function signatures:

ghost fn f ... : requires p returns x:t ensures q opens o $\triangleq$ f ... : erased (ghost_act sig_3 t o p (λ x → q))
atomic fn f ...: requires p returns x:t ensures q opens o $\triangleq$ f ... : act sig_3 t o p (λ x → q)
fn f ... : requires p returns x:t ensures q $\triangleq$ f ... : stt sig_3 t p (λ x → q)

We have used USSLang to build a variety of verified libraries and applications, as summarized in the table alongside, representing about 20,000 lines of code and proof. Our code includes basic libraries for references, arrays, and a model of heap and stack allocation; derived combinators including the connectives of §3.6; libraries of PCM constructions; data structures hash tables, linked lists, a double-ended queue, utilities on arrays; and various concurrency-related libraries, including mutexes and barriers. We also report on an implementation of DPE, a low-level, cryptographic measured boot protocol service with support for concurrent sessions. Independently of the work reported here, USSLang has also been used to model of task-parallel programs, in the development of parsing and serialization libraries, and other related efforts, giving us some confidence that abstractions provided by USSL are sufficient for a range of verified software applications. In the space that remains, we focus on describing two small case studies that show aspects of USSL at work, followed by an overview of our implementation of DPE.

Basic	4,334
Derived combinators	1,695
PCMs	1,696
Data Structures	6,900
Concurrency	2,490
DPE protocol	3,673
Total (LOC)	20,888

4.1 An n-way Parallel Increment with Dependently-typed Ghost State

Owicki & Gries show how to prove that a program correctly adds 2 to an integer reference by atomically incrementing it twice in parallel. Their proof, repeated in innumerable variations in many systems, makes essential use of ghost state, to track the contributions of each thread, together with an invariant stating the current value of the reference is the initial value plus the sum of the contributions. We show how to do this proof in USSLang, with a twist: our construction works generically for an arbitrary number of n threads each incrementing the reference in parallel, and illustrates the use of invariants over custom dependently-typed PCMs for ghost state. Of course, incrementing a reference in parallel is just an idealization of the more general problem of reasoning about multiple threads mutating a shared data structure.

For starters, we assume an atomic primitive to increment an integer reference (ignoring that an integer increment may overflow—handling overflow is orthogonal to the point of this example).

```
atomic fn atomic_incr (r:ref nat) : requires r ↦ i ensures r ↦ (i + 1)
```

Next, we define for any `n: nat`, a PCM `pcm_of n` based on the commutative monoid `(nat, 0, +)` on natural numbers, where `(pcm_of n).refine x = (x==n)`. One could also do this example in other ways, but we show this style to illustrate a simple use of a refined PCM as mentioned at the start of §3.1. The type `tank n` below is the building block of our ghost state, a ghost reference to a `pcm_of n`.

```
let pcm_of n = pcm_of_cm nat_plus_cm n        let tank (n:nat) = ghost_ref (pcm_of n)
```

The assertions we can form on `r:tank n` are of the form `own_units r i = ghost_pts_to₀ r i`, and since `i` must always be compatible with a "full value" for `pcm_of n`, we can prove from `own_units r i` that `i ≤ n`. Further, we can convert between `owns_unit r i ** own_units r j` and `own_units r (i + j)`, with `share_units` and `gather_units`, respectively.

```
ghost fn own_units_bound (r:tank n) requires own_units r i ensures own_units r i ** pure (i ≤ n)
ghost fn share_units (r:tank n) requires own_units r (i + j) ensures own_units r i ** own_units j
ghost fn gather_units (r:tank n) requires own_units r i ** own_units r j ensures own_units r (i + j)
```

The main idea of the proof is captured by the following `contribs` predicate—notice that F★'s refinement typing automatically proves that it is storable. The ghost state is represented (below) by two n-sized tanks: `given`, which tracks how many units `g` have already been contributed to the current value of the reference `r` (i.e., `v==init+g`); and `to_give`, which tracks how many units are still outstanding. The sum of what has been given and what is still outstanding is always `n`.

```
type ghost_state (n:nat) = { given:tank n; to_give:tank n }
let contribs n init (gs:ghost_state n) (r:ref int) : storable =
 ∃* (v g t:nat). r ↦ v ** own_units gs.given g ** own_units gs.to_give t ** pure (v==init+g ∧ g+t==n)
```

The proof itself involves shuffling knowledge of the ghost state. The specification from the perspective of the threads is the dual of the specification of the invariant. For a thread specification, we write can_give gs n to mean own_units gs.given n and has_given gs n to mean own_units gs.to_give n.

First, we initialize the ghost state, giving each thread ownership of one unit of the given tank.

```
ghost fn init n r  requires r ↦ i returns gs:ghost_state n  ensures contribs n i gs r ** can_give gs n
```

A key property of the ghost state is the following lemma: if the threads can prove has_given gs n, then the reference has been incremented *n* times—since we know that contribs owns t units of the to_give tank, we can conclude that t==0, since n+t≤n and hence g==n.

```
ghost fn finish gs requires contribs n i gs r ** has_given gs n ensures r ↦ (i+n)
```

As each thread increments its reference, it gives one unit of ownership of the given tank to contribs; and takes one unit of ownership from the to_give tank for has_given gs 1

```
atomic fn incr_core gs r requires can_give gs 1 ** contribs n i gs r
                  ensures has_given gs 1 ** contribs n i gs r {
 unfold contribs; unfold can_give; gather_units gs.given;
 atomic_incr r; (* the actual increment *)
 share_one_unit gs.to_give; fold (has_given gs 1); fold (contribs n i gs r); }
```

At the top-level we initialize the ghost state, allocate contribs n i gs r as an invariant (using a library for "cancellable invariants", which we describe shortly), then spawn n threads by recursively using the par combinator shown in §3.7; cancel the invariant and call finish.

```
fn incr_n (r:ref nat) (n:nat) requires r ↦ i ensures r ↦ (i+n) {
 let gs = init n r; let ci = CI.new_cancellable_invariant (contribs n i gs r);
 incr_n_aux r n ci; (* spawn n increment threads *)
 CI.cancel ci; finish gs r gs; }
```

About cancellable invariants: the library CI provides a way to allocate an invariant tracked by a fractional permission (CI.active i p), which can be cancelled by a thread that has full ownership of the invariant. Using this library, we wrap incr_core so that it can be called in parallel, as shown below.

```
atomic fn increment r (i:CI.inv)
requires can_give gs 1 ** CI.active i p ** inv (CI.iname_of i) (CI.cinv i (contribs n i gs r))
ensures has_given gs 1 ** CI.active i p ** inv (CI.iname_of i) (CI.cinv i (contribs n i gs r))
opens {CI.iname_of i} { with_invariants (CI.iname_of i) { CI.unpack i; incr_core gs r; CI.pack i; } }
```

Using dependently typed ghost-state, the code for increment including its ghost code, is identical for every thread, and generalizes to an arbitrary number of threads. In contrast, other proofs have required a separate proof for each thread, or parameterizing each thread by the ghost code and proving and instantiating that ghost code differently for each thread (Jacobs and Piessens 2011).

4.2 Barriers & Higher-order Ghost State

Jung et al. (2016), Dodds et al. (2016) and others show how to specify and verify a barrier library in impredicative logics. The interface of a barrier, adapted to USSLang syntax, is shown below:

```
val t : Type u#0                                          val iname_of (_:t) : iname
val send (b:t) (p:slprop) : slprop                        val recv (b:t) (p:slprop) : slprop
fn signal (b:t) requires send b p ** p ensures emp        fn wait (b:t) requires recv b p ensures p
ghost fn split (b:t) requires recv b (p ** q) ensures recv b p ** recv b q opens {iname_of b}
fn icreate (p:slprop (* ideal *)) requires emp returns b:t ensures send b p ** recv b p
```

We have an abstract type t representing a barrier, which holds an invariant iname_of t. The predicate send b p gives the signalling thread permission to send p over b; while a thread with recv b p blocks using wait until the signal is received and can then use the received p. The most interesting part of this interface is the ghost function split: if a thread holds recv b (p ** q), it can split it into recv b p and recv b q and share that among multiple threads. The tricky bit is that doing so requires knowing that when the barrier is signalled with p, multiple threads waiting on the barrier can safely proceed with their own separate fragment of p—we need some ghost state accounting for how much of p has already been handed out to waiting threads. In other words, we need ghost state which stores a proposition, aka higher-order ghost state.

Now, the ideal interface for a barrier would provide a function icreate that allows creating a barrier for any p:slprop—this is what Jung et al. accomplish with impredicativity in Iris. However, with USSL, we cannot provide this interface, since we cannot store an arbitrary slprop. But, with stratification and dependent types, we can come close.

4.2.1 A Code-Generic Interface for Barriers. We start by showing an interface to barriers that is parameterized by a language of codes. A c:code is a type t : Type u#2 that can be "decoded" into a storable proposition, with at least a code z for the empty proposition. A p:slprop is codeable if there is a code that decodes to p.

```
type code : Type u#4 = { t : Type u#2; up : t → storable;  z : t{ up z == emp } }
class codeable (code:code) (p:slprop) = { c : code.t; laws : squash (code.up c == p) }
```

Using codes, we can define the following interface to barriers, which is nearly identical to our ideal interface above, except that the type of a barrier is now indexed by a language of codes, c:code.

```
val ct (c:code) : Type u#0                                          val iname_of (b:ct c) : iname
val send (b:ct c) (p:slprop) : slprop                               val recv (b:ct c) (p:slprop) : slprop
fn signal (b:ct c) requires send b p ** p ensures emp               fn wait (b:ct c) (p:slprop) requires recv b p ensures p
```

Where things are different, however, is in the interface to create and split. For create, we can allocate a barrier for any codeable proposition p; and for split, we can split a recv b (p ** q) if we are given codes for p and q.

```
fn create (p:slprop) (_:codeable c p) requires emp returns b:ct c ensures send b p ** recv b p
ghost fn split (b:ct c) (cq:codeable c p) (cr:codeable c q)
requires recv b (p ** q) ensures recv b p ** recv b q opens {iname_of b}
```

The implementation of this interface is interesting, taking a few hundred lines of USSLang—so we only provide a glimpse of its main invariant.

A barrier b:ct c is represented by a single concrete reference r, and a ghost map from numbers to fractionally owned codes. The ghost ref ctr is a high water mark for the map, above which the map is empty. The name of the barrier invariant is stored in the last field i.

```
type ct (c:code) = { r:ref U32.t;  ctr:ghost_ref nat; map:ghost_ref (pointwise nat (pcm_frac c.t)); i:iname }
```

The invariant, shown below, says that the map is stored in a universe 2 ghost reference and holds full permission to the map above the high-water mark n, and half permission to the rest of the map m. If the reference r is not yet set (v=0), then the iterated conjunction of the decoding of the map elements in m is equal to the p; otherwise, the invariant owns the iterated conjunction of the decoding—again, F* automatically proves that ct_inv is storable.

```
let ct_inv (b:ct c) (p:slprop) : storable = ∃* v n m. b.r ↦.5 v ** b.ctr ↦ n **
  ghost_pts_to2 b.gref m ** pure (all_perms m 0 n 0.5) ** ghost_pts_to2 b.gref (full_map_above n) **
  (if v=0 then pure (p==on_range (predicate_at m) 0 n) else on_range (predicate_at m) 0 n)
```

The send predicate simply holds the invariant and half ownership to the un-set reference—allowing the signaling thread to transfer ownership of p to the invariant by flipping the reference. The recv b p predicate holds the invariant and knowledge of an entry n in the map that holds a code that decodes to p, allowing a waiting thread to learn that when the flag is set, the invariant holds an iterated conjunction with p at index n. It can then clear the entry at n setting it to c.z the code for emp, and return p to the waiting thread. Splitting a recv b (p ** q) involves clearing the index n and setting codes for p and q at fresh indexes, and moving the ctr up accordingly.

```
let send b p = inv b.i (ct_inv b p) ** b.r ↦.5 0
let recv b p = ∃* n k. inv b.i (ct_inv b p) ** ghost_pts_to2 b.map (singl n .5 k) ** pure (c.up k == p)
```

4.2.2 Using the Code-Generic Interface. Using the code generic interface with storable$_2$ propositions is easy, since USSL already effectively provides slprop$_2$ as a code for them:

```
let free_code = { t = slprop2; up = ↑3 ∘ ↑2; z=↓2 ↓3 emp }
let free_code_of (p:storable2) : codeable free_code p = { c = ↓2 ↓3 p; laws = () }
```

This suffices for most common cases, i.e., communicating ownership of propositions over heap cells with universe 0 or 1 values. However, the predicates send b p and recv b p are not storable$_2$, since they hold invariants and recv holds a permission to b.map, which is storable, though not storable$_2$. So, if one wanted to communicate permissions to one barrier b over another barrier b', we either need to work another heap layer (creating a barrier first to communicate storable$_3$ predicates, which could then be communicated once over another barrier etc.), or, using dependent types, we can just come up with a custom language of codes, as we'll see next.

First, we prove that the send (and recv) predicates can be decomposed into a storable part send_core (resp. recv_core) and a duplicable part.

```
val binv (b:ct c) (p:slprop) : slprop                val send_core b : storable
ghost fn dup_binv b p requires binv b p ensures binv b p ** binv b p
ghost fn decompose_send b requires send b p ensures binv cv p ** send_core b
ghost fn recompose_send b requires binv b p ** send_core b ensures send b p
```

Next, we can define a custom language of codes in u#2 covering send_core and recv_core, which would otherwise only be representable in u#3.

```
type cc (c:code) = | Small of c.t | Star : cc c → cc c → cc c | Send : ct c → cc c |  Recv : ct c → c.t → cc c
let rec up_cc #c (t:cc c) : storable = match t with
  | Small s → c.up s | Star c1 c2 → up_cc c1 ** up_cc c2
  | Send cv → send_core cv | Recv cv s → recv_core cv (c.up s)
let cc_code (c:code) : code = { t = cc c; up = up_cc; z = Small c.z }
```

Finally, we can use this custom language of codes to create a barrier that can communicate permissions to another barrier.

```
fn barrier2 (p:slprop2) requires p ensures p
{ let b1 = create p (free_code_of p); let b2 = create (send_core cv1) (code_of_send_core b1);
 decompose_send b1; signal b2 || (wait b2; recompose_send b1; signal b1) || wait b1 }
```

4.3 A Verified, Executable Implementation of DICE Protection Environment (DPE)

A measured boot protocol computes the measurements (e.g., cryptographic hashes) of firmware and software on a device, as it boots up, and records them for subsequent verification. DICE, a Trusted Computing Group standard, specifies a layered architecture for measured boot. In DICE, each firmware layer L_n, before it transfers the control to the next layer L_{n+1}, computes a secret called Compound Device Identifier (CDI) C_{n+1} derived from C_n and the measurement of L_{n+1} (the lowest firmware layer uses a hardware-based secret as the root of trust). Layer L_n may also derive a public-private key-pair for L_{n+1} (using C_{n+1}) and issue an X.509 certificate. DICE★ (Tao et al. 2021) is a verified implementation of DICE in F★.

DPE is an evolution of the DICE standard, aimed at confining the DICE secrets to a separate component, as opposed to being directly accessible to the firmware layers. DPE specifies an API that a client may use to implement DICE; the API is designed in a way that only public information (public keys, certificates) is exchanged with the client. DPE supports multiple sessions, each running its own DICE protocol instance, and concurrent operations on different sessions, to scenarios such as when a device has multiple chips that each have separate certified boot sequences using DICE.

We implement DPE in USSLang, focusing on a basic profile that provides a function call interface, plaintext sessions, no session migration, and support for message signing and X.509 certificates (no sealing). We specify the DPE API in USSLang, supporting multiple concurrent sessions, and prove that (a) each session follows the DICE protocol state machine (the firmware layers boot up in order), and (b) concurrent operations on different sessions do not interfere with each other. Using the USSLang extraction pipeline, we extract this implementation to Rust. Finally, using Rust Foreign Function Interface (FFI), we link our code with the (extracted) C libraries for DICE★ (for verified X.509 certificates serialization) and EverCrypt (for verified cryptographic primitives) (Protzenko et al. 2020), to provide an end-to-end implementation of DPE published as a Rust crate.

The USSLang implementation of DPE relies on several USSLang libraries. We build a verified linear probing hash table to map sessions to their DICE states in the DPE. The hash table is protected with a mutex—we build a mutex library modeled on (and extracted to) Rust mutexes. A DPE function on a session acquires the mutex, reads session state from the table, updates the table with a special `InUse` state, and releases the mutex. This allows for concurrent operations on different sessions. At the end of the function, the session table is updated with the new state. We specify the DICE state machine correctness for individual sessions using a ghost reference to a pointwise map PCM (one entry per session), where each entry in the map itself is a monotonic trace PCM with fractional permissions. The trace captures the valid state machine transitions, while fractional permissions allow for sharing half the permission for individual sessions with the client. Each DPE API is specified using this ghost reference and ensures that every session follows the DICE state machine. The extracted DPE implementation (excluding libraries such as the hash table, which we also extract) consists of 1575 lines of formatted Rust code. The USSLang specification of the state machine and the DPE API consists of 276 lines of code, while the implementation contains 215 lines of computational code and 442 lines of lemmas, ghost functions, and proof hints.

5 RELATED WORK & CONCLUSIONS

We have discussed related work throughout the paper, but cover the relation to other logics here. The closest related work to ours is SteelCore (Swamy et al. 2020). Like USSL, it provides a shallow embedding of CSL in F★, but, as mentioned earlier, its formalization relies on an effectful semantics of monotonic state developed by Ahman et al. (2018) and axiomatized in F★. This axiomatization provides a predicate `witnessed (p:mem → prop) : prop`, where one can introduce `witnessed p` by proving that `p` holds for the current memory and remains true as the memory evolves. An effectful operation recall eliminates witnessed p, allowing the caller to assume that p is true in the current memory. Unfortunately, this construction can be broken by using parametricity-breaking classical axioms, such as indefinite description, which is commonly used in ghost code. The authors of SteelCore propose and implement various fixes to the SteelCore logic to curtail the impact of this problem, though their fixes are not backed by a formal proof of soundness. Further, the fixes introduce various restrictions on the use of ghost code, and requires treating invariant tokens as non-erasable values, despite their having no actual runtime significance. These issues are described at length online at https://github.com/FStarLang/FStar/issues/2814.

USSL avoids these problems by providing a foundational model without relying on effectful axioms. Though we have applied USSL to F* programs with the effect of divergence, the logic itself stands on its own—one could also use it as the basis of a logic for total correctness, if that were desired. There are two other foundational differences with SteelCore. First, although SteelCore also provides a notion of ghost computation, this is also axiomatic, whereas in USSL, ghost computations are derived from F*'s notions of erasure. Second, SteelCore's representation of computation trees are more complex than the Ret–Act–Par definition of USSL shown in §3.7. It turns out that SteelCore's definition fails to be properly universe-polymorphic and cannot be used as the basis for representing computations in Steel (Fromherz et al. 2021), a surface language for proving in SteelCore, the analog of USSLang to USSL.

SteelCore offers a different specification style than USSL. The core proposition in Steel is called vprop, a "proposition with an associated value", that allows treating a proposition like a heap fragment from which values can be projected. The authors of Steel show that vprops can make certain kinds of specifications easier and more amenable to SMT-based automation, similar to implicit dynamic frames (Smans et al. 2012), however, this comes at the loss of extensionality for vprop, i.e., equivalent vprops are not necessarily equal. In contrast, USSL provides a more canonical model of separation logic propositions, just affine PCM-predicates, validating the extensionality principle. USSL is also more expressive, with its stratification enabling storable propositions and nested invariants, features lacking in SteelCore.

We have already compared our work to Iris, HTT, and FCSL. iCAP (Svendsen and Birkedal 2014) is an impredicative separation logic, and an ancestor of Iris. Its model is also based on interpreting propositions in a custom category with step-indexing and, like Iris' model, it does not appear amenable to a shallow embedding within dependent types. Preceding iCAP, HOCAP (Svendsen et al. 2013) provides a higher-order separation logic, that like USSL is predicative, but uses step-indexing for nested Hoare triples and guarded recursive predicates. Instead of nested Hoare triples, USSL, being dependently typed, supports abstracting over functions with pre- and post-conditions expressed in types. Guarded recursive predicates would be useful to investigate in the future for USSL.

Koronkevich and Bowman (2024) study structuring heaps using multiple universes, storing total heap-reading functions at higher universe levels than the fragments of the heaps that they read. However, their heap is acyclic, unlike ours where cycles within a universe level are allowed, and they do not provide a program logic.

Conclusions. USSL is a foundational concurrent separation logic with a direct semantics in dependent types. By stratifying the heap into universes levels, it provides many features of modern separation logics, including PCM-based higher-order ghost state and dynamic invariant allocation, with the limitation that it is predicative. We have used the logic to build a growing collection of verified libraries, and even a real-world security critical application, giving us some confidence that the logic is expressive and usable, and suggesting that we have made progress towards developing high-assurance, concurrent systems code within dependently typed languages like F*.

REFERENCES

M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1999.

D. Ahman, C. Fournet, C. Hriţcu, K. Maillard, A. Rastogi, and N. Swamy. Recalling a witness: Foundations and applications of monotonic state. *PACMPL*, 2(POPL):65:1–65:30, 2018.

A. W. Appel. Verified software toolchain. In A. Goodloe and S. Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. 2012.

A. Arasu, T. Ramananandro, A. Rastogi, N. Swamy, A. Fromherz, K. Hietala, B. Parno, and R. Ramamurthy. FastVer2: A provably correct monitor for concurrent, key-value stores. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2023.

J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis*. 2003.

A. Charguéraud. Characteristic formulae for the verification of imperative programs. *ICFP*. 2011.

L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. *TACAS*. 2008.

M. Dodds, S. Jagannathan, M. J. Parkinson, K. Svendsen, and L. Birkedal. Verifying custom synchronization constructs using higher-order separation logic. *ACM Trans. Program. Lang. Syst.*, 38(2), 2016.

A. Fromherz, A. Rastogi, N. Swamy, S. Gibson, G. Martínez, D. Merigoux, and T. Ramananandro. Steel: Proof-oriented programming in a dependently typed concurrent separation logic. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2021.

S. Gruetter, V. Fukala, and A. Chlipala. Live verification in an interactive proof assistant. *PACMPL*, 8(PLDI), 2024.

A. Hobor, R. Dockins, and A. W. Appel. A theory of indirection via approximation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2010.

B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2011.

J. B. Jensen and L. Birkedal. Fictional separation logic. *ESOP*. 2012.

R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016.

R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

P. Koronkevich and W. J. Bowman. Type universes as allocation effects. https://arxiv.org/abs/2407.06473, 2024.

A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, 2008.

A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In Z. Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 2014.

A. Nanevski, A. Banerjee, G. A. Delbianco, and I. Fábregas. Specifying concurrent programs in separation logic: morphisms and simulations. *PACMPL*, 3(OOPSLA):161:1–161:30, 2019.

J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1), 2012.

K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In Z. Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 2014.

K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.

N. Swamy, A. Rastogi, A. Fromherz, D. Merigoux, D. Ahman, and G. Martínez. SteelCore: An extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.*, 4(ICFP), 2020.

Z. Tao, A. Rastogi, N. Gupta, K. Vaswani, and A. V. Thakur. DICE*: A formally verified implementation of DICE measured boot. In *30th USENIX Security Symposium (USENIX Security 21)*. 2021.

Trusted Computing Group. DICE. https://trustedcomputinggroup.org/work-groups/dice-architectures/.

Trusted Computing Group. DICE protection environment. https://trustedcomputinggroup.org/wp-content/uploads/TCG-DICE-Protection-Environment-Specification_14february2023-1.pdf, 2023.

L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL), 2019.