PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs

GABRIEL EBNER, Microsoft Research, USA
GUIDO MARTÍNEZ, Microsoft Research, USA
ASEEM RASTOGI, Microsoft Research, India
THIBAULT DARDINIER*, ETH Zurich, Switzerland
MEGAN FRISELLA*, University of Washington, USA
TAHINA RAMANANANDRO, Microsoft Research, USA
NIKHIL SWAMY, Microsoft Research, USA

PulseCore is a new program logic suitable for intrinsic proofs of higher-order, stateful, concurrent, dependently typed programs. It provides many of the features of a modern, concurrent separation logic, including dynamically allocated impredicative invariants, higher-order ghost state, step-indexing with later credits, and support for user-defined ghost state constructions. PulseCore is developed foundationally within the F* programming language with fully mechanized proofs, and is applicable to F* programs itself.

To evaluate our work, we use Pulse, a surface language within F* for PulseCore, to develop a range of program proofs. Illustrating its suitability for proving higher-order concurrent programs, we present a verified library for task pools in the style of OCaml5, together with some verified task-parallel programs. Next, we present various data structures and synchronization primitives, including a barrier that requires the use of higher-order ghost state. Finally, we present a verified implementation of the DICE Protection Environment, an industry standard secure boot protocol. Taken together, our evaluation consists of more than 31,000 lines of verified code in a range of settings, providing evidence that PulseCore is both highly expressive as well as practical for a variety of program proof applications.

CCS Concepts: • **Theory of computation** → **Separation logic**; *Program verification.*

Additional Key Words and Phrases: dependent types, program proof, concurrency, separation logic

ACM Reference Format:
Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *Proc. ACM Program. Lang.* 9, PLDI, Article 208 (June 2025), 24 pages. https://doi.org/10.1145/3729311

1 Introduction

We seek to expand the scope of dependently typed programming, traditionally used for provably correct functional programs, to concurrent programs, including those that use shared memory. We envision integrating concurrent separation logic (O'Hearn 2004; Reynolds 2002) (CSL) within a dependently typed programming language, enabling specifications and proofs in an *intrinsic*

*Work done while at Microsoft Research, USA

Authors' Contact Information: Gabriel Ebner, Microsoft Research, Redmond, USA, gabrielebner@microsoft.com; Guido Martínez, Microsoft Research, Redmond, USA, guimartinez@microsoft.com; Aseem Rastogi, Microsoft Research, Bengaluru, India, aseemr@microsoft.com; Thibault Dardinier, ETH Zurich, Zurich, Switzerland, thibault.dardinier@inf.ethz.ch; Megan Frisella, University of Washington, Seattle, USA, mfris@cs.washington.edu; Tahina Ramananandro, Microsoft Research, Redmond, USA, taramana@microsoft.com; Nikhil Swamy, Microsoft Research, Redmond, USA, nswamy@microsoft.com.

ACM 2475-1421/2025/6-ART208
https://doi.org/10.1145/3729311

style, i.e., the type of a program directly expresses its correctness specification. We aim for an approach that is powerful enough to express both the higher-order concurrency libraries provided in high-level languages, as well as the low-level concurrency primitives used to implement those libraries and in systems programming. For example, consider the following API to a task pool, similar to what is provided in OCaml5, but with richer specifications:

```
fn setup_pool (nworkers:pos) requires emp returns p : pool ensures pool_alive 1.0 p
ghost fn share_pool (p:pool) requires pool_alive (f +. g) p ensures pool_alive f p ** pool_alive g p
ghost fn gather_pool (p:pool) requires pool_alive f p ** pool_alive g p ensures pool_alive (f +. g) p
fn spawn (p:pool) (t:task pre post) requires pool_alive f p ** pre returns h : handle
                                            ensures pool_alive f p ** joinable p post h
fn await (p:pool) (h:handle) requires pool_alive f p ** joinable p post h ensures pool_alive f p ** post
```

We develop (a more complete version of) this API in §5.1; for now, we point out just some high-level features. First, notice that the specification of each function is captured intrinsically by its type, rather than with a separate specification language or assertion on the side. Second, the specifications use separation logic, with setup_pool creating a new pool p and providing full ownership to the pool (pool_alive 1.0 p) to its caller, and share_pool enabling sharing ownership of the pool among multiple threads—we write ** for separating conjunction. Next, the language is higher-order, allowing spawning tasks t (closures with arbitrary pre- and post-conditions) and waiting (using await) for a task to complete and yield its postcondition. Finally, specifications are precise and modular, enabling functional correctness proofs of clients.

A logic expressive enough for such specifications and proofs must support a few important features. First, it must allow dynamically allocating *impredicative* invariants, i.e., it should be possible to promote any predicate to an invariant, including an invariant itself. For our task pool, this is necessary to allow tasks spawned in a pool to depend on that pool's pool_alive predicate, allowing a task to recursively spawn further tasks. Second, the logic must support ghost-state constructions to express both spatial and temporal sharing. We see a simple form of this in the pool_alive predicate, indexed by a fraction to distinguish full versus partial ownership. Internally to the pool, more advanced forms of ghost state constructions are necessary, including monotonic state and higher-order ghost state. Finally, it must be possible to embed the logic within dependent types and use it for intrinsic specifications and proofs.

In terms of expressive power, one would naturally consider Iris (Jung et al. 2018), a state-of-the-art CSL formalized in Coq, supporting impredicative invariants and higher-order ghost state. These features involve an inherent circularity between heaps and heap predicates, which Iris resolves by step-indexing (Appel and McAllester 2001). Iris' treatment of step-indexing involves interpreting predicates in a custom category of COFEs and solving a system of recursive domain equations to find a fixed point. Adapting this to other settings requires considerable expertise and effort in mechanized domain theory (Sieczkowski et al. 2015). Further, rather than being used to reason intrinsically about Coq programs, one usually uses Iris to reason extrinsically about other languages deeply embedded in Coq.

In principle, any approach to solving recursive domain equations could be used to support impredicative invariants and higher-order ghost state. For instance, Birkedal et al.'s (2011) ultrametric spaces could also be used as a foundation. However, with thanks to reviewers of a previous version of this paper, we turn instead to *indirection theory* (Hobor et al. 2010), an approach that provides a way to construct a step-indexed model directly within dependent types, without resorting to the use of advanced categorical techniques. Predicates in indirection theory are just predicates in the host theory (Hobor et al. use Coq), though with a step-indexed notion of equivalence.

Indirection theory has for long been used as a foundation for VST (Appel 2012), though it lacked support for features such as higher-order ghost state and impredicative invariants, instead only providing dynamically allocated locks. Recently, the foundation of VST has been revamped to use Iris instead (Mansky 2022; Mansky and Du 2024), motivated, at least in part, to avail of the more advanced forms of ghost state and invariants that Iris provides. Our work shows that with some careful design, indirection theory itself can support impredicative invariants and the most commonly used forms of higher-order ghost state.

PulseCore: A CSL embedded in F* using indirection theory. Our work begins by revisiting indirection theory and providing a new, more compact model of it formalized in F* (Swamy et al. 2016). Next, using indirection theory we develop PulseCore, a new CSL with dynamically allocated impredicative invariants, higher-order ghost state, and user-defined ghost state constructions based on partial commutative monoids. We then show how to apply PulseCore to intrinsically type and reason about monadic F* programs itself, distinguishing total correctness for atomic computations and ghost code, from partial correctness for general-purpose potentially non-terminating programs.

Working with intrinsically typed programs instrumented with ghost code provides a different perspective on step-indexing. In particular, rather than tying the reduction relation to the step index, we can explicitly reduce the step index in ghost code. By adapting *later credits* in the style of Spies et al. (2022) to our setting, we obtain a convenient way to work with step-indexed predicates.

To evaluate PulseCore, we use Pulse, a surface language in F* for developing programs specified and proven using PulseCore, implemented as an F* compiler plugin. Pulse programs can make use of various features of F*, including refinement types, higher-order unification, typeclasses, SMT, and tactic-based proof automation, illustrating the benefits of CSL embedded within a full-fledged dependently typed language. However, we do not describe Pulse in detail in this paper, and do not claim it as a contribution, though we use it for all our examples.

We have used Pulse to build verified libraries and programs, including data structures like arrays, hash tables, singly & doubly linked lists, trees, and various synchronization devices. We have also developed libraries for various ghost state constructions, including higher-order ghost state and monotonic ghost state. Our libraries also include notions similar to Iris' view shifts and fancy updates, which we call *shifts* and *trades*. Our flagship case study is an OCaml5-style task pool together with a task-parallel Quicksort on mutable arrays. The proofs use *pledges*, a form of *trades* which enable modular reasoning about properties that will hold after a future event. Additionally, we present a barrier in the style of Jung et al. (2016) using higher-order ghost state; and a verified implementation of the DICE Protection Environment (DPE) (Trusted Computing Group 2023), an industry-standard cryptographic measured boot protocol, extracted to either C or Rust.

In summary, we claim the following contributions:

(1) PulseCore, a CSL with impredicative invariants and user-defined higher-order ghost state, relying on indirection theory, for which we provide a new, simpler model. (§3)
(2) A definitional interpreter for stateful programs, with ghost actions, fork/join concurrency, and a partial-correctness program logic by intrinsic typing in PulseCore. (§4)
(3) Step-indexing with later credits for intrinsically typed programs, explicitly reducing the step index in ghost code, and the ability to buy credits in partial correctness contexts. (§3,4)
(4) Several verified libraries and applications, including a task pool and task-parallel Quicksort using the new *pledge* connective; a barrier; and the DPE boot protocol. (§5)

In summary, our work reinterprets the key ideas and goals of Iris on an alternative, existing foundation (i.e., indirection theory), meeting different design goals, notably intrinsic proofs of dependently typed programs. PulseCore is fully formalized in about 15,000 lines of F*, and our evaluation has an additional 31,000 lines of verified Pulse code, all available in our supplement.

2 PulseCore, by Example

We present PulseCore using Pulse 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 expect that a reader familiar with Iris will find the reasoning principles provided by PulseCore familiar, though we presume no background in Iris and other CSLs. We also present background on F* as we go.

2.1 An Interface for a Spin Lock

Consider the following interface to 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              val protects (l:lock) (p:slprop) : slprop
```

The listing above shows an abstract type lock and an abstract predicate protects l p, associating a lock l with a p:slprop. The type slprop is the type of separation logic propositions interpreted as affine heap predicates. The trivial proposition, emp, is valid in all heaps, while p ** q, the separating conjunction,[1] is 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 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 handle both these features.

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

The signature of create p states that the caller gives up ownership of p:slprop to create a new lock l:lock and gains ownership instead of a predicate protects l p.

Next, the function dup duplicates protects l p, allowing ownership of a lock to be shared among threads. The return type of dup is just unit: Pulse lets us omit it, instead of having to write **returns** _:unit. The **ghost** keyword indicates that dup l is a *ghost function*, which 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. As such, ghost functions have no observable effect on the program's execution, and are erased when a program is compiled.

Finally, we have functions to acquire and release locks: acquire l blocks until the lock becomes available and then grants the caller ownership of p; while release l gives up ownership of p to the lock. Unlike ghost functions, regular functions like acquire may loop forever—PulseCore is, in general, a program logic of *partial correctness*.

2.2 Implementing a Lock with Invariants

Intuitively, a lock is a mutable reference to a machine word, set to 0 when the lock is free, and 1 when it is held. A thread acquiring a lock repeatedly attempts to atomically compare-and-set the reference from 0 to 1. We associate an *invariant* with a lock to say that when the lock is free, the lock owns the slprop that it protects.

Invariants are program properties that are valid before and after every step of computation. In PulseCore, invariant predicates are given names. In particular inv i p asserts that p:slprop is an

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

invariant named i, where the type of i is iname. Invariant names are only needed for specification and proof purposes and, like ghost computations, they are erased when a program is compiled.

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

In the listing above, a lock is 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 r is 0, the lock owns p; otherwise it owns nothing—the proposition r ↦ v asserts that the reference r *points-to* a heap cell containing the value v, and ∃* is an existential quantifier for slprop. The protects predicate states that the invariant name l.i is associated with lock_inv l.r p.

With our representation chosen, Figure 1 shows the implementation of the lock API. Program proofs like this are implemented interactively in Pulse, where the user queries a VS Code-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). Pulse 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.

Creating a lock with new_invariant. The ghost operation new_invariant p, a primitive in PulseCore, dynamically allocates an invariant, requiring the caller to give up ownership of p, and gaining instead an invariant inv i p. We'll soon see how we can temporarily regain ownership of p for the duration of an atomic step.

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

Using new_invariant, the code in create is relatively straightforward: we allocate a new reference cell r. Then allocate an invariant i, taking ownership of the predicate p, since the initial value of r is 0ul. Finally, we pack the reference and invariant name into a record and return it.

Duplicating a lock. 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, a PulseCore primitive shown below. This is important since it allows invariants to be shared among multiple threads. Since protects l p is essentially just an invariant, the implementation of dup is easy.

```
ghost fn dup_invariant (i:iname) (p:slprop) requires inv i p ensures inv i p ** inv i p
```

Once an invariant inv i p is allocated, PulseCore enforces that p is valid before and after every step of computation in all threads throughout the remainder of the program. For this purpose, PulseCore distinguishes a class of *atomic* computations, computations that involve at most one physically atomic memory operation. The memory operation can be preceded or followed by an arbitrary number of ghost operations (since these are erased) or pure operations (since these are not observable to other threads). We rely on F*'s effect system to distinguish pure, ghost, and memory operations. For example, most platforms support several forms of atomic operations, e.g., atomic reads, writes, or compare-and-swap (CAS) instructions. The Pulse library only exposes those atomic operations that are actually atomic on the target platform, even if more operations are atomic in the model. Atomic functions in PulseCore are defined with the **atomic** keyword:[2]

```
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)
```

[2]The relation == denotes the propositional (Leibniz) equality in F*.

```
fn create p requires p
returns l:lock ensures protects l p {
  let r = alloc 0ul;
  let i = new_invariant (lock_inv r p);
  {r;i}
}

fn release l
requires p ** protects l p
ensures protects l p {
 later_credit_buy 1;
 with_invariants l.i
 { later_elim _;
  drop (maybe _ _);
  l.r := 0ul;
  later_intro (lock_inv l.r p)
 }}
```

```
ghost fn dup l p requires protects l p
ensures protects l p ** protects l p
{  dup_invariant l.i (lock_inv l.r p); }

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

```
(* pre ** inv i p *) with_invariants i
{ (*  pre ** ▷p *) ghost_or_pure(); at_most_one_atomic_op(); ghost_or_pure(); (* post ** ▷p *) }
(* post ** inv i p *)
```

```
ghost fn later_intro p requires p ensures ▷p          fn later_credit_buy (n: nat) requires emp ensures £n
ghost fn later_elim p requires £1 ** ▷p ensures p      val later_credit_add a b : Lemma (£(a+b)==£a ** £b)
```

Fig. 1. Implementing the lock API, along with a sketch of the rules for opening invariants and later credits

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 (pure p lifts pure propositions to slprop); otherwise, r is unchanged.

The key reasoning rule about invariants is encapsulated in PulseCore's with_invariants combinator. It allows the caller to *open* an invariant for the period of a single atomic step. Atomic computations can be preceded or followed by any number of ghost steps—the resulting computation is still considered atomic. While an invariant is opened the caller gets temporary access to the resource under the invariant, but this resource has to be returned at the end of the atomic step. The computation inside the with_invariants may only open invariants whose names are distinct from i; Pulse automatically infers and checks this side condition in this example.

The proof rule that with_invariants provides is sketched in Figure 1. Notice that if one initially owns pre ** inv i p, then within the with_invariants block, one gains pre ** ▷p—the predicate ▷p guards p under a *later* modality. This is analogous to the behavior of invariant opening in logics like Iris and, just as in Iris, it is unsound for with_invariants to give access to the resource p directly, rather than ▷p (Jung et al. 2018).

The later modality can be freely introduced using later_intro, a ghost function whose signature is shown in Figure 1. The later modality satisfies various useful algebraic properties, e.g., ▷(p ** q) == ▷p ** ▷q, and for a class of *timeless* predicates p:slprop { timeless p }, ▷p == p. However, in the general case, eliminating ▷p to p, comes at a cost, accounted for using later credits (Spies et al. 2022). In particular, later_elim consumes a single later credit, £1, to eliminate ▷p to p. Later credits can be bought in any quantity using later_credit_buy, but only in non-atomic and non-ghost contexts. Quite often later credits can be bought right before they are needed, and they can also be encapsulated behind other abstract predicates for later use like any other resource, so they often do not show up in function signatures. Finally, later_credit_add, a lemma in F*, proves that later credits can be split and combined additively.

Implementing acquire. We show a tail-recursive acquire at the right of Figure 1. At line 8, we buy a later credit, before opening the invariant l.i. The specification of the with_invariants block states that it returns a boolean retry, which if true, indicates that the cas failed, and the lock was not acquired, and otherwise, the lock was acquired and the resource p is available. Additionally, unconditionally, we restore the invariant ▷(lock_inv l.r p) before exiting the block. We have ▷(lock_inv l.r p) at line 13 and can eliminate it to lock_inv l.r p using the credit we just bought. Next, 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 PulseCore are only proven partially correct—they are allowed to loop indefinitely.

Implementing release. The implementation of release is simpler, and follows the same pattern as acquire: we buy a credit before opening an invariant, and spend it immediately to eliminate the later modality, before using an atomic write to clear the flag and return the resource 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 PulseCore is an *affine* separation logic. Our artifact includes a better implementation of locks that forbids releasing a lock without acquiring it first.

We hope this conveys a flavor of PulseCore, focusing primarily on its support for dynamically allocated invariants, later modality, and credits, as well as atomic, ghost, and potentially non-terminating computations. We show our formal model for PulseCore next, supporting these and many other features, summarizing the key ideas of our development below.

2.3 Key Ideas of the Formal Development

We represent each thread in a program as a tree of atomic or ghost actions, where the formal, dynamic semantics is a state-passing definitional interpreter. The main goal of the development is to find a representation for the state, a type called mem, such that all the actions can be given a model that justifies their typing.

The type mem is represented as product of (1) a *timeless heap*, itself a product of the concrete heap of a program and a heap containing all its non-higher-order ghost state; (2) a *higher-order ghost store*, in which one can store mem predicates, including certain predicates marked as invariants; and (3) the *saved credits*, to model later credits.

To represent the higher-order ghost store, we need to tie a knot, which is enabled by indirection theory and step indexing. A key invariant of mem is that a predicate p stored in a memory m and marked as invariant is valid in m *after* a (possibly fictional) time step, i.e., ▷p is valid in m. Following Hobor et al., we call the steps evolving the memory over time "aging," and we prove that a memory with n saved credits can always be aged at least n times.

With this representation of mem, all the actions (notably: allocating, reading, and writing both concrete references and references to first- and higher-order ghost state; allocating and opening an invariant; buying and eliminating credits) can be derived and given their desired separation logic specifications. With actions grounded in this semantic model, the rest of the PulseCore system is a collection of verified libraries.

3 The PulseCore Separation 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. 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 2.5 minutes on a modern laptop using 12 threads, though we expect the proofs to become more compact over time.

As in many standard models of separation logic (Jensen and Birkedal 2012), heaps are modeled using partial commutative monoids (PCM), as defined by the pcm a type below from F⋆'s library.

```
type pcm a = { (.?): symrel a;    (⊙): (x:a → y:a{x .? y} → a);    one:a;    laws: assoc_comm_unit (⊙) one; }
let affine (#p:pcm a) (f: (a → prop)) = ∀ x y. f x ∧ x .? y ⟹ f (x ⊙ y)
```

Note, in the definition of affine the (#p:pcm a) notation marks p as an implicit argument. More generally, for any partial commutative semigroup (which may not necessarily have a unit) we write x .? y for the composability predicate, and x ⊙ y for the composition, and leave the specific partial semigroup implicit. As a broad outline of this section, we first define a core heap and a pcm core, which models the concrete mutable state of a program. Next, we define a ghost heap, as an erased core, where the erased type is an existing F⋆ notion to mark a type as computationally irrelevant. We define a *timeless* heap as a pair core & erased core. Finally, we use indirection theory to define a knot representing the full PulseCore heap: we define slprop as affine predicates on this full heap. The full heap combines a timeless heap, with accounting for later credits, and a *step-indexed* heap in which one can allocate invariants or allocate references containing slprop values for higher-order ghost state. The separation logic defined in this manner is independent of the semantics of programs. In §4, we apply this separation logic for intrinsically typed proofs of F⋆ programs, though we expect that one could use our core logic in other styles, e.g., to build an extrinsic logic for a deeply embedded language, as is typical in, say, Iris.

3.1 The Timeless Heap: Product of Concrete and Ghost Heaps

The core type is a partial map from abstract heap addresses to cells, and a counter for allocation:

```
type cell = | Cell : a:Type → p:pcm a → v:a → cell
type core = { cell: (nat → option cell); ctr: nat { ∀ i. i ≥ ctr ⟹ cell i == None } }
```

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, PulseCore allows each abstract memory address to model an allocation unit, where the value stored is a product or sum, chosen from some appropriate PCM to model a struct or union.

A PCM on core is 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 the values are composable in the cell's PCM. In other words, two cores h_0, h_1 are composable 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. Composing cores is the pointwise composition of their cells, and the maximum of their counters.

Ghost heap: erased core. A ghost heap is defined as erased core, where we rely on F⋆'s existing model of erasure. Briefly, 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. F⋆ provides an abstract, monadic type erased t, with a constructor 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 () and the addition will not be evaluated at runtime. Rather than working with erased values in an explicitly monadic style, F*'s effect system tracks when a computation may depend on a ghost value, tainting it with a **Ghost** effect, and enforces that pure computations are not tainted. In particular, the function reveal: erased t → **Ghost** t is tainted with a ghost effect, and is the inverse of hide. This means that no pure computation can depend on the result v:t of a ghost computation, 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 }.

With that preface, defining the ghost heap as erased core and lifting pcm core to pcm (erased core) is straightforward, since a and erased a are isomorphic. We define timeless_heap = core & erased core and a PCM on timeless heaps is just the product of the PCMs of its components, i.e.,

$$(c_1, g_1)\ .?\ (c_2, g_2) \Longleftrightarrow c_1\ .?\ c_2 \land g_1\ .?\ g_2 \qquad (c_1, g_1) \odot (c_2, g_2) == (c_1 \odot c_2, g_1 \odot g_2)$$

A tlprop is defined as an affine predicate p: (timeless_heap → prop) { affine p }. The trivial proposition emp and the separating conjunction (p ** q) are defined on tlprop in a standard way:

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

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

```
let ref (#t:Type u#a) (p:pcm t) = nat          let ( ⪯ ) (x y:a) = ∃ frame. x .? frame ∧ x ⊙ frame == y
let pts_to #t (#p:pcm t) (r:ref t p) (x:t) : tlprop = λ h → match (fst h).cell r with
  | Some (Cell t' p' y) → t == t' ∧ p == p' ∧ x ⪯ y  | _ → False
```

A reference r:ref p is just an abstract memory address nat, 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 r x and F* can infer the implicit arguments #t and #p.

If the same address points to two values p and q of a PCM separately, then they are composable: (pts_to x p ** pts x q) h ⟹ p .? q, equivalently: pts_to x p ** pts x q == pts_to x p ** pts x q ** pure (p .? q). And for any two composable values p .? q we have pts_to x p ** pts_to x q == pts_to x (p ⊙ q).

Ghost references ghost_ref t and the associated points-to predicate ghost_pts_to x p are defined analogously, just on the second (ghost) heap. As separation logic propositions, (concrete) references and ghost references are interchangeable. Only once we give them a computational meaning in §4.1 will there be a difference between the two.

3.2 Knot Construction using Indirection Theory

To support stored memory predicates (mem → prop), we would like to pair a timeless heap with a higher-order ghost store (hogs), ideally writing the type below (omitting refinements to affine predicates and credits for brevity):

```
type mem = { tlh: timeless_heap; hogs: nat → erased (option (mem → prop)); }
```

However, such a definition is not sound since mem occurs negatively in the hogs field, and inductively defined types in dependent type theories like F* must be strictly positive.

Instead, following Hobor et al. (2010), we will construct an approximation to the type above using indirection theory, such that the equation mem = f (mem → prop) is approximately true (in a sense made precise below) for the functor f p = timeless_heap & (nat → erased (option p)). Every heap will have a level and store predicates that range over heaps of a lower level. This level decreases as the program executes, although not in lockstep with physical machine instructions. In fact, later_elim is the only primitive in PulseCore that decreases the level: it is a ghost function compiled to a noop.

We present a simplified version of the knot construction of Hobor et al. (2010). The construction is parameterized by a functor f mapping heap predicates to heaps. That is, given an abstract type heap_pred of heap predicates, f heap_pred is the type of heaps storing heap predicates in heap_pred. That f is a functor means that there is an fmap: (a→b) → f a → f b operation which commutes with identity (fmap id == id) and composition (fmap (a ∘ b) == fmap a ∘ fmap b). From this functor f we construct a "knot" type k f such that k f ≈ f (k f → prop) in the following sense:

```
val unpack : k f → f (k f → prop)        val pack : erased nat → f (k f → prop) → k f
val level : k f → erased nat             let approx n = λ g x → if level x ≥ n then False else g x
val pack_unpack x : Lemma (pack (level x) (unpack x) == x)
val unpack_pack n x : Lemma (level (pack n x) == n ∧ unpack (pack n x) == fmap (approx n) x)
```

We can define this type k f as a pair of a level and the auxiliary type k' f n, which represents heaps of level n; the function k' f is clearly well-defined by recursion on n:

```
let k' f n = f (m:nat{m<n} → k' m → prop)          let k f = n:erased nat & k' f n
```

We can then define the functions that allow us to unpack and pack this type.[3] The predicates stored in k' n are only defined for heaps of level less than n. We extend the predicates to heaps of higher levels by setting them to false, giving us unpack, where (| a, b |) constructs a dependent pair:

```
let level (| n, x |) = n       let unpack (| n, x |) = fmap (λ g (|m,y|) → if m ≥ n then False else g m y) x
```

The pack function on the other hand simply restricts the predicates to heaps of level less than n; ignoring their behavior for heaps of higher levels:

```
let pack n x = (| n, fmap (λ g m y → g (|m,y|)) x |)
```

The two functions are not inverses of each other since unpack (pack n x) approximates the stored predicates by setting them to false for heaps of levels higher than n. The lemmas pack_unpack and unpack_pack follow easily from functorial properties and function extensionality.

Our formalization is slightly different than the Coq code presented by Hobor et al. (2010): we recursively define the type of heaps instead of the type of predicates; and our preliminary predicate type is a function type ranging over heaps of lower levels while their predicate type is an n-tuple of predicates for each level. Their nested predicates require a recursive floor operation and associated inductive proofs while our function type can be easily restricted to a smaller range. We also make use of F★'s extensional type theory and subtyping: (nat → pred) is a subtype of (m:nat { m < n } → pred) and the function restriction above is thus completely implicit; folding and unfolding the recursive type definitions is transparent as well due to extensionality. As a result, our formalization of the knot type only takes 76 lines of code.

Using only the knot interface, we make two important definitions: *aging* is the process that evolves the heap over time and decreases the level. We define age_1 h = pack (max 0 (level h − 1)) (unpack h), which reduces the heap level by one if possible, or returns the heap unchanged if the level is already zero. A heap predicate is *hereditary* if it is preserved under aging: hereditary p = ∀ h. p h ⟹ p (age_1 h).

3.3 Heap and Propositions

For PulseCore, we use the following functor, where hogs is short for higher-order ghost store:

```
type hogs_val p = | None : hogs_val p | Inv : p → hogs_val p | Pred : p → hogs_val p
type hogs_func p = { tlh: timeless_heap; credits: erased nat; hogs: nat → erased (hogs_val p); }
```

[3]These functions are called squash and unsquash by Hobor et al. (2010), however the name squash is already used by the F★ library so we picked different names.

That is, a heap consists of the timeless heap, the number of saved later credits, and the higher order ghost store which maps addresses to either a stored invariant, a stored predicate, or nothing. The fmap function is defined in the obvious way by applying the function to the values stored in hogs_val showing that hogs_func is a functor.

PulseCore only implements two primitive kinds of higher order ghost state: invariants and predicates. In our experience, these two primitives are sufficient to express every kind of higher order ghost state we have encountered (albeit with a bit of encoding). In principle, we could also add other kinds of higher order ghost state to the heap, so long as they can be described as a covariant functor on predicates similar to hogs_val. For example, we could also allow sequences of predicates nat → p to be stored. We could even make the construction parametric in the supported higher-order ghost state cells and their associated functors, but we decided against the API complexity of extra parameters. Iris and its domain equation solver are parametric in this way, and also support functors that are contravariant in the predicate type.

The knot construction then gives us the type of *prememories* premem = k hogs_func. (In a moment we will refine prememories to the subtype of memories by requiring extra conditions that cannot be expressed in the functor.) First, we give prememories a natural partial commutative semigroup structure by setting:

```
let (.?) (x y: premem) = level x == level y ∧ (unpack x).tlh .? (unpack y).tlh ∧
 (∀ a. let v = (unpack x).hogs a in v == None ∨ v == (unpack y).hogs a)
let (⊙) (x y: premem) = pack (level x) {
 tlh = (unpack x).tlh ⊙ (unpack y).tlh;
 credits = (unpack x).credits + (unpack y).credits;
 hogs = (λ a → let v = (unpack x).hogs a in if v == None then (unpack y).hogs a else v); }
```

Notably, this composition (⊙) does not have a unit since prememories only compose with other prememories of the same level. We refine prememories to obtain the final type of *memories* and *separation logic propositions* by requiring that all predicates are hereditary and affine and that only finitely many cells are allocated:[4]

```
let is_affine (p:premem → prop) = ∀ m₀ m₁. (p m₀ ∧ m₀ .? m₁) ⟹ p (m₀ ⊙ m₁)
let slprop = p:(premem → prop) { is_affine p ∧ hereditary p }
let mem = m:premem { all_preds_are_slprops m ∧ (∃ c. ∀ i. i > c ⟹ (unpack m).hogs i == None) }
```

The separating conjunction and the empty proposition are defined in the same way as we did for timeless heaps. They form a commutative monoid over slprop.

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

We can lift affine predicates and define duplicable predicates, existential quantification:

```
let lift p m = p (unpack m).tlh   let duplicable p = p == p ** p   let (∃*) #t (p: t→slprop) m = ∃ x. p x m
```

We use standard definitions of later credits, the later modality, and timeless predicates. Note that later is vacuously true on heaps of level zero.

```
let £n m = (unpack m).credits ≥ n                let ▷p m = level m > 0 ⟹ p (age₁ m)
let timeless p = ∀ m. level m > 0 ⟹ ▷p m ⟺ p m
```

The definitions above satisfy a form of Löb induction: $(\forall\ m.\ \triangleright p\ m \implies p\ m) \implies (\forall\ m.\ p\ m)$. However, PulseCore does not expose this induction principle and we have not yet found a need for it in

[4]While $F^\star$ is an extensional type theory (i.e., the conversion typing rule requires only provable equality of the types and not definitional equality), $F^\star$ does not satisfy function extensionality for all functions (i.e., $\forall\ h.\ (p\ h \iff q\ h) \implies p == q$). For this reason, slprop is actually defined as a refinement to those functions for which function extensionality holds.

Pulse. Common uses of Löb induction in Iris are subsumed by F*'s native support for recursive definitions: fixed point combinators for non-terminating programs are represented using F*'s divergence effect, and recursively defined predicates can be written using well-defined recursion.

Later credits clearly satisfy £0 == emp and £(a+b) == £a ** £b. Our definition of later has an interesting corner case if the memory has level zero: in that case we have ▷p m $\iff$ p m. For this reason some equalities between slprops will only be valid for heaps of nonzero level. This is not a major restriction: the actions we will define in §4.1 have as a precondition that the level of the input memory is strictly greater than the number of saved credits (and so, in particular, nonzero).

Propositions that do not mention the higher-order ghost state, namely emp, pure p, £n, lift p (and in particular pts_to x y and ghost_pts_to x y) are all timeless. The later modality commutes as expected with the separation logic connectives: ▷(p ** q) == ▷p ** ▷q and ▷(∃* x. p x) == ∃* x. ▷(p x). As a consequence, p ** q is timeless if both p and q are timeless. The Pulse library contains all the usual separation logic connectives including ∀* and iterated star, except for the magic wand (we use trades instead which encapsulate ghost functions, see §4.1).

Predicates stored in the higher-order ghost state are not stored losslessly. They are approximated to the current level. (Note that approx n p is only equivalent to p for heaps of a level strictly less than n.) Following Hobor et al. (2010) we define an equivalence relation to express that two slprops are approximately equal for heaps of the current (and any lower) level:

```
let eq_at n p q = approx n p == approx n q        let (≡) p q : slprop = λ m → eq_at (level m + 1) p q
```

The main feature of this equivalence is that in the presence of p ≡ q, we can rewrite p to q using the equality (p ≡ q) ** p == (p ≡ q) ** q. It is also duplicable, commutative ((p ≡ q) == (q ≡ p)), transitive ((p ≡ q) ** (q ≡ r) == (p ≡ q) ** (p ≡ r)), and reflexive (i.e., true in every memory: ∀ m. (p ≡ p) m). It also commutes with later: ▷(p ≡ q) == ▷p ≡ ▷q. And it satisfies congruence laws such as for **: (q ≡ r) == (q ≡ r) ** ((p ** q) ≡ (p ** r)). For timeless predicates, ≡ is equivalent to ==.

Finally, we define the higher-order ghost state predicates for invariants and stored predicates:

```
let iref = erased nat              let inv (i: iref) (p: slprop) : slprop =
 λ m → match (unpack m).hogs i with | Inv p' → eq_at (level m) p p' | _ → False
let slprop_ref = erased nat        let slprop_pts_to (i: slprop_ref) (p: slprop) : slprop =
 λ m → match (unpack m).hogs i with | Pred p' → eq_at (level m) p p' | _ → False
```

Both are defined identically (except for the names). Their only difference is how they are interpreted by actions. The slprop_pts_to predicate is duplicable, and all predicates a reference points to are equivalent: level m > 0 ∧ (slprop_pts_to i p ** slprop_pts_to i q) m $\Longrightarrow$ ▷(p ≡ q) m. This implication is restricted to heaps of nonzero level. The reason for that is simple: if a heap h satisfies the left-hand side, then we know that eq_at (level h) p q. But the right-hand side says ▷(p ≡ q) h which unfolds to eq_at (level (age$_1$ h) + 1) p q. To show that level (age$_1$ h) + 1 == level h, the level of h needs to be nonzero (since age$_1$ is the identity otherwise). Due to the level requirement we do not expose this rule in Pulse as an equality, but as a ghost function (which require nonzero level as input).

Invariants obey the same laws as stored predicate references, mutatis mutandis. However they are interpreted differently by the program logic: any invariant resource that is not opened remains in the ownership of the interpreter, and actions need to preserve the unopened invariants as frames. Therefore we define an slprop for the iterated separating conjunction ** of all unopened invariants (each guarded by later, because otherwise mem_owns e m could never be true of m itself):

```
val mem_owns (e: set iref) (m: mem) : slprop
let read_inv (m: mem) (i: iref { ∃ p. inv i p m }) = let Inv p' = (unpack m).hogs i in p'
val mem_owns_equiv e m (i: iref { (∃ p. inv i p m) ∧ (i ∉ e) }) :
  Lemma (mem_owns e m == mem_owns (e ∪ i) m ** ▷(read_inv i m))
```

4 PulseCore for Intrinsic Proofs of Dependently Typed Programs

To derive a program logic, we define the basic building blocks of PulseCore programs—atomic and ghost actions. We then derive implementations of combinators as actions, such as new_invariant, with_invariant, and later_elim etc. A program is defined as a forest of infinitely branching trees of actions, and we prove that an interpreter of action trees that non-deterministically interleaves actions from each thread satisfies the expected Hoare triples. The interpreter is parameterized with a fuel:nat that counts the number of actions it can reduce—we prove that so long as the interpreter is not out of fuel, programs can safely buy and spend later credits.

4.1 Actions

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 }
```

Standard Hoare-style rules apply 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 PulseCore, frame-preserving, state-passing functions, whose state is a memory mem.[5] The signature below describes a total function, which for any frame is an st computation over mem, with result type a; the flag ghost is true is if the action only modifies the ghost heap, opens is a set of invariant names that may be opened by the computation.

```
let inames_ok i m = ∀ n ∈ i. ∃ p. inv n p m        let budget m : erased int = level m − credits m − 1
let action (ghost:bool) (a:Type u#a) (opens:set iref) (pre:slprop) (post: a → slprop)
= except:set iref { except ∩ opens = ∅ } → frame:slprop → st mem a
  (requires λ m₀ → inames_ok except m₀ ∧ (pre ** frame ** mem_owns except m₀) m₀ ∧
    (if ghost then budget m₀ ≥ 0 else budget m₀ ≥ 1))
  (ensures λ m₀ x m₁ → inames_ok except m₁ ∧ (post x ** frame ** mem_owns except m₁) m₁ ∧
    (if ghost then eq_ghost m₀ m₁ ∧ budget m₀ == budget m₁ else budget m₀ − budget m₁ ≤ 1))
```

The budget of a memory as defined above measures how often we can call buy on it. Atomic actions can decrease the budget by one, but the buy action is the only one that actually decreases it. The other actions keep the budget constant. Atomic and ghost actions have different preconditions for the budget: this is because we want the composition of an atomic and a ghost action to be another atomic action. Therefore we need to be able to apply a ghost action to the result of an atomic action. The precondition also enforces that level m > 0—note the −1 in the definition of budget.

The eq_ghost relation below models a necessary condition for ghost actions: that they do not modify the concrete heap. This relation only needs to be a preorder for the rest of the development to work out, though commonly, it would also be an equivalence relation. PulseCore uses erasure to delineate ghost steps from concrete programs. It ensures that ghost steps have no observable effects and can indeed be erased. To this end we require that the result of a ghost action is noninformative:

```
let eq_ghost : preorder mem = λ m₀ m₁ → fst (unpack m₀).tlh == fst (unpack m₁).tlh
let upd_ghost (m₀: mem) (m₁: erased mem { eq_ghost m₀ m₁ }) : m:mem { m == reveal m₁ } =
  pack (level m₀) { m₁ with tlh = (fst (unpack m₀).tlh, hide (snd (unpack m₁).tlh)) }
```

An erased ghost action with a non-informative result is itself non-informative:

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

[5]The actual definition additionally restricts the PCM values in the full memory to a subset specified in the PCM structure. Such refined PCMs can represent non-trivial gadgets for spatial and temporal sharing (Arasu et al. 2023).

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 eq_ghost relation to apply the upd_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. We define the following abbreviations, noting that most actions open no invariants, so we usually omit the is : set iref argument, rather than writing $\emptyset$ explicitly:

```
let stt_ghost t is p q = erased (action true t is p q)          let stt_atomic t is p q = action false t is p q
ghost fn f (x : t) requires p returns y:t ensures q opens is ≜ val f (x : t) : stt_ghost t is p (λ y → q)
atomic fn f (x : t) requires p returns y:t ensures q opens is ≜ val f (x : t) : stt_atomic t is p (λ y → q)
```

Later credits and elimination. The later modality comes with two ghost actions, later_intro and later_elim shown in Figure 1, as well as an atomic action buy1. The later_intro action does not modify the heap at all, it simply reflects the implication p m $\Longrightarrow$ ▷p m, while later_elim decreases the level of the heap and decreases the credits to ensure that budget m remains constant. The **atomic fn** buy1 () **requires** emp **ensures** £1 action simply increments the credits counter, thereby decreasing the budget by one, which satisfies the postcondition as buy1 is an atomic action.

Predicate references. We define two actions for slprop_pts_to. The allocation action extends the higher-order ghost store by a new cell and stores the given slprop there. The gather action makes use of the fact that (slprop_pts_to i p ** slprop_pts_to i q) m $\Longrightarrow$ ▷(p ≡ q) m, which only holds if level m > 0 and hence needs to be exposed as an action instead of an equality.

```
ghost fn slprop_ref_alloc (p: slprop) requires emp returns x:slprop_ref ensures slprop_pts_to x p
ghost fn slprop_ref_gather x p q requires slprop_pts_to x p ** slprop_pts_to x q ensures ▷(p ≡ q)
```

Invariants. Allocating an invariant consumes the resource p of the invariant, this is because the actions needs to maintain the mem_owns property, which contains p after the invariant is allocated. Secondly, the action takes an additional ctx argument and ensures the freshness property i ∉ ctx. The gather_invariant action is similar to the gather action on predicate references.

```
ghost fn fresh_invariant p ctx requires p returns i:iref {i ∉ ctx} ensures inv i p
ghost fn gather_invariant i p q requires inv i p ** inv i q ensures ▷(p ≡ q)
```

Finally we get the with_invariant combinator, which takes an action preserving ▷p and returns an action preserving inv i p. The combinator's implementation gets an except set disjoint from o, and in particular i ∉ except. This enables the use of the mem_owns_equiv lemma to split off the ▷(read_inv i m) resource from mem_owns, which is equivalent to ▷p on m, and we can call the nested action with except ∪ i. Also note that the combinator is ghostness-parametric: with_invariants on a ghost action returns a ghost action, on an atomic action it returns an atomic action.[6]

```
val with_invariant (g: bool) (a: Type) (p q: slprop) (r: a→slprop) (o: set iref) (i:iref { i ∉ o }) :
  action g a o (▷p ** q) (λ x → ▷p ** r x) → action g a (o ∪ i) (inv i p ** q) (λ x → inv i p ** r x)
```

Points-to. For the pts_to resource we define three actions, to allocate, write, and read the reference. Allocation increments the freshness counter in the concrete part of the timeless heap and puts the specified PCM value in the next empty cell:

```
val extend #a (p: pcm a) (x:a) : act (ref p) emp (λ r → pts_to r x)
```

[6]Our type of with_invariant requires invariants to be opened and closed in a stack discipline, simplifying the handling of invariant names. Iris offers a lower-level interface, enabling them to be opened and closed in any order. We have yet to find need for this, though it is conceivable that we would need to add such a feature in the future.

For writes, we need to show that the update is frame-preserving, that is, we can appropriately write the cell in the full memory which may be framed. Note that the existing value is provided as an erased argument, the new value is provided implicitly in the frame-preserving write function.

```
let fp_upd (p:pcm a) (x y:a) = v:a{ x ⪯ v } → u:a{ y ⪯ u ∧ (∀ (f:a{x .? f}). y .? f ∧ (x ⊙ f == v ⟹ y ⊙ f == u))}
val write (r:ref p) (x y: erased a) (f: fp_upd p x y) : action false unit (pts_to r x) (λ _ → pts_to r y)
```

In addition to the reference, the read action takes an additional argument that allows us to relate the returned value, which is the full in-memory value, and the argument in the pts_to relation, which is the share that we own.

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

Note, although read and write are atomic actions from the perspective of the logic, when verifying programs for execution on a specific machine, one may choose to only expose certain operations as atomic. The ghost_pts_to resource has analogous actions, except that they are all ghost. We use typeclasses in F* to overload $r \mapsto x$ instead of writing pts_to, ghost_pts_to, slprop_pts_to, etc.

Shifts, Trades, & Pledges. Shifts, trades, and pledges are three related binary separation logic connectives in Pulse using actions. A shift is analogous to a view shift in Iris, and is the existence of a ghost action that can transform a resource p to q, while opening the invariants is and having access to a duplicable resource e. We omit the invariant names when they are empty.

```
let shift (is: set iref) (p q: slprop) : slprop = ∃* (e: slprop) (f: stt_ghost unit is (e ** p) q) (d: duplicable e). e
```

Shifts are duplicable and come with the obvious introduction and elimination rules, transitivity, and monotonicity in the set of invariants the ghost computation is allowed to open. The elimination rule makes essential use of noninformativeness of ghost computations: shift merely requires the *existence* of the ghost function f. In order to run this ghost function, we need to apply indefinite description which taints the whole resulting heap as erased. The only way to recover the concrete heap is via the upd_ghost function.

A trade holds a resource and, unlike shifts, is not duplicable—it is similar to a fancy update in Iris, though slightly different, since it is only indexed by the set of invariants that it may open. Like shift, it is transitive and monotonic in the set of invariants.

```
let trade (is: set iref) (p q: slprop) : slprop = ∃* (e: slprop) (f: stt_ghost unit is (e ** p) q). e
```

Pledges are trades which preserve the hypothesis—they satisfy the following properties:[7]

```
let pledge is p q = trade is p (p ** q)
ghost fn return_pledge p q requires q ensures pledge ∅ p q
ghost fn squash_pledge is p q requires pledge is p (pledge is p q) ensures pledge is p q
ghost fn join_pledge is p q r requires pledge is p q ** pledge is p r ensures pledge is p (q ** r)
ghost fn map_pledge is p q r requires pledge is p q ** shift q r ensures pledge is p r
ghost fn split_pledge is p q r requires pledge is p (q ** r) ** £2 returns i: iname
  ensures pledge (is ∪ i) p q ** pledge (is ∪ i) p r ** pure (i ∉ is)
```

The split_pledge function allocates ghost state and a new invariant to synchronize between the two returned pledges, ensuring that the input pledge is p (q ** r) is only eliminated once. Both returned pledges need to open this new invariant, which is why we need £2 as a precondition. Those two later credits are stored in the resources of the returned pledges, respectively.

[7]The actual definition has an additional technical side condition that all invariants in is are allocated. This is necessary to ensure that we can always allocate a fresh invariant which is not in is.

4.2 Representing & Running Concurrent Computations

The semantics of PulseCore are given by a fueled, definitional interpreter. Starting from a fuel *n*, the interpreter runs the first *n* actions of the computation and then loops. The fuel is a lower bound for the budget, ensuring that we never run out of budget and that the actions in the computation can be executed. Computations are represented as trees of type 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 begins with the action f and then continues with k. Par m k represents m and k executing concurrently.

```
type stt : a:Type u#a → slprop → (a → slprop) → Type =
| Ret: x:a → stt a (p x) p          | Act: f:act o p q → k:(x:b → Dv (stt a (q x) r)) → stt a p r
| Par: m:stt (raise_t unit) p₀ (λ _ → emp) → k:stt a p₁ q → stt 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 PulseCore, enabling programming functions such as acquire to loop indefinitely until the lock becomes available.[8] We only have one kind of action node, Act, since, via ghost_act_non_info, an erased ghost action can be promoted to a ghost action and then subsumed to an action. In Par m k, the sub-tree m returns a unit value 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 can model a fork and, using the barrier described in §5.3, a join construct.

We can finally implement the function later_credit_buy n from Figure 1 as an stt value by simply calling the buy1 action n times.

To interpret computation trees, we work in an extension of the st monad that provides an additional action to consume 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) and a fuel parameter, 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, looping if the fuel is exhausted.

```
val run (t:tape) (f:stt a pre post) (m₀:mem { (pre ** mem_owns ∅ m₀) m₀ }) (f: nat { budget m₀ ≥ f })
: Dv (x:a & m₁:mem { (post x ** mem_owns ∅ m₁) m₁ })
```

The definition of run is part of our trusted computing base, since it defines the semantics of programs. Although its type guarantees that if it terminates, it ensures the verified postcondition of a program, one should still review its implementation to confirm that it does not gratuitously loop always. That said, its implementation, including the proof that it is well-typed, is only 61 lines long, and the code 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. The main loop decreases the fuel after each action; it diverges if the fuel reaches 0—it could also return the partially reduced program.

To use the run function we need to specify a fuel f and an initial memory. The initial memory consists of the empty timeless heap, zero saved credits, and a higher-order ghost store of a level f+1 (which has a budget of f+1).

[8] It is also possible to require k to be a total function, yielding a total correctness semantics for, say, 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).

```
let run_top (t:tape) (f:stt a emp post) (f: nat) : Dv (x:a & m:mem { (post x ** mem_owns ∅ m) m }) =
  run t f (empty_mem_with_level (f+1)) f
```

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

This native compilation of Pulse is part of our trusted computing base. However, to justify this step, we point out, first, that the observable behavior of stt programs does not depend on the choice of the fuel and, second, that we can choose the fuel high enough for the program to run for any given number of actions. For the first point, we note that the fuel is never passed to the actions and the level of the higher-order ghost store is erased. A metatheoretic parametricity property of F* guarantees that any concrete value in the non-erased heap does not depend on erased values. For the second point, note that we have defined the interpreter to execute n actions when given the initial fuel n.

5 Evaluating PulseCore

We have used Pulse to build a variety of verified libraries and applications, as summarized in the table alongside, representing about 31,000 lines of code and proof. Our code includes basic libraries for references, arrays, and a model of heap and stack allocation; derived combinators such as trades and pledges; libraries of PCM constructions; data structures, hash tables, linked lists, a double-ended queue, trees, and utilities on arrays; and various concurrency-related libraries, including mutexes, barriers, and a task pool. We also report on an implementation of DPE, a low-level, cryptographic measured boot protocol service with support for concurrent sessions.

In this section, we describe the task pool, which makes essential use of impredicativity and ghost state; a task-parallel Quicksort, showing that our specification of the task pool is precise and can be used to verify clients, while also highlighting the use of pledges and a barrier that uses higher-order ghost state; and DPE, illustrating that Pulse can also be used to build verified systems applications.

References, arrays	5,671
Trades, pledges	1,665
Other combinators	4,724
PCM	1,739
Data Structures	5,039
Task pool	1,375
DPE	3,074
Other examples	8,504
Total (LOC)	31,791

Fig. 2. Lines of Pulse code

5.1 A Task Pool

We revisit the task pool from §1 and describe its main invariants, rather than the full implementation, which takes about 1,400 lines of code. Our API also supports tearing down a pool:

```
fn teardown_pool (p:pool) requires pool_alive 1.0 p ensures pool_done p
```

Our API is inspired by OCaml5's Domainslib, but there is one discrepancy: our teardown_pool blocks until the pool becomes empty, and only then deallocates it. In Domainslib, teardown_pool is meant to be called only when it is known that the pool is already empty, its behavior being undefined otherwise. An alternative would have been an interface where teardown_pool requires callers to prove the pool is empty, e.g., by adding a counter to the pool to track the emptiness, and making client modules track it. Arguably, our version of teardown_pool is safer and easier to use. We also point out that our library is not designed for efficiency, rather to show that the API can be implemented and verified. Nevertheless, our code can be extracted to OCaml and runs correctly. It would be interesting to consider an optimized implementation in the future.

The main idea of the proof is for the pool to maintain a list of tasks, and for the state of each task to be represented by a ghost state machine. When the task is ready to be run, the pool holds its precondition; when a task is complete, the pool holds its postcondition; and after a task has been joined, the postcondition is no longer held by the pool, having been given back to the caller, but the invariant records that the task has already been claimed.

We represent a pool as the following structure, where runnable is a mutable list of tasks, lk is a lock, and g_runnable is a ghost list of tasks, a reference to ghost state constructed using the PCM of monotonic state to describes ownership over a monotonically growing list of tasks.

```
type pool = { runnable:ref (list task_t); lk:lock; g_runnable:monotonic_ref (list task_t) (⪯) }
```

The main pool_alive predicate is shown below, where lock_alive is a fractional version of the protects predicate for spin locks shown in §2. The predicate lock_inv correlates the concrete and ghost state, and all_state_pred v is the main invariant of the task list. We describe it in detail shortly, though we note that all_state_pred encapsulates pre- and postconditions of a spawned task, including pool_alive itself, i.e., the lock contains permissions that may refer to itself, an essential use of impredicativity.

```
let lock_inv r gr = ∃* v. r ↦ v ** gr ↦ v ** all_state_pred v
let pool_alive (f : perm) (p:pool) = lock_alive p.lk (f /. 2.0) (lock_inv p.runnable p.g_runnable)
```

Next, we represent a task handle as a record with a reference to the current state of a task, together which a ghost state that encodes the state machine. This state machine is developed using the PCM of fractional, anchored preorders proposed by Arasu et al. (2023), who use this PCM in a proof of correctness of a concurrent monitor for key-value stores. The PCM works by starting with a model of monotonic state defined using preorders, i.e., a reference to mutable ghost state s can be associated with a preorder ⪯, such that threads can hold partial knowledge of s and independently advance the state according to the preorder ⪯. The main unusual element is the notion of an *anchor*, a relation on states that limits the extent to which threads can independently advance the ghost state. For example, with a monotonically increasing counter, one could define an anchor that allows a thread to increment the counter from an odd to an even number even with partial ownership, while increments from even to odd numbers require full ownership.

In our setting, we use this PCM to define a state machine for tasks with four states Ready, Running, Done, and Claimed, with the preorder p_st and anchor relation anchor_rel enforcing that state updates proceed in this order, and further that a thread can update a task's state from Done to Claimed only with full knowledge of the state. The type anchored_ref task_state p_st anchor_rel is the type of a reference to ghost state in the fractional, anchored preorder PCM.

```
type task_state = Ready | Running | Done | Claimed
type handle = { state:ref task_state;  g_state : anchored_ref task_state p_st anchor_rel }
```

The state_owns predicate describes resources owned by the pool depending on the task's state:

```
let state_owns pre post h st = match st with
| Ready → pre | Running → emp | Done → post | Claimed → anchored h.g_state Claimed
```

Initially, the pool holds the task's precondition; while the task is running, it holds nothing; when the task is done, it holds the postcondition; and when the task is claimed, the pool holds knowledge that the task is claimed—the anchored assertion, from Arasu et al., ensures that every other thread also knows that the task is claimed, so it cannot be claimed again.

Additionally, for a running task, the pool owns only a fraction of the task's state; the state and the ghost state are always correlated, except that the concrete state is never set to Claimed.

```
let state_pred pre post h = ∃* st. h.state $\overset{frac(st)}{\mapsto}$ unclaimed st ** h.g_state ↦ st ** state_owns pre post h st
let frac s = match s with | Running → 0.5 | _ → 1.0        let unclaimed = function Claimed → Done | x → x
```

A task_t packages a thunk:dyn, with two slprop_refs to store its pre- and postcondition, and the task handle. The type dyn is from F⋆'s library, and provides a way to promote v:t to d:dyn{dyn_has_ty d t}, and then downcast it back to the original type. The model of dyn in F⋆'s FStar.Dyn module relies on the fact that the type unit → **Dv** a is in universe 0, no matter the universe of a.[9]

```
type task_t = { pre : slprop_ref; post : slprop_ref; h : handle; thunk : dyn; }
```

We can then define task_typing, a predicate that expresses the typing of a task, relating the contents of the slprop_refs to the type of the thunk, using higher-order ghost state.

```
let task pre post = unit → stt unit pre (λ _ → post)
let task_typing t = ∃* pre post. t.pre ↦ pre ** t.post ↦ post ** pure (dyn_has_ty t.thunk (task pre post))
```

The main invariant all_state_pred is just an iterated separating conjunction over the tasks:

```
let all_state_pred tasks = fold (λ out t → out ** task_typing t ** state_pred t.pre t.post t.h) emp tasks
```

Finally, we build towards the definition of the joinable predicate, starting with task_spotted p t a duplicable predicate which states that the task t is in the pool p. Since the ghost state is monotonic, the predicate snapshot p.g_runnable v once proven, remains true.

```
let task_spotted p t = ∃* v. snapshot p.g_runnable v ** pure (List.mem t v)
```

Next, we show the definition of handle_spotted, which states that the task associated with a handle is the pool, and that the postcondition q of the handle, using a later credit, can be traded for the postcondition of the task, i.e., the two are equivalent after a step.

```
let handle_spotted p q h = ∃* t. task_spotted p t ** pure (t.h == h) ** trade (∃* p. t.post ↦ p ** p ** £1) q
```

Finally, the joinable predicate says that the task is anchored to Ready (so it has not been claimed yet), that the handle is in the pool, and stashes a later credit so that the trade in handle_spotted can be eliminated eventually.

```
let joinable p post h = anchored h.g_state Ready ** handle_spotted p post h ** £1
```

This is a complex invariant, but the pieces fit together well. For instance, in the implementation of await, we start with pool_alive f p and joinable p post h. We acquire the lock and from joinable p post h, the handle h is in the pool and its state is not Claimed. If its state is not Done, await takes work from the pool, if any task is ready, before retrying. When the handle state is Done, from state_owns, the pool holds the task's postcondition. Using the later credit in joinable, we can eliminate the trade in handle_spotted to get post. Finally, we advance the state of the handle to Claimed and anchor it there, so no other thread can try to claim it again, release the lock, and return.

5.2 A Task Parallel Quicksort

Next, we show how to use our task pool API to implement a provably correct Quicksort on mutable arrays. We show only the concrete steps in our implementation; including all the ghost steps needed for the proof adds less than 100 lines of Pulse code. The code uses await_pool and spawn_, from our extended task pool interface described below.

[9]A good intuition for the type unit → **Dv** a is to think of it as the set of lambda terms t such that t () either diverges or reduces to a valid runtime representation of type a (i.e., where types, ghost values, etc. are erased). In particular, the map a → (unit → **Dv** a) is not an injection in general. Thus unit → **Dv** a is inhabited by only countably many lambda terms and its cardinality is small enough to fit in universe 0. Further, functions with **Dv** effect cannot be eliminated within F⋆'s logical fragment, e.g., given f and g of type unit → **Dv** a, there is no way to even state that f() ≠ g(), let alone prove it. Such an interface could be axiomatized in languages without the **Dv** effect.

```
fn rec qsort (p:pool) (a:array int) (lo hi:nat) {
  if (hi − lo > 1) { let j = partition a lo hi; spawn_ p (λ () → qsort p a lo j); qsort p a (j+1) len; }}
fn quicksort (a:array int) (len:nat) { let p = setup_pool n; qsort p a 0 len; await_pool p; teardown_pool p }
```

The algorithm begins by allocating a task pool p; then calls qsort, which partitions the array a with the pivot at j; then spawns a task to sort the left partition and recursively call qsort to sort the right partition. Rather than wait for the spawned task to finish, using await, we just wait once at the end for all tasks to be done, using await_pool. As such, when qsort returns, the array is not yet sorted; qsort only guarantees that the array will be sorted *at some point in the future*, when all the spawned tasks are done. The pledge connective works well for this style of reasoning, and our API allows turning joinable into a pledge, using the disown ghost function shown below. Combining disown with spawn gives spawn_.

```
ghost fn disown p post h requires joinable p post h ensures pledge (pool_done p) post
fn spawn_ p f requires pool_alive f p ** pre ensures pool_alive f p ** pledge (pool_done p) post
```

The specification of qsort is shown below:

```
fn qsort p a lo hi requires pool_alive f p ** pts_to_range a lo hi s0 ** pure_pre a lo hi lb rb s0
ensures pledge (pool_done p) (pool_alive f p ** ∃* s. (pts_to_range a lo hi s ** pure_post a lo hi lb rb s0 s))
```

The pts_to_range predicate expresses ownership of a range of an array, and pure_pre and pure_post are functional specifications proving that s is a sort of the initial s0. The interesting part is that the postcondition appears beneath a pledge. This means that the top-level quicksort function cannot call teardown_pool directly, since it does not regain full permission to the pool. Instead, it first calls await_pool, which waits until all the tasks in the pool are done, and then redeems the pledge, without tearing it down. Finally, having regained full permission to the pool, we can tear it down.

```
fn await_pool p requires pool_alive f p ** pledge (pool_done p) post ensures pool_alive f p ** post
```

The omitted proof steps include a join_pledge after the recursive call in qsort to combine the pledged postconditions of the sub-tasks; sharing and gathering permissions to the pool; and splitting and recombining ownership of disjoint sub-ranges of the array. The main functional correctness part relating pure_pre to pure_post reuses an existing proof a purely functional Quicksort in F*.

5.3 A Barrier with Higher-Order Ghost State

Jung et al. (2016) present a proof of a barrier in Iris that makes essential use of higher-order ghost state. We implement a barrier in Pulse (in about 400 lines of code) providing a very similar signature, shown below, starting with an abstract type b of barriers and two abstract predicates send b p and recv b p. We also have a function i that projects the invariant name associated with a barrier.

```
val b : Type     val send (_:b) (_:slprop) : slprop     val recv (_:b) (_:slprop) : slprop     val i (_:b) : iname
```

One can allocate a barrier for any predicate p and obtain send b p and recv b p to be given to threads that wish to communicate over the barrier. The signaling thread gives up ownership of p to the barrier, and the receiving thread blocks until the barrier is signaled and then gains p.

```
fn create (p:slprop) requires emp returns b:b ensures send b p ** recv b p
fn signal b requires send c p ** p ensures emp        fn wait b requires recv b p ensures p
```

An interesting part is that recv b (p ** q) can be split into a recv b p and a recv b q, allowing multiple threads to wait on the barrier, so long as they only extract disjoint resources from it. The ghost operation split opens the barrier invariant, updates an internal table of higher-order ghost references to record the split, and then returns the split permission. Doing this requires two later credits.

```
ghost fn split b requires recv b (p ** q) ** £2 ensures recv b p ** recv b q opens [ i b ]
```

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

As a final example, we highlight a verified, low-level, systems application, DPE a measured boot protocol, which computes cryptographic hashes of firmware and software on a device, as it boots up, and records them for subsequent verification. DPE is designed for scenarios such as when a device has multiple chips that each have separate certified boot sequences, where each boot sequence is served in a concurrent session handled by a DPE server, typically running on dedicated hardware. DPE specifies an API that exchanges public information (public keys, certificates) with a client, while derived secrets remain confined to the DPE server itself, and each session is governed by a state machine described by the DICE (Trusted Computing Group [n. d.]) standard.

Our implementation of DPE provides a basic profile with a function call interface, plaintext sessions, message signing and X.509 certificates, but no sealing or session migration. The API supports multiple concurrent sessions, and proves 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. Internally, session state is maintained in a verified linear probing hash table, protected with a variation of our spin lock library modeled on (and extracted to) Rust mutexes. A typical DPE API call 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 per-session monotonic trace PCM with fractional permissions. The overall proof is relatively straightforward and technically less complex than some of our other examples, though it does demonstrate Pulse at work on a real-world systems application.

Using the Pulse extraction pipeline, we extract this implementation to Rust, and use Rust's foreign function interface to link our code with the (extracted) C libraries for verified serialization of X.509 certificates (from DICE* (Tao et al. 2021)) and cryptographic primitives (from EverCrypt (Protzenko et al. 2020)), to provide an end-to-end implementation of DPE as a Rust crate. Note, not all Pulse programs can be extracted to Rust and pass the borrow-checker; after all, Pulse is more expressive than Rust. However, our DPE implementation is carefully written in a subset of Pulse that does correctly borrow-check when extracted to Rust. In total, our DPE consists of about 3,000 lines of Pulse code, extracting to 1,955 lines of formatted Rust code. Alternatively, we also extracted it to 2,315 lines of C code using Karamel (Protzenko et al. 2017).

6 Related Work & Conclusions

PulseCore is inspired by Iris (Jung et al. 2018), and aims to provide a logic with similar reasoning principles. As explained in §1, the main point of distinction is that PulseCore's model is given directly in terms of the propositions of the host language, making it possible to apply PulseCore to intrinsically typed host language programs—F* in our case. However, as in Iris, PulseCore's separation logic stands independently of programs, and one could use it in other styles, e.g., to derive program logics for deeply embedded DSLs. In terms of expressive power, we believe PulseCore and Iris are comparable, though Iris and its many extensions are likely still more expressive than PulseCore. With thanks to the reviewers of this paper for this explanation, we point out that a cross-cutting difference is that in PulseCore, we start by giving a model of heaps with a knot via indirection theory for storing heap predicates; in contrast, in Iris, the iProp type itself is the solution of a recursive domain equation over an arbitrary set of functors. As pointed out in §3.3, one could extend PulseCore to also support storing values taken from any covariant functor of a heap predicate, though we have yet to do so.

We have also yet to attempt several proof styles that are common in Iris. Notably, Iris can be used for relational refinement proofs and logical relations, and we have not attempted any such proofs in PulseCore. We conjecture that PulseCore could be used in conjunction with relational Hoare logics, such as those developed by Maillard et al. (2020), or that one could develop step-indexed logical relations in PulseCore for deeply embedded languages, rather than for F* itself. Iris also has support for logically atomic triples (LATs). While logical atomicity is also expressible in PulseCore, we have used a ghost-function-passing style similar to Jacobs and Piessens (2011), rather than via structured LATs. Extensions to Iris, notably Transfinite Iris (Spies et al. 2021) enable proofs of termination. Although PulseCore's sub-language of ghost functions is terminating, it is, in general, a partial-correctness logic. Our representation of programs as indexed action trees is similar in spirit to Frumin et al.'s (2024) use of interaction trees (Xia et al. 2019), although Frumin et al. reason extrinsically about programs in Iris and a "guarded" type theory, rather than intrinsically dependently typed programs.

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. The authors of iCAP present a proof of an event loop, which has some similarities to our task pool, inasmuch as it also relies on an impredicative invariant and allows recursively registering handlers. However, their specifications express safety, rather than also enabling correctness proofs of clients.

Cosmo (Mével et al. 2020) is a separation logic for Multicore OCaml developed in Iris, providing both a low-level logic to reason with respect to its weak memory model, and a high-level logic for reasoning about data-race free programs. It would be interesting in the future to connect our model of task pools to Cosmo, and to use it as a basis for a foundational model of Domainslib.

PulseCore's is built using indirection theory (Hobor et al. 2010). In comparison to the original formulation of CSL in indirection theory as used by VST (Appel 2012), PulseCore provides invariants rather than dynamic locks. VST is also designed for use with C programs, and supports lower-level reasoning about stack pointers and byte-addressable memory. Although we think PulseCore could, in principle, be applied to a model of C programs as in VST, in practice we apply it to reason about dependently typed, higher order programs.

SteelCore (Swamy et al. 2020) is a CSL shallowly embedded in F*, with support for dynamically allocated invariants, but without higher-order ghost state or impredicativity. SteelCore's model relies on a non-standard axiom for monotonic state (Ahman et al. 2018) which, is unsound when combined with certain commonly used classical axioms. SteelCore also lacks a foundational model of ghost state. SteelCore's representation of computation trees are more complex than the Ret–Act–Par definition of PulseCore shown in §4.2, and is not fully universe-polymorphic.

Other separation logics embedded in dependently typed languages include Hoare Type Theory & FCSL (Nanevski et al. 2019, 2014, 2008), and CFML (Charguéraud 2011). However, they lack support for dynamically allocated invariants and higher-order ghost state, e.g., one cannot program a dynamically allocated mutex in HTT or FCSL, while CFML focuses on sequential programs.

Conclusions. We have presented PulseCore, an expressive CSL with a fully mechanized theory, and applicable to higher-order, dependently typed, concurrent programs. While prior work (Bhargavan et al. 2017) has been successful in using F* to build and deploy verified industrial software, it has been limited primarily to sequential programs. We are optimistic that our work now opens the door to the development of high-assurance concurrent programs using dependently typed languages like F*.

Data-Availability Statement. This paper describes the core logic used in Pulse as of April 2025: https://github.com/FStarLang/pulse An executable artifact is also available (Ebner et al. 2025).

References

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. doi:10.1145/292540.292555

Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a Witness: Foundations and Applications of Monotonic State. *PACMPL* 2, POPL (jan 2018), 65:1–65:30. https://arxiv.org/abs/1707.02466

Andrew W. Appel. 2012. Verified Software Toolchain. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7226)*, Alwyn Goodloe and Suzette Person (Eds.). Springer, 2. doi:10.1007/978-3-642-28891-3_2

Andrew W. Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683. doi:10.1145/504709.504712

Arvind Arasu, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Aymeric Fromherz, Kesha Hietala, Bryan Parno, and Ravi Ramamurthy. 2023. FastVer2: A Provably Correct Monitor for Concurrent, Key-Value Stores. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Boston, MA, USA) *(CPP 2023)*. Association for Computing Machinery, New York, NY, USA, 30–46. doi:10.1145/3573105.3575687

Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Cătălin Hriţcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages*. http://drops.dagstuhl.de/opus/volltexte/2017/7119/pdf/LIPIcs-SNAPL-2017-1.pdf

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed kripke models over recursive worlds. *SIGPLAN Not.* 46, 1 (Jan. 2011), 119–132. doi:10.1145/1925844.1926401

Arthur Charguéraud. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. ACM, New York, NY, USA, 418–430. doi:10.1145/2034773.2034828

Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs—PLDI 2025 Artifact. doi:10.5281/zenodo.15047546

Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.* 8, POPL, Article 12 (Jan. 2024), 30 pages. doi:10.1145/3632854

Samuel Gruetter, Viktor Fukala, and Adam Chlipala. 2024. Live Verification in an Interactive Proof Assistant. *PACMPL* 8, PLDI (2024). doi:10.1145/3656439

Aquinas Hobor, Robert Dockins, and Andrew W. Appel. 2010. A theory of indirection via approximation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 171–184. doi:10.1145/1706299.1706322

Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 271–282. doi:10.1145/1926385.1926417

Jonas Braband Jensen and Lars Birkedal. 2012. Fictional Separation Logic. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 377–396. doi:10.1007/978-3-642-28869-2_19

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 256–269. doi:10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

Kenji Maillard, Catalin Hritcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 relational program logics. *Proc. ACM Program. Lang.* 4, POPL (2020), 4:1–4:33. doi:10.1145/3371072

William Mansky. 2022. Bringing Iris into the Verified Software Toolchain. arXiv:2207.06574 [cs.PL] https://arxiv.org/abs/2207.06574

William Mansky and Ke Du. 2024. An Iris Instance for Verifying CompCert C Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 6 (Jan. 2024), 27 pages. doi:10.1145/3632848

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: a concurrent separation logic for multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 96 (Aug. 2020), 29 pages. doi:10.1145/3408978

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019. Specifying concurrent programs in separation logic: morphisms and simulations. *PACMPL* 3, OOPSLA (2019), 161:1–161:30. doi:10.1145/3360587

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *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 (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 290–310. doi:10.1007/978-3-642-54833-8_16

Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911. http://ynot.cs.harvard.edu/papers/jfpsep07.pdf

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–67.

Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. 983–1002. doi:10.1109/SP40000.2020.00114

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *PACMPL* 1, ICFP (Sept. 2017), 17:1–17:29. doi:10.1145/3110261

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 55–74.

Filip Sieczkowski, Ales Bizjak, and Lars Birkedal. 2015. ModuRes: A Coq Library for Modular Reasoning About Concurrent Higher-Order Imperative Programming Languages. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 375–390. doi:10.1007/978-3-319-22102-1_25

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 80–95. doi:10.1145/3453483.3454031

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. doi:10.1145/3547631

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *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 (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 149–168. doi:10.1007/978-3-642-54833-8_9

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. *Proc. ACM Program. Lang.* 4, ICFP, Article 121 (Aug. 2020), 30 pages. doi:10.1145/3409003

Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V. Thakur. 2021. DICE*: A Formally Verified Implementation of DICE Measured Boot. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1091–1107. https://www.usenix.org/conference/usenixsecurity21/presentation/tao

Trusted Computing Group. [n. d.]. DICE. https://trustedcomputinggroup.org/work-groups/dice-architectures/.

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

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (dec 2019), 32 pages. doi:10.1145/3371119

Received 2024-11-15; accepted 2025-03-06