SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs

NIKHIL SWAMY, Microsoft Research, USA
ASEEM RASTOGI, Microsoft Research, India
AYMERIC FROMHERZ, Carnegie Mellon University, USA
DENIS MERIGOUX, Inria Paris, France
DANEL AHMAN, University of Ljubljana, Slovenia
GUIDO MARTÍNEZ, CIFASIS-CONICET, Argentina

Much recent research has been devoted to modeling effects within type theory. Building on this work, we observe that effectful type theories can provide a foundation on which to build semantics for more complex programming constructs and program logics, extending the reasoning principles that apply within the host effectful type theory itself.

Concretely, our main contribution is a semantics for concurrent separation logic (CSL) within the F* proof assistant in a manner that enables dependently typed, effectful F* programs to make use of concurrency and to be specified and verified using a full-featured, extensible CSL. In contrast to prior approaches, we directly derive the partial-correctness Hoare rules for CSL from the denotation of computations in the *effectful* semantics of non-deterministically interleaved atomic actions.

Demonstrating the flexibility of our semantics, we build generic, verified libraries that support various concurrency constructs, ranging from dynamically allocated, storable spin locks, to protocol-indexed channels. We conclude that our effectful semantics provides a simple yet expressive basis on which to layer domain-specific languages and logics for verified, concurrent programming.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Program Proofs, Separation Logic, Concurrency

ACM Reference Format:
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 (August 2020), 30 pages. https://doi.org/10.1145/3409003

1 INTRODUCTION

Proof assistants based on type theory can be a programmers' delight, allowing one to build modular abstractions coupled with strong specifications that ensure program correctness. Their expressive power also allows one to develop new program logics within the same framework as the programs themselves. A notable case in point is the Iris framework (Jung et al. 2018) embedded in Coq (The Coq development team), which provides an impredicative, higher-order, concurrent separation logic (CSL) (O'Hearn 2004; Reynolds 2002) within which to specify and prove programs.

Authors' addresses: Nikhil Swamy, Microsoft Research, USA; Aseem Rastogi, Microsoft Research, India; Aymeric Fromherz, Carnegie Mellon University, USA; Denis Merigoux, Inria Paris, France; Danel Ahman, University of Ljubljana, Slovenia; Guido Martínez, CIFASIS-CONICET, Argentina.

2475-1421/2020/8-ART121
https://doi.org/10.1145/3409003

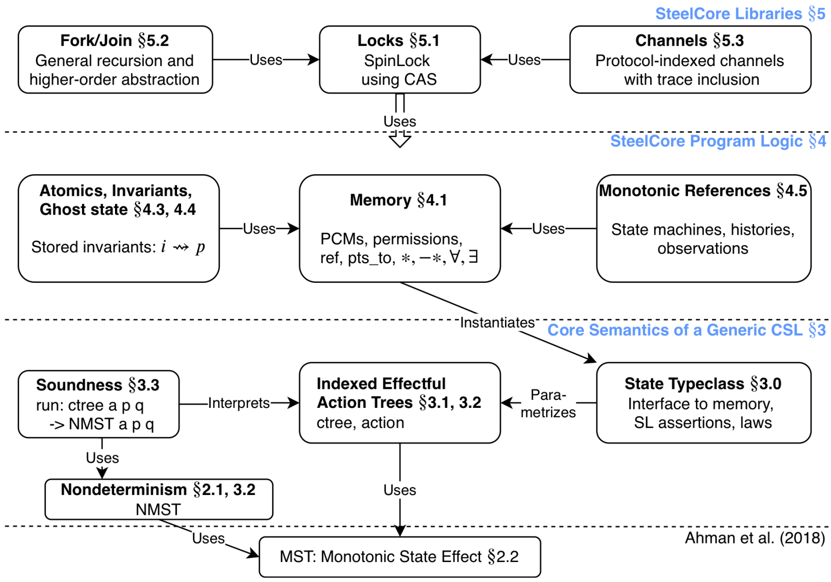

Fig. 1. An overview of SteelCore

Iris has been used to model various languages and constructs, and to verify many interesting programs (Chajed et al. 2019; Hinrichsen et al. 2019; Krogh-Jespersen et al. 2019). However, Iris is not in itself a programming language: it must instead be instantiated with a *deeply embedded* representation and semantics of one provided by the user. For instance, several Iris-based papers work with a mini ML-like language deeply embedded in Coq (Krebbers et al. 2017).

Taking a different approach, FCSL (Nanevski et al. 2008, 2014, 2019) embeds a predicative CSL in Coq enabling proofs of Coq programs (rather than embedded-language programs) within a semantics that accounts for effects like state and concurrency. This allows programmers to use the full power of type theory not just for proving, but also for programming, e.g., building dependently typed programs and metaprograms over inductive datatypes, with typeclasses, a module system, and other features of a full-fledged language. However, Nanevski et al.'s program logics are inherently predicative, which makes it difficult to express constructs like dynamically allocated invariants and locks, which are natural in impredicative logics like Iris.

In this paper, we develop a new framework called SteelCore that aims to provide the benefits of Nanevski et al.'s shallow embeddings, while also supporting dynamically allocated invariants and locks in the flavor of Iris. Specifically, we develop SteelCore in the *effectful* type theory provided by the F* proof assistant (Swamy et al. 2016). One of our main insights is that an effectful type theory is not only useful for programming; it can also be leveraged to build new program logics for effectful program features like concurrency. Building on prior work (Ahman et al. 2018) that models the effect of monotonic state in F*, we develop a semantics for concurrent F* programs while simultaneously deriving a CSL to reason about F* programs using the effect of concurrency. The use of monotonic state enables us to account for invariants and atomic actions entirely within SteelCore. The net result is that we can program higher order, dependently typed, generally recursive, shared-memory and message-passing concurrent F* programs and prove their partial correctness using SteelCore.

1.1 SteelCore: A Concurrent Separation Logic Embedded in F*

SteelCore is the core semantics of Steel, a DSL under development in F* for programming and proving concurrent programs. In this paper, we focus primarily on the semantics, leaving a detailed treatment of other aspects of the Steel framework to a separate paper. The structure of SteelCore is shown in Figure 1. Building on the monotonic state effect, we prove sound a generic program logic for concurrency, parametric in a memory model and a separation logic (§3). We instantiate this semantics with a separation logic based on partial commutative monoids, stored invariants, and state machines (§4). Finally, using this logic, we program verified, dependently typed, higher-order libraries for various kinds of concurrency constructs, culminating in a library for message-passing on typed channels (§5). We describe several novel elements of our contributions, next.

For starters, we need to extend F* with concurrency. To do this, we follow the well-known approach of encoding computational effects as definitional interpreters over free monads (Hancock and Setzer 2000; Kiselyov and Ishii 2015; Swierstra 2008; Xia et al. 2019). That is, we can represent computations as a datatype of (infinitely branching) trees of atomic actions. When providing a computational interpretation for action trees, one can pick an execution strategy (e.g., an interleaving semantics) and build an interpreter to run programs. The first main novelty of our work is that we provide an intrinsically typed definitional interpreter (Bach Poulsen et al. 2018) that both provides a semantics for concurrency while also deriving a CSL in which to reason about concurrent programs. Enabling this development is a new notion of indexed action trees, which we describe next.

Indexed action trees for structured parallelism. We represent concurrent computations as an instance of the datatype ctree st a pre post, shown below. The ctree type is a tree of atomic computational actions, composed sequentially or in parallel.

```
type ctree (st:state) : a:Type → pre:st.slprop → post:(a → st.slprop) → Type =
  | Ret : x:a → ctree st a (post x) post
  | Act : action pre post → ctree st a pre post
  | Par : ctree st a p q → ctree st a' p' q' → ctree st (a & a') (p `st.star` p') (λ (x, x') → q x `st.star` q' x')
  | Bind : ctree st a p q → ((x:a) → Dv (ctree st b (q x) r)) → ctree st b p r
```

The type ctree st a pre post is parameterized by an instance st of the state typeclass, which provides a generic interface to memories, including st.slprop, the type of separation logic assertions, and st.star, the separating conjunction. The index a is the result type of the computation, while pre and post are separation logic assertions. The Act nodes hold stateful atomic actions; Par nodes combine trees in parallel; while Bind nodes sequentially compose a computation with a potentially divergent continuation, as signified by the Dv effect label. Divergent computations are primitively expressible within F*, and are soundly isolated from its logical core of total functions by the effect system.

Interpreting action trees in the effects of nondeterminism and monotonic state. We interpret a term (e : ctree st a pre post) as both a computation e as well as a proof of its own partial correctness Hoare triple {pre} e : a {post}. To prove this sound, we define an interpreter that non-deterministically interleaves atomic actions run in parallel. The interpreter is itself an effectful F* function with the following (simplified) type, capturing our main soundness theorem:

```
val run (e:ctree st a p q) : NMST a st.evolves (λ m → st.interp p m) (λ _ x m' → st.interp (q x) m')
```

where NMST is the effect of monotonic stateful computations extended with nondeterminism. Here, we use it to represent abstract, stateful computations whose states are constrained to evolve according to the preorder st.evolves, and which when run in an initial state m satisfying the interpretation of the precondition p, produce a result x and final state m' satisfying the postcondition q x. As such, using the Hoare types of NMST, the type of run validates the Hoare rules of CSL given by the indexing structure on ctree. In doing so, we avoid the indirection of traces in Brookes's (2004) original proof of CSL as well as in the work of Nanevski et al. (2014).

Atomics and Invariants: Breaking circularities with monotonic state. Although most widely used concurrent programming frameworks, e.g., the POSIX pthread API, support dynamically allocated locks, few existing CSL frameworks actually support them, with some notable exceptions (Buisse et al. 2011; Dodds et al. 2016; Gotsman et al. 2007; Hobor et al. 2008; Jung et al. 2018). The main challenge is to avoid circularities that arise from storing locks that are associated with assertions about the memory in the memory itself. Iris, with its step-indexed model of impredicativity, can express this. However, other existing state of the art logics, including FCSL, cannot. In §4.3 and §4.4, we show how to leverage the underlying model of monotonic state to allocate a stored invariant, and to open and close it safely within an atomic command, without introducing step indexing.

PCMs, ghost state, state machines, and implicit dynamic frames. We base our memory model on partial commutative monoids (PCMs), allowing the user to associate a PCM of their choosing with each allocation unit. Relying on F*'s existing support for computationally irrelevant erased types, we can easily model *ghost state* by allocating values of erased types in the heap, and manipulating these values only using atomic ghost actions—all of which are erased during compilation. PCMs in SteelCore are orthogonal from ghost state: they can be used both to separate and manage access permissions to both concrete and ghost state—in practice, we use fractional permissions to control read and write access to references. Further, SteelCore includes a notion of *monotonic* references, which when coupled with F*'s existing support for ghost values and invariants, allow programmers to code up various forms of *state machines* to control the use and evolution of shared resources. Demonstrating the flexibility of our semantics, we extend it to allow augmenting CSL assertions with frameable heap predicates, a style that combines CSL with *implicit dynamic frames* (Smans et al. 2012) within the same mechanized framework.

Putting it to work. We present several examples showing SteelCore at work, aiming to illustrate the flexibility and extensibility of the logic and its smooth interaction with dependently typed programming in F*. Starting with an atomic compare-and-set (CAS) instruction, we program verified libraries for spin-locks, for fork/join parallelism, and finally for protocol-indexed channel types. Our channel-types library showcases dependent types at work with SteelCore: its core construct is a type of channels, chan p, where p is itself a free-monad-like computation structure "one-level up" describing an infinite state machine on types. We prove, once and for all, that programs using a c:chan p exchange a trace of messages on c accepted by the state machine p.

Mechanization. SteelCore is a fully mechanized CSL embedded in F*, and applicable to F* itself. The code and proofs are available from https://fstar-lang.org/papers/steelcore.

Summary of contributions. In summary, the main contributions of our work include the following:

- A new construction of indexed, effectful action trees, mixing data and effectful computations to represent concurrent, stateful and potentially divergent computations, with an indexing structure capturing the proof rules of a generic CSL.
- An intrinsically typed definitional interpreter that interprets our effectful action trees into another effect, namely the effect of nondeterminism layered on the effect of monotonic state. This provides both a new style of soundness proof for CSL, as well as providing a reference executable semantics for our host language F* extended with concurrency.
- An instantiation of our semantics with a modern CSL inspired by recent logics like Iris, with a core memory model based on partial commutative monoids and support for dynamically

allocated invariants. Relying on the underlying semantic model of monotonic state is a key element, allowing us to internalize the step-indexing that is necessary in Iris for dealing soundly with invariants.

- We use our logic to build several verified libraries, programmed in and usable by dependently typed, effectful host-language programs, validating our goal of providing an Iris-style logic applicable to a shallow rather than a deeply embedded programming language.

2 F⋆ BACKGROUND AND BASIC INDEXED ACTION TREES

F⋆ is a program verifier and a proof assistant based on a dependent type theory with a hierarchy of predicative universes (like Coq or Agda). F⋆ also has a dependently typed metaprogramming system inspired by Lean and Idris (called Meta-F⋆ (Martínez et al. 2019)) that allows using F⋆ itself to build and run tactics for constructing programs or proofs. More specific to F⋆ is its effectful type system, extensible with user-defined effects, and its use of SMT solving to automate some proofs.

Basic Syntax. F⋆ syntax is roughly modeled on OCaml (val, let, match etc.) although there are many differences to account for the additional typing features. Binding occurrences b of variables take the form x:t, declaring a variable x at type t; or #x:t indicating that the binding is for an implicit argument. The syntax $\lambda(b_1) \ldots (b_n) \to t$ introduces a lambda abstraction, whereas $b_1 \to \ldots \to b_n \to c$ is the shape of a curried function type. Refinement types are written b{t}, e.g., x:int{x≥ 0} is the type of non-negative integers (i.e., nat). As usual, a bound variable is in scope to the right of its binding; we omit the type in a binding when it can be inferred; and for non-dependent function types, we omit the variable name. For example, the type of the pure append function on vectors is written #a:Type → #m:nat → #n:nat → vec a m → vec a n → vec a (m + n), with the two explicit arguments and the return type depending on the three implicit arguments marked with '#'. The type of pairs in F⋆ is represented by a & b with a and b as the types of the first and second components respectively. In contrast, dependent tuple types are written as x:a & b where x is bound in b. A dependent pair value is written (| e, f |) and we use $x._1$ and $x._2$ for the first and second dependent projection maps.

2.1 A Total Semantics of Concurrency

As an introduction to F⋆ and a warm-up towards the main ideas behind our indexed actions trees, we start by presenting a very simple total semantics for concurrency. Relying only on the pure rather than effectful features of F⋆, some of the ideas in this section should also transfer to pure type theories like Agda or Coq. However, our main construction involves a partial-correctness semantics with effects like divergence, which may be harder to develop in non-effectful type theories.

A disclaimer: total correctness for realistic concurrent programs (e.g., under various scheduling policies) is a thorny issue that our work does not address at all. For this introductory example, we focus only on programs with structured parallelism, without any other synchronization constructs, and where loop bounds do not depend on effectful computations.

Our first step is to define a type of state-passing atomic actions, action_tot a = state → Tot (a & state). This is the type of a function that transforms an initial state to a pair of an a-typed result and a final state. The Tot at the right of the arrow is a *computation type* emphasizing that this is a *total* function; we will soon see other kinds of computation types and effectful arrows in F⋆. All unannotated arrows are Tot by default.

Action trees for concurrency. To model concurrency, we define an inductive type ctree_total, for trees of action_tot actions, indexed by a natural number (used for a termination proof). This is our first and simplest instance of an indexed action tree, one that could easily be represented in another type theory. In §3, we will enrich ctree_total to the CSL-indexed ctree shown in §1.

```
type ctree_total : nat → Type → Type =
| Ret : #a:_ → x:a → ctree_total 0 a
| Act : #a:_ → act:action_tot a → ctree_total 1 a
| Par : (#aL #aR #nL #nR:_) → ctree_total nL aL → ctree_total nR aR → ctree_total (nL+nR+1) (aL & aR)
| Bind : (#a #b #n1 #n2:_) → f:ctree_total n1 a → g:(x:a → ctree_total n2 b) → ctree_total (n1+n2+1) b
type nctree_total (a:Type) = n:nat & ctree_total n a
```

The type ctree_total induces a monad-like structure (under a suitable equivalence that quotients the use of Bind) by representing computations as trees of finite depth, with pure values (Ret) and atomic actions (Act) at the leaves; a Bind node for sequential composition of two subtrees; and a Par node for combining a left and a right subtree. The monad induced by ctree_total differs from the usual construction of a free monad for a collection of actions by including an explicit Bind node, instead of defining the monadic bind recursively. This makes ctree_total more similar to Piróg et al.'s (2018) scoped operations, with f being in the "scope" of Bind. The nat index counts the number of Act, Par and Bind nodes. We also define an abbreviation nctree_total a to package a tree with its index as a dependent pair.

A definitional interpreter for ctree_total. To give a semantics to ctree_total, we interpret its action trees in an interleaving semantics for state-passing computations, relying on a boolean tape to resolve the nondeterminism inherent in the Par nodes. To that end, we define a state and nondeterminism monad, with sample, get, and put actions:

```
type tape = nat → bool
type nst (a:Type) = tape & nat & state → a & nat & state
let return (a:Type) (x:a) : nst a = λ(_, n, s) → x, n, s
let bind (a b:Type) (f:nst a) (g:a → nst b) : nst b = λ(t, n, s) → let x, n1, s1 = f (t, n, s) in (g x) (t, n1, s1)
let sample () : nst bool = λ(t, n, s) → t n, n+1, s
let get () : nst state = λ(_, n, s) → s, n, s
let put (s:state) : nst unit = λ(_, n, _) → (), n, s
```

We can now interpret ctree_total trees as nst computations. It should be possible to define such an interpreter in many type theories, in a variety of styles. Here, we show one way to program it in F*, making use of its effect system to package the nst monad as a *user-defined effect*.

A user-defined effect in F* introduces a new abstract computation type backed by an existing F* definition (in our case, a computation type NST backed by the monad nst). Based on work by Swamy et al. (2011b), computations and computation types enjoy some conveniences in F*. In particular, F* automatically elaborates sequencing and application of computations using the underlying monadic combinators, without the need for do-notation, e.g., let in NST is interpreted as bind in nst. Further, F* supports sub-effects to lift between computation types, relying on a user-provided monad morphism, e.g., pure computations are silently lifted to any other effect. The following incantation turns the nst monad into the NST effect, with three actions, sample, get and put.

```
total new_effect { NST : a:Type → Effect with repr=nst; return=return; bind=bind}
let sample () = NST?.reflect (sample())     let get () = NST?.reflect (get())     let put s = NST?.reflect (put s)
```

The type of sample is unit → NST bool, where the computation type at the right of the arrow indicates that sample has NST effect—calling sample in a pure context is rejected by F*'s effect system. We will soon see examples of computation types with a richer indexing structure. The total qualifier on the first line ensures that all the computations in the NST effect are proved terminating.

Using NST, we build an interpreter for ctree_total trees by defining run as the transitive closure of a single step. The main point of interest is the last case of step, reducing a Par l r node by sampling a boolean and recursing to evaluate a step on either the left or the right.

```
let reduct #a (r:nctree_total a) = r':nctree_total a{ Return? r' ∨ r'.1 < r.1 }
let rec step #a (redex:nctree_total a) : NST (reduct redex) (decreases redex.1)
  match redex.2 with
  | Ret _ → redex | Act act → let s0 = get () in let x, s1 = act s0 in put s1; (| _, Ret x |)
  | Bind (Ret x) g → (| _, g x |) | Bind f g → let (| _, f' |) = step (| _, f |) in (| _, Bind f' g |)
  | Par (Ret x) (Ret y) → (| _, Ret (x, y) |)
  | Par l (Ret y) → let (| _, l' |) = step (| _, l |) in (| _, Par l' (Ret y) |)
  | Par (Ret x) r → let (| _, r' |) = step (| _, r |) in (| _, Par (Ret x) r' |)
  | Par l r →
    if sample () then let (| _, l' |) = step (| _, l |) in (| _, Par l' r |) else let (| _, r' |) = step (| _, r |) in (| _, Par l r' |)
let rec run #a (p:nctree_total a) : NST (nctree_total a) (decreases p.1) = if Return? p then p else run (step p)
```

Having concluded our basic introduction to F⋆ and indexed action trees, we move beyond totality to general recursion and other effects, and in §3 to indexed, effectful action trees.

2.2 The Effects of Divergence and Monotonic State

Dv : an effect for divergence. In addition to user-defined effects like NST, F⋆ provides an abstract primitive effect of divergence represented by the computation type Dv. As with any other effect, the Dv effect is isolated from the logical core of F⋆: general recursive functions in Dv cannot mistakenly be used as proofs. Swamy et al. (2016) prove the soundness of a core F⋆ calculus in a partial correctness setting for divergent computations, while also proving that Tot terms are normalizing. As such the following term is well-typed in F⋆: `let rec loop : unit → Dv unit = λ() → loop ()`. From the perspective of F⋆'s logical core, `a → Dv b` is an abstract, un-eliminable type.

MST : an effect for monotonic state. MST is another effect in F⋆ for computations that read and write primitive state, while restricting the state to evolve according to a given preorder, i.e., a reflexive, transitive relation. Ahman et al. (2018) observe that for such computations, witnessing a property p of the state that is invariant under the preorder is sufficient to recall that p is true in the future. Ahman et al. propose the following signature for such an MST effect, and prove the partial correctness of the Hoare logic encoded in the indexes of MST against an operational semantics for a λ-calculus with primitive state.

```
effect MST (a:Type) (state:Type) (p:preorder state) (req:state → prop) (ens:state → a → state → prop)
```

When executing a computation (`c : MST a state p req ens`) in an initial state `s0:state` satisfying `req s0`, the computation either diverges, or returns a value `x:a` in a final state `s1:state` satisfying `ens s0 x s1`. Further, the state is transformed according to the preorder p, i.e., the initial and final states are related by `p s0 s1`. The MST effect provides the following actions—for readability, we tag the pre- and postcondition with requires and ensures respectively:

• *Get* the current state:

```
val get #state #p () : MST state state p (requires λs → ⊤) (ensures λs0 r s1 → s0==s1 ∧ r==s0)
```

• *Put* the state, but only when the new state s1 is related to the old one s by p:

```
val put #state #p (s1:state) : MST unit state p (requires λs → p s s1) (ensures λ_ _ s → s==s1)
```

• *Witness* stable predicates: A stable predicate is maintained across preorder-respecting state evolutions. The witness action proves an abstract proposition, `witnessed q`, attesting that the stable predicate q is valid.

```
let stable_sprop #state (p:preorder state) = q:(state → prop){∀ s0 s1. q s0 ∧ p s0 s1 ⟹ q s1}
val witnessed #state #p (q:stable_sprop p) : prop
```

```
val witness #state #p (q:stable_sprop p) : MST unit state p (λ s0 → q s0) (λ s0 _ s1 → witnessed q ∧ s0==s1)
```

- *Recall* stable predicates: Having witnessed q, one can use recall q to re-establish it at any point.

```
val recall #state #p (q:stable_sprop p{witnessed q}) : MST unit state p (λ s0 → ⊤) (λ s0 _ s1 → s0==s1 ∧ q s1)
```

As such, the MST effect provides a small program logic for monotonic state computations, which we leverage for SteelCore's semantic foundation in §4.

NMST: extending MST with nondeterminism. The MST effect only models state and does not provide the nondeterminism we need for interleaving the subtrees of Par nodes. Therefore, we layer a user-defined effect of nondeterminism on top of MST, and define a new effect NMST that provides an additional sample action—much as we did in the previous section. We use NMST in the next section as the target denotation for the semantics of a generic partial correctness separation logic.

Erased types and the ghost effect. A final remark on F⋆'s effect system has to do with its support for erasure. As described by Swamy et al. (2016), F⋆ encapsulates computations that are not meant to be executed in a *ghost effect*. Terms with ghost effect can be used in proofs and specifications, but not in executable code. Further, F⋆ provides an extensible mechanism to mark certain types as non-informative, including, notably the type erased t. Eliminating a term of a non-informative type (e.g., pattern matching on it) incurs a ghost effect, ensuring that such uses never occur in computationally relevant code. F⋆ also supports an implicit coercion mechanism that allows an erased t to be used as a t (with a ghost effect)—such coercions are only legal in computationally irrelevant contexts, e.g., proofs and specifications. F⋆'s extraction pipeline to several target languages begins by erasing terms with non-informative types or ghost effect to the unit value (). In SteelCore, we rely on these features implicitly. However, a full treatment of the erasure of SteelCore terms for efficient extraction is beyond the scope of this paper—indeed, we have yet to extract and run any Steel program, though we do not foresee any major difficulties in doing so.

3 INDEXED ACTION TREES AND A PARTIAL CORRECTNESS SEPARATION LOGIC

Recall from §1 that our goal is to define the indexed action trees with the following type:

```
type ctree (st:state) (a:Type) (pre:st.slprop) (post:a → st.slprop) : Type
```

The type is indexed by st:state, a typeclass encapsulating (at least) the type of the memory st.mem and the type of separation logic assertions on the memory st.slprop. Intuitively, a ctree st a fp0 fp1 is the type of a potentially divergent, concurrent program manipulating shared state of type st.mem. The program expects the fp0 footprint of some initial memory m0:st.mem. When run in m0, it may diverge or produce a result:a and m1:st.mem, providing the (fp1 result) fragment of m1 to the context.

The state typeclass for the semantics is shown below. First, we define a pre_state containing all the operations we need. A state is a refinement of pre_state satisfying various laws.

```
type pre_state = { mem: Type; (* The type of the underlying memory *)
                   slprop: Type; (* The type of separation logic assertions *)
                   equals: equiv slprop; (* An equivalence relation on slprops *)
                   emp: slprop; (* With a unit *)
                   star: slprop → slprop → slprop; (* And separating conjunction *)
                   interp: slprop → mem → prop; (* Interpreting slprop as a mem predicate *)
                   evolves: preorder mem; (* A preorder for MST: constrains how the state evolves *)
                   inv: mem → slprop; (* A separation logic invariant on the memory *) }
let st_laws (st:pre_state) =
  associative st.equals st.star ∧ commutative st.equals st.star ∧ is_unit st.emp st.equals st.star ∧
  interp_extensionality st.equals st.interp ∧ star_extensionality st.equals st.star ∧ affine st
```

```
type state = s:pre_state{st_laws s}
```

We expect emp and star to form a commutative monoid over slprop and the equivalence relation equals. The relation interp interprets an slprop as a predicate on mem and we expect the interpretation of star to be compatible with slprop-equivalence. We also expect the interpretation to be affine, in the sense that interp (p `st.star` q) m ⟹ interp q m.

As we will see in §4.1 we can instantiate our semantics with a separation logic containing the full gamut of connectives, including conjunction, disjunction, separating implication, and universal and existential quantification. The preorder evolves and the invariant inv are opaque as far as the semantics is concerned—in §4, we'll instantiate them in a way that allows us to model a source of freshness for allocating ref cells and also to support dynamically allocated invariants. In the following, we write ∗ for st.star where st is clear from the context.

3.1 Frame-Preserving Actions

To define the type of action trees ctree, let's start by defining the type of atomic actions at the leaves of the tree:

```
let action #st a (fp0:st.slprop) (fp1:a → st.slprop) =
  unit → MST a st.mem st.evolves
  (requires λm0 → st.interp (st.inv m0 ∗ fp0) m0)
  (ensures λm0 x m1 → st.interp (st.inv m1 ∗ fp1 x) m1 ∧ preserves_frame fp0 (fp1 x) m0 m1)
```

An action is an MST computation that requires its initial footprint fp0 to hold on the initial state m0. It returns an x:a and ensures its final footprint fp1 x on the final state m1. In both the pre- and postcondition, we expect st.inv to hold separately. Finally, and perhaps most importantly, the preserves_frame side condition ensures that actions are frameable. We elaborate on that next.

Frame preservation. We would like to derive a framing principle for computations as a classic frame rule (and its generalization, the rule for separating parallel composition). As observed by Dinsdale-Young et al. (2013), it is sufficient for the leaf actions to be frame-preserving for computations to be frame preserving too. To that end, the definition of preserves_frame (that an action must provide in its postcondition) states that all frames separate from st.inv m0 ∗ pre and valid in the initial state m0 remain separate from st.inv m1 ∗ post and are valid in m1.

```
let preserves_frame #st (pre post:st.slprop) (m0 m1:st.mem) =
  ∀(frame:st.slprop). st.interp (st.inv m0 ∗ pre ∗ frame) m0 ⟹ st.interp (st.inv m1 ∗ post ∗ frame) m1
```

3.2 CSL-Indexed Action Trees with Monotonic State

Figure 2 shows the way we represent computation trees in SteelCore—extending the ctree type from the introduction. To reduce clutter, we omit binders for implicit arguments—in the type of each constructor of the inductive ctree, we only bind names that do not appear free in other arguments of the constructor. These trees differ from the simple action trees we used in §2.1. The additional indexing structure in each case of ctree posits the proof rules of a program logic for reasoning about ctree computations. In §3.3, we show that this logic is sound by denoting ctree st a fp0 fp1 trees via an interleaving, definitional interpreter into NMST computations. As NMST computations are potentially divergent, we do not need to prove termination of the definitional interpreter. Thus the type ctree does not carry a natural number index as we did in §2.1.

We describe the structure of ctree in detail, discussing each of its constructors in turn.

Atomic actions. At the leaves of the tree, we have nodes of the form Act e, for some action e: the index of the computation inherits the indexes of the action.

```
type ctree (st:state) : a:Type → fp0:st.slprop → fp1:(a → st.slprop) → Type =
| Act: e:action a fp0 fp1 → ctree st a fp0 fp1
| Ret: fp:(a → st.slprop) → x:a → ctree st a (fp x) fp
| Bind: f:ctree st a fp0 fp1 → g:(x:a → Dv (ctree st b (fp1 x) fp2)) → ctree st b fp0 fp2
| Par: cL:ctree st aL fp0L fp1L → cR:ctree st aR fp0R fp1R →
    ctree st (aL & aR) (fp0L ∗ fp0R) (λ (xL, xR) → fp1L xL ∗ fp1R xR)
| Frame: c:ctree st a fp0 fp1 → f:st.slprop → ctree st a (fp0 ∗ f) (λ x → fp1 x ∗ f)
| Sub: c:ctree st a fp0 fp1 { sub_ok fp0 fp1 fp0' fp1' } → ctree st a fp0' fp1'
with
let sub_ok #st fp0 fp1 fp0' fp1' = fp0' `stronger_than` fp0 ∧ fp1' `weaker_than` fp1
let stronger_than #st fp0' fp0 = ∀m f. st.interp (fp0' ∗ f) m ⟹ st.interp (fp0 ∗ f) m
let weaker_than #st fp1' fp1 = ∀x m f. st.interp (fp1 x ∗ f) m ⟹ st.interp (fp1' x ∗ f) m
```

Fig. 2. SteelCore's representation of computations as indexed action trees

Returning pure values. Also at the leaves of the tree are Ret fp x nodes, which allow returning a pure value x in a computation. The Ret node is parametric in a footprint fp, and the indexes on ctree state that in order to provide fp, we expect the fp x to hold in the initial state m0. An alternative formulation could also have used Ret : x:a → ctree st a st.emp (λ_ → st.emp), although, as we discuss in §3.3, this form is less convenient in conjunction with the frame rule.

Sequential composition. The Bind f g node sequentially composes f and g. Its indexing structure should appear fairly canonical. The footprints of f and g are "chained" as in Atkey's (2009) parameterized monads, except our indexes (notably fp2) are dependent. The computation type of g has the Dv effect, indicating a potentially divergent continuation.

Parallel composition. Par cL cR composes computations in parallel. The indexing structure yields the classic CSL rule for parallel composition of computations with disjoint footprints.

Structural rules: Framing and Subsumption. The Frame c f node preserves the frame f across the computation c. The Sub c node allows strengthening the initial footprint and weakening the final footprint of c. These nodes directly correspond to the canonical CSL frame and consequence rules.

These structural rules are essential elements of our representation. The indexing structure of ctree defines a program logic and the structural rules are manifested as a kind of re-indexing, which must be made explicit in the inductive type as additional constructors. Further, given such structural rules, the need for a separate Bind, as opposed to continuations in each node, becomes evident. Consider verifying a Hoare triple {P1 ∗ P} a1; a2; a3 {Q}, where a1, a2, a3 are actions with { P1 } a1; a2 { P1 }, and { P1 ∗ P } a3 { Q }. The canonical proof frames P across a1; a2 together, which is trivial to do with our representation, as Bind (Frame (Bind (Act a1) (λ_ → (Act a2))) P) (λ_ → (Act a3)). The frames can be easily added *outside* of a proof derivation, making the proofs modular. However, if the continuations were part of the Act (and Par) nodes, such a structural frame rule would not apply. We would have to bake-in framing in the Act nodes, and even then we would have to frame P across a1 and a2 individually. This makes the proofs less modular, since we can't directly use the given derivation { P1 } a1; a2 { P1 }.

Although we include Frame and Sub, we lack the structural rule for disjunction. Accommodating disjunction in a shallow embedding is hard to do, since it requires giving to the same computation more than one type. One possibility may be to adopt a relational specification style, as Nanevski et al. (2010) do—we leave an exploration of this possibility to future work. Meanwhile, as we instantiate the semantics with a state model in §4, we also provide several lemmas to destruct combinations of separating conjunctions and existentials (with disjunctions as a special case).

3.3 Soundness

To prove the soundness of the proof rules induced by the indexing structure of ctree, we follow the strategy outlined in §2.1, with NMST from §2.2 as the target denotation. Our goal is to define an interpreter with the following type, showing that it maintains the memory invariant while transforming fp0 to fp1 x.

```
val run #st #a #fp0 #fp1 (f:ctree st a fp0 fp1) : NMST a st.mem st.evolves
  (requires λm0 → st.interp (st.inv m0 ∗ fp0) m0) (ensures λm0 x m1 → st.interp (st.inv m1 ∗ fp1 x) m1)
```

As before, we proceed by first defining a single-step interpreter and then closing it transitively to build a general recursive, multi-step interpreter. The single-step interpreter has the following type, returning (as in §2.1) the reduced computation tree packaged with all its indices.

```
type reduct #st a = | Reduct: fp0:_ → fp1:_ → ctree st a fp0 fp1 → reduct a
val step (f:ctree st a fp0 fp1) : NMST (reduct a) st.mem st.evolves
  (requires λm0 → st.interp (st.inv m0 ∗ fp0) m0)
  (ensures λm0 (Reduct fp0' fp1' _) m1 →
    st.interp (st.inv m1 ∗ fp0') m1 ∧ preserves_frame fp0 fp0' m0 m1 ∧ fp1' `stronger_than` fp1)
```

In addition to requiring and ensuring the invariant and footprint assertions, we have additional inductive invariants that are needed to take multiple steps. As is typical in such proofs, one needs to show that given a term in a context E[c], reducing c by a single step produces c' that can be correctly typed within the same context, i.e., E[c'] must be well-typed. Towards that end, we need two main properties of step: (a) preserves_frame, defined in §3.1, ensures that the reduct c' can be framed with any frame used with the redex c; and (b) that the postcondition fp1' of the reduct c' is stronger than the postcondition fp1 of the redex c. Interestingly, we don't explicitly need to show that the precondition of the reduct is weaker than the precondition of the redex: that the initial footprint of the reduct holds in m1 is enough. We show all the main cases of the single-step reduction next. In all cases, the code is typechecked as shown, with proofs semi-automated by F*'s SMT solving backend.

Framing. The code below shows stepping through applications of the Frame c0 f rule. In the case where c0 is a Ret node, we remove the Frame node and restore the derivation by extending the footprint of the Ret node to include the frame f—this is one reason why it is convenient to have Ret nodes with parametric footprints, rather than just the emp footprint.

```
let rec step #st #a #fp0 #fp1 (c:ctree st a fp0 fp1) = match c with | . . .
  | Frame (Ret fp0' x) f → Reduct (fp0' x ∗ f) (λ x → fp0' x ∗ f) (Ret (λ x → fp0' x ∗ f) x)
  | Frame c0 f → let m0 = get () in let Reduct fp0' fp1' c' = step c0 in let m1 = get () in
                        preserves_frame_star fp0 fp0' m0 m1 f; Reduct (fp0' ∗ f) (λ x → fp1' x ∗ f) (Frame c' f)
```

When c0 is not a Ret, we recursively evaluate a step within c0 and then reconstruct a Frame around its reduct c'. This proof step makes use of a key lemma, preserves_frame_star, which states preserves_frame fp0 fp0' m0 m1 $\Longrightarrow$ preserves_frame (fp0 ∗ f) (fp0' ∗ f) m0 m1).

Subsumption. Reductions of the other structural rule, Sub, is simpler, we just remove the Sub node, as shown below; the refinement sub_ok on the c argument of the Sub node allows F* to prove the inductive invariants of step. Although we remove Sub nodes, the rule for sequential composition (next) adds them back to ensure that the reduct remains typeable in context. An alternative may have been to treat Sub like we treat Frame, however, this form is more convenient when adding support for implicit dynamic frames (as mentioned briefly below).

```
| Sub #fp0' #fp1' c → Reduct fp0' fp1' c
```

Sequential composition. In case f is fully reduced to a Ret node, we simply apply the continuation g. Otherwise, we take a step in f producing a reduct f' that may have a stronger final footprint. To reconstruct the Bind node, we need to strengthen the initial footprint of g with the final footprint of f', we do so by wrapping g with a Sub:

```
| Bind #fp2 (Ret fp0 x) g → Reduct (fp0 x) fp2 (g x)
| Bind #fp0 #fp1 #fp2 f g → let Reduct fp0' fp1' f' = step f in
                            Reduct fp0' fp2 (Bind f' (Sub #fp1 #_ #fp1' #_ g))
```

Parallel composition. The structure of reducing Par nodes is essentially the same as in §2.1. When both branches are Ret nodes, we simply create a reduct with a Ret node capturing the two values.

```
| Par (Ret fp0L xL) (Ret fp0R xR) →
  Reduct (fp0L xL ∗ fp0R xR) (λ (xL, xR) → fp0L xL ∗ fp0R xR) (Ret (λ (xL, xR) → fp0L xL ∗ fp0R xR) (xL, xR))
| Par #aL #fp0L #fp1L cL #aR #fp0R #fp1R cR →
  if sample() then let m0 = get () in let Reduct fp0L' fp1L' cL' = step cL in let m1 = get () in
                   preserves_frame_star fp0L fp0L' m0 m1 fp0R;
                   Reduct (fp0L' ∗ fp0R) (λ (xL, xR) → fp1L' xL ∗ fp1R xR) (Par cL' cR)
  else ... (∗ similarly for the right branch ∗)
```

When only one of the branches is Ret, we descend into the other one (we elide these cases from the presentation). When both the branches are candidates for reduction, we sample a boolean and pick either the left or right branch to descend into. Having obtained a reduct, we reconstruct the Par node, by appropriately framing the initial footprint of the unreduced branch, as shown above.

Atomic actions. An Act e node is reduced by applying it, and returning its result in a Ret node.

```
| Act #fp1 e → let x = e () in Reduct (fp1 x) fp1 (Ret fp1 x)
```

Multi-step interpreter. Implementing a general recursive, multi-step interpreter is straightforward: we recursively evaluate single steps until we reach a Ret node. The type of the interpreter, shown below, is the main statement of partial correctness for our program logic.

```
let rec run #st #a #fp0 #fp1 (f:ctree st a fp0 fp1) : NMST a st.mem st.evolves
  (requires λm0 → st.interp (st.inv m0 ∗ fp0) m0) (ensures λm0 x m1 → st.interp (st.inv m1 ∗ fp1 x) m1)
= match f with | Ret _ x → x | _ → let Reduct _ _ f' = step f in run f'
```

The type states that when run in an initial state m0 satisfying the memory invariant st.inv m0 and separately the footprint assertion fp0, the code either diverges or returns x:a in a final state m1 with the invariant st.inv m1, and the footprint assertion fp1 x. The inductive stronger_than invariant about the step function providing a stronger postcondition is crucial to the proof here: the recursive call to run ensures the validity of the post-footprint of f' in the final memory, we need the inductive invariant to relate it to the post-footprint of f, as required by the postcondition of run.

Extension: Implicit Dynamic Frames. While we have presented the action trees, and hence the CSL semantics, using only the slprop indices, our actual implementation also contains two further indices for specifications in the style of implicit dynamic frames (Smans et al. 2012). In our representation type, ctree_idf st a fp0 fp1 req ens, the last two indexes req and ens indicate the pre- and postcondition of a computation, where the precondition is a *fp0-dependent* predicate on the initial memory and postcondition is a two-state predicate that is fp0-dependent on the initial memory and fp1-dependent on the final one. The dependency relation captures the requirement that the predicates are "self-framing" (Parkinson and Summers 2012), i.e., the slprop footprint indices fp0 and fp1 limit the parts of the memory that these predicates can depend on. In addition to these indices, we add frameable memory predicates to the Frame and Par rule.

For example, the following is the Frame rule in our implementation (fp_prop f is the type of an f dependent memory predicate):

```
| Frame: c:ctree_idf st a fp0 fp1 req ens → f:st.slprop → p:fp_prop f →
          ctree_idf st a (fp0 ∗ f) (λ x → fp1 x ∗ f) (frame_req req p) (frame_ens ens p)
```

with

```
let frame_req req p = λm → req m ∧ p m          let frame_ens ens p = λm0 x m1 → ens m0 x m1 ∧ p m1
```

Finally, our soundness theorem, i.e. the specification of the run function, requires the precondition (req) of the computation and ensures its postcondition (ens). To prove this theorem, we have to enhance the inductive invariant of the step function to also mention weakening of the preconditions and strengthening of the postconditions. By incorporating both the slprop indices, and implicit dynamic frame-style requires- and ensures-indices, we hope to provide more flexibility in writing specifications for client programs.

Discussion. It's worth noting that although we've built a definitional interpreter with an interleaving semantics for concurrent programs, we do not intend to run programs using run, since it would be very inefficient, primarily because the interleaving semantics is encoded via sampling, but also because the representation includes a full proof tree, including the structural rules like Frame and Sub. Instead, relying on F*'s support for extraction to OCaml and C, we intend to compile effectful, concurrent programs to native concurrency in the target platforms, e.g., POSIX threads. As such, the main value of run is its proof of soundnesss: we now have in hand a semantics for concurrent programs and a means to reason about them deductively using a concurrent separation logic. We've built our semantics atop the effect of monotonic state, parameterizing our semantics with a preorder governing how the state evolves. So far, this preorder has not played much of a role. For the payoff, we'll have to wait until we instantiate the state interface, next.

4 THE STEELCORE PROGRAM LOGIC

The core semantics developed in the previous section provides a soundness proof for a generic, minimalistic concurrent separation logic. In this section, we instantiate the semantics with a model of state, assertions, invariants and actions defining the logic for Steel programs.

The logic includes the following main features:

- A core heap model addressed by typed references with explicit, manually managed lifetimes.
- Each heap cell stores a value in a user-chosen, cell-specific partial commutative monoid, supporting various forms of sharing disciplines and stateful invariants, including, e.g., a discipline of fractional permissions (Boyland 2003), for sharing among multiple threads.
- A separation logic, with all the usual connectives.
- Ghost state and ghost actions, relying on F*'s existing support for erasure.
- A model of atomic actions, including safe composition of ghost and concrete actions.
- Invariants, that can be dynamically allocated and freely shared among multiple threads and accessed and restored by atomic actions only.
- Monotonic references controlling the evolution of memory, built using preorders from the underlying monotonic state effect.

The result is a full-featured separation logic shallowly embedded in F⋆, with a fully mechanized soundness proof, and applicable directly to dependently typed, higher order, effectful host language programs. We provide several small examples throughout this section, and further ones in §5.

4.1 Memory

At the heart of our state model is a representation of memory, as outlined in the type below.

```
let addr = nat        let heap = addr → option cell        type mem = { heap:heap; ctr:nat; istore:istore }
let mem_inv m : slprop = (∀ i.i ≥ m.ctr ⟹ m.heap i == None) ∧ istore_inv m
let mem_evolves m0 m1 = h_evolves m0.heap m1.heap ∧ i_evolves m0.istore m1.istore ∧ m0.ctr ≤ m1.ctr
```

A heap is a map from abstract addresses (nat) to heap cells defined below.[1] A memory augments a heap with two important fields of metadata. First, we have a counter to provide fresh addresses for allocation (with an invariant guaranteeing that all addresses above ctr are unused). Second, we have an istore for tracking dynamically allocated invariants. Actions maintain a memory invariant inv and the memory is constrained to evolve according to the preorder mem_evolves. We discuss all these elements in detail throughout this section.

For the definition of heap cells, we make use of partial commutative monoids (PCMs). Using PCMs to represent state is typical in the literature: starting at least with the work of Jensen and Birkedal (2012), PCMs have been used to encode a rich variety of specifications, ranging from various kinds of sharing disciplines, fictional separation, and also various forms for state machines. We represent PCMs as the typeclass pcm a shown below, where we account for partiality by restricting the domain of op by a predicate composable. We write $\preceq_p$ for the partial order induced by p:pcm a.

```
type pcm (a:Type) = { one:a; composable: a → a → prop {sym composable};
                      op: x:a → y:a{composable x y} → a { comm op ∧ assoc op ∧ is_unit op one } }
let (≼) (#a:Type) (#pcm:pcm a) (x y : a) = ∃frame. pcm.op x frame == y
type cell = | Cell: a:Type → pcm:pcm a → v:a → cell
```

A cell is a triple of a type a, an instance of the typeclass of partial commutative monoids (pcm a), and a value of that type.[2]

With this representation of heap, it's relatively straightforward to define two functions, disjoint and join, which we use to separate and combine disjoint memories. For an address that appears in both heaps, we require the cell at that address to agree on the type, the PCM instance, and for the values to be composable in the PCM.

```
let disjoint_addr (h h':heap) (a:addr) = match h a, h' a with
  | Some (Cell t pcm v), Some (Cell t' pcm' v') → t==t' ∧ pcm==pcm' ∧ pcm.composable v v' | _ → ⊤
let disjoint h0 h1 = ∀a. disjoint_addr h0 h1 a
let join (h0:heap) (h1:heap{disjoint h0 h1}) = λa → match h0 a, h1 a with
  | None, None → None | None, Some x | Some x, None → Some x
  | Some (Cell t pcm v0), Some (Cell _ _ v1) → Some (Cell t pcm (pcm.op v0 v1))
```

[1] In our F⋆ sources, we define heap as the type addr ^→ option cell, the type of functions for which the functional extensionality axiom is admissible in F⋆; we gloss over this technicality in our presentation here.

[2] On universes and higher order stores: We define our memory model universe-polymorphically, so that it can store values in higher universes, e.g., values at existential types (§5.4.3). However, the cell type resides in a universe one greater than the type it contains. By extension, heap is in the same universe as its cells. As a result, although heaps and heap-manipulating total functions cannot be stored in cells, functions in F⋆ that include the effect of divergence are always in universe 0 and can be stored in the heap, i.e., this model is adequate for partial correctness of programs with higher order stores.

4.2 Separation Logic Propositions

We define the type slprop of separation logic propositions as the type of heap propositions p that are preserved under disjoint extension. We emphasize that slprops are affine *heap* propositions, rather than mem propositions—the non-heap fields in a memory (e.g., freshness counters etc.) are meant for internal bookkeeping and (intentionally) cannot be described by slprops. We use interp to apply an slprop to the heap within a memory. Further, being heap predicates, slprops reside in the same universe as heap. As such, slprops cannot themselves be stored in the heap, although doing so is sometimes convenient for encoding various forms of higher-order ghost state (Jung et al. 2016)—this is the main limitation of our model. However, since Steel is embedded within F*, one can sometimes work around this restriction by adopting various dependently typed programming tricks, e.g., rather than storing slprops in the heap, one might instead store codes for a suitably small sub-language of slprops instead and work with interpretations of those codes.

```
let slprop = p:(heap → prop) { ∀(h0 h1: heap). p h0 ∧ disjoint h0 h1 ⟹ p (join h0 h1) }
let interp (p:slprop) (m:mem) = p m.heap
```

We define several basic connectives for slprop, as shown below. The existential and universal quantifiers, slex and slall support quantification over terms in arbitrary universes, including quantification over slprops themselves.

```
let slstar p1 p2 h = ∃h1 h2. h1 `disjoint` h2 ∧ h == join h1 h2 ∧ p1 h1 ∧ p2 h2
let slwand p1 p2 h = ∀h1. h `disjoint` h1 ∧ p1 h1 ⟹ p2 (join h h1)
let slemp p h = ⊤            let sland p1 p2 h = p1 h ∧ p2 h      let slor p1 p2 h = p1 h ∨ p2 h
let slex p h = ∃x. p x h     let slall p h = ∀x. p x h
```

This interpretation also induces a natural equivalence relation on slprop, i.e., p ∼ q iff (∀m. interp p m ⟺ interp q m) (extensional equivalence of heap predicates) and it is easy to prove that star and emp form a (total) commutative monoid with respect to ∼.

We also define the atomic points-to assertion on references.

```
let ref (a:Type) (p:pcm a) = addr
let pts_to (r:ref a p) (v:a) (h:heap) = match h r with Some (Ref a' p' v') → a == a' ∧ p == p' ∧ v ≤p v' | _ → ⊥
val pts_to_compatible (r:ref a p) (v0 v1:a) (m:mem) : Lemma
    (interp (pts_to r v0 ∗ pts_to r v1) m ⟺ (p.composable v0 v1 ∧ interp (pts_to x (p.op v0 v1)) m))
```

A reference is represented by its address in the heap and pts_to r v asserts partial knowledge of the contents of the reference r, i.e., that r contains some value v' compatible with v according to the PCM associated with r. The pts_to_compatible lemma relates the separating conjunction to composition in the underlying PCM. In coming sections we will see how to choose specific PCMs to model fractional permissions and monotonic references.

We now have most of what we need to instantiate the state interface of our semantics—two key ingredients, the memory invariant and preorder will be presented in detail in the the next three subsections. Foreshadowing their presentation, our state instantiation is:

```
let st : state = { mem = mem; slprop = slprop; equals = ∼; emp = slemp; star = slstar; interp = interp;
                   inv = mem_inv (∗ cf. §4.3 ∗); evolves = mem_evolves (∗ cf. §4.4, 4.5 ∗) }
```

Actions on PCM-indexed references. Given this instantiation, one can define several basic actions (cf. §3.1), such as the following primitives on references. Building on these generic primitives, we implement libraries for several more common use cases, including references with fractional permissions and monotonic references.

To allocate a reference, one presents both a value and a PCM to use for that reference.

```
val alloc (#p:pcm a) (v:a) : action (ref a p) emp (λ r → pts_to r v)
```

Reading a reference with (!) returns a value compatible with the caller's partial knowledge.

```
val (!) (r:ref a p) (v:erased a) : action (x:a{v ≤_p x}) (pts_to r v) (λ v' → pts_to r v)
```

Mutating a reference r requires the new value v to be compatible with all frames compatible with the caller's partial knowledge of r.

```
let frame_preserving (#p:pcm a) (x y:a) = ∀f. p.composable f x ⟹ p.composable f y ∧ p.op f y == y
val (:=) (r:ref a p) (v0:erased a) (v:a{frame_preserving v0 v}) : action unit (pts_to r v0) (λ _ → pts_to r v)
```

Finally, to de-allocate a reference the caller must possess exclusive non-trivial knowledge of it.

```
let exclusive (#p:pcm a) (v:erased a) = ∀frame. p.composable frame v0 ⟹ frame==p.one
val free (r:ref a p) (v0:erased a{exclusive v0}): action unit (pts_to r v) (λ _ → emp)
```

In what follows, we overload the use of F*'s existing connectives $\exists, \forall, \land, \lor$ for use with slprop. We write emp for slemp; $*$ and $-\!\!*$ for slstar and slwand. Borrowing F*'s notation for refinement types, we also write h:p{f} for sland p (λh → f) and pure p for _:emp{p}.

4.3 Introducing Invariants: Preorders and the istore

Beyond the traditional separation logic assertions, it is useful to also support a notion of *invariant* that allows a non-duplicable slprop to be shared among multiple threads. For some basic intuition, it's instructive to look at the design of invariants in Iris—we reproduce, below, three of Jung et al.'s (2018) rules related to invariants (slightly simplified).

$$(1)\ P \Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}} \qquad\qquad (2)\ \text{persistent}\left(\boxed{P}^{\mathcal{N}}\right)$$

$$(3)\ \frac{\{\,\triangleright P * Q\,\}\ e\ \{\,\triangleright P * R\,\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \text{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{\boxed{P}^{\mathcal{N}} * Q\right\}\ e\ \left\{\boxed{P}^{\mathcal{N}} * R\right\}_{\mathcal{E}}}$$

The first rule states that at any point, one can turn a resource assertion P into an *invariant* $\boxed{P}^{\mathcal{N}}$. An invariant is associated with a name, $\mathcal{N}$—we shall see its significance in §4.4.

The second rule states that an invariant is persistent, which implies it is duplicable: i.e., $\boxed{P}^{\mathcal{N}} \implies \boxed{P}^{\mathcal{N}} * \boxed{P}^{\mathcal{N}}$. Thus, by turning a resource assertion P into an invariant, one can share the invariant among multiple threads, frame it across other computations, etc.

The final rule shows how an invariant can be used. This rule is quite technical, but intuitively it states that an atomic command e can assume the resource assertion P associated with an invariant $\boxed{P}^{\mathcal{N}}$, so long as it also restores P after executing (atomically). Some of the technicality in the rule has to do with impredicativity and step indexing. In Iris, $\boxed{P}^{\mathcal{N}}$ is a proposition in the logic like any other, and Iris allows quantification over all such propositions, including invariants themselves. This is very powerful, but it also necessitates the use of step indexing, i.e., the "later" modality $\triangleright P$ in the premise of the rule. For SteelCore, we seek to model invariants of a similar flavor, but while remaining in our predicative setting—our use of the monotonic state effect will give us a way.

Invariants in SteelCore. To allocate an invariant, we provide an action with the signature below:

```
val new_invariant (p:slprop) : action (ival p) p (λ _ → emp)
```

Recall the action type from §3.1. The type above states that given possession of p, new_invariant consumes p, providing only emp, but importantly, returning a value of type ival p: our representation of an invariant—new_invariant models Iris' update modality to allocate an invariant, i.e., the first of the three rules above. Being a value, ival p is freely duplicable, like any other value in F*—mimicking Iris' rule of persistence of invariants. Finally, sketching (imprecisely) what we develop in detail in §4.4, we provide a combinator below that is the analog of Iris' rule for eliminating and restoring invariants in atomic commands—an atomic command that expects p ∗ q and provides p ∗ r can be turned into a command that only expects q and provides r, as long as an ival p value can be presented as evidence that p is an invariant.

```
val with_invariant (i:ival p) (e:atomic a (p ∗ q) (p ∗ r)) : atomic a q r
```

Representing Invariants. We will use the istore component of a mem to keep track of invariants allocated with new_invariant: an istore is a list of slprops and the name associated with an invariant is its position in the list. The invariant of the istore (included in inv, which, recall from §3, is expected and preserved by every step of the semantics) requires every invariant in the istore to be satisfied separately. The i_evolves preorder (part of the mem_evolves preorder shown in §4.1) states that when the memory evolves, the istore only grows. The predicate inv_for_p i p m states that the invariant name i is associated with p in the memory m—its stable form, i⤳p, makes use of the witnessed connective used with the MST effect introduced in §2.2. i⤳p is the SteelCore equivalent of $\boxed{p}^{i}$, i.e., the name i is always associated with invariant p. Since i⤳p is just a prop, it is naturally duplicable. It's also convenient to treat invariants as a value type, ival p—just an invariant name i refined to be associated with p.

```
let istore = list slprop        let istore_inv (i:istore) : slprop = List.fold_right (∗) emp i        let inv_name = nat
let i_evolves is0 is1 = ∀(i:inv_name). List.nth i is0 == None ∨ List.nth i is0 == List.nth i is1
let inv_for_p (i:inv_name) (p:slprop) (m:mem) = Some p == List.nth i m.istore
let (⤳) i p = witnessed (inv_for_p i p)             let ival (p:slprop) = i:inv_name{i⤳p}
```

Now, to define the new_invariant p action we simply extend the istore, witness that p is now an invariant, and return the address of the newly allocated invariant.

```
let new_invariant (p:slprop) : action (ival p) p (λ _ → emp) = λ() →
  let m = get () in put ({m with istore=m.istore@[p]}); let i = List.length m.istore in witness (inv_for_p i p); i
```

With these definitions in place, we have all we need to instantiate the state interface of the semantics, using for each of its fields (mem, slprop, evolves etc.) the definitions shown here.

4.4 Using Invariants in Atomic Commands

We have seen how to allocate duplicable invariants, i.e., the analog of the first two rules for manipulating invariants in Iris. What remains is the third rule that allows invariants to be used in atomic commands.

For starters, this requires carving out a subset of computations that are deemed to be atomic, i.e., we need a way to express something like the premise atomic(e) from the Iris rule. However, observe that our semantics from §3 already provides a notion of atomicity: individual actions in Act nodes are run to completion without any interference from other threads. Specific actions in our memory model can be marked as atomic, depending on the particular architecture being modeled. For example, one might include a primitive, atomic compare-and-set action, while other primitive actions like reading, writing or allocating references may or may not be atomic, depending on the architecture being modeled. Further, some actions can be marked as *ghost* and sequences of such commands may also be considered atomic, since they are never actually executed concretely.

Next, we need a way to determine which invariants are currently "opened" by an atomic command. Recursively opening the same invariant ival p is clearly unsound since, although ival p is duplicable, p itself need not be.

Finally, to explain Iris's atomic actions rule in full, we also need to model the *later* modality ▹. As we will see, the witnessed modality provided by the monotonic state effect serves that purpose well.

The type of atomic actions. The type 'atomic a uses is_ghost p q' below is a refinement of the type of actions, action a p q presented in §3.1. The first additional index, uses, indicates the set of opened invariants—in particular, an atomic action can only assume and preserve the invariants not included in uses, as shown in the definition of istore_inv'. The second index, is_ghost, is a tag that indicates whether or not this command is a ghost action. The atomic type represents an effectful operation a total sub-effect NMSTTot of the effects of nondeterminism and monotonic state—by choosing a total sub-effect, we avoid pitfalls of infinitely opening invariants or introducing divergence in ghost computations. As such, due to the restriction to total computations, the type atomic a {} b p q is a subtype of the action a p q type defined in the semantics.

```
let istore_inv' uses ps = List.fold_right_i (λ p i q → if i ∈ uses then q else p ∗ q) ps emp
let inv' uses m = . . . m.ctr . . . ∧ istore_inv' uses m.istore
let atomic (a:Type) (uses:set inv_name) (is_ghost:bool) (p:slprop) (q: a → slprop) =
    unit → NMSTTot a mem mem_evolves
            (requires λm → interp (inv' uses m ∗ p) m)
            (ensures λm0 x m1 → interp (inv' uses m1 ∗ q x) m1 ∧ preserves_frame p (q x) m0 m1)
```

We treat the atomic type as a user-defined abstract effect in $F^\star$ and insist on at most one non-ghost action in a sequential composition, as shown by the signature of bind_atomic below.

```
val bind_atomic #a #b #u #p #q #r #g1 (#g2:bool{g1 || g2})
                    (e1:atomic a u g1 p q) (e2: (x:a → atomic b u g2 (q x) r)) : atomic b uses (g1 && g2) p r
```

Opening and closing an invariant. The final piece of the puzzle is the with_invariant construct, whose signature is shown below. Given an atomic command e that uses the invariant i:ival p to gain and restore p, it can be turned into an atomic command that no longer uses i, and whose use of p is no longer revealed in its specification.

```
val with_invariant #a #p #q #r #u #g (i:ival p) (e:atomic a (i ⊎ u) g (p ∗ q) (λ x → p ∗ r x)) : atomic a u g q r
```

Finally, given a value of type e:atomic a {} _ p q we can promote it to an e:action a p q (since the types are equivalent) and then turn it into a computation Act e : ctree a p q.

See ya, later. The with_invariant rule presented above does not have Iris' *later* modality, yet the later modality is essential for soundness in Iris and in other logics (Dodds et al. 2016) that support stored propositions. Paraphrasing Jung et al. (2018), a logic that supports allocating persistent propositions, together with a deduction rule for the injectivity of stored propositions of the form $i \rightsquigarrow p * i \rightsquigarrow q \vdash (p \Longleftrightarrow q)$ is inconsistent—the conclusion of the rule must be guarded under a later, i.e., it should be $\triangleright(p \Longleftrightarrow q)$. Although it may not be immediately evident, Ahman et al.'s (2018) model of monotonic state also has a "later" modality in disguise. In their model, witnessed $\bot \nvdash \bot$: instead, an explicit step of computation via the recall action is necessary to extract a contradiction from witnessed $\bot$. As such, $i \rightsquigarrow p * i \rightsquigarrow q \vdash (p \Longleftrightarrow q)$ is *not* derivable in SteelCore, although with a step of computation, the Hoare triple $\{i \rightsquigarrow p * i \rightsquigarrow q\}$ recall i $\{ p \Longleftrightarrow q \}$ is derivable. In summary, the effect of monotonic state provides a way to account for the necessary step indexing without making it explicit in the logic.

The update modality and ghost actions. As a final remark, allocating an invariant in Iris is done using its *update* modality, $\Rrightarrow_{\mathcal{E}}$. Besides allocating invariants, updates in Iris are also used to transform ghost state. In SteelCore, rather than including such a modality within the logic, we rely on $F^\star$'s existing support for erased types to model ghost state and updates within Hoare triples, rather than within the logic itself.[3] For instance, the following action represents a ghost read: it dereferences x, returning its contents only as an erased a.

```
val ghost_read (x:ref a p) : atomic (erased a) u true (∃ v. pts_to r v) (λ v → pts_to r v)
```

4.5 Fractional Permissions and Monotonic References

Several prior works have provided PCM-based constructions both to capture various sharing idioms as well as to define state machines that constrain how the state is permitted to evolve. In this section, we show how to use PCMs to encode Ahman et al.'s (2018) preorder-indexed monotonic references. We start, however, with a simpler construction of references with fractional permissions, a construction we reuse for monotonic references.

References with fractional permissions. To model references to t-typed values with fractional permissions we store at each cell a value of type frac t = option (t & r:real{0.0 < r}) with pcm_frac as shown below—the composable predicate allows us to use undecidable relations like propositional equality in our notion of partiality.

```
let pcm_frac : pcm (frac t) = { one = None;
  composable = (λ f0 f1 → match f0, f1 with | Some (v0, r0), Some(v1, r1) → v0==v1 ∧ r0+r1 ≤ 1.0 | _ → ⊤);
  op = (λ f0 f1 → match f0, f1 with | None, f | f, None → f | Some (v, r0), Some(_, r1) → Some(v, r0 + r1)) }
```

Specializing the type of references and the points-to assertion for use with pcm_frac, we recover the traditional injective points-to assertion on references and a lemma that relates the separating conjunction in slprop to composition in pcm_frac.

```
let ref t = ref (frac t) pcm_frac
let (↦_f) r v = pts_to r (Some (v, f)) ∗ pure (f ≤ 1.0)
val share_gather (r:ref t) (f g:real) (u v:t) : Lemma (r ↦_f u ∗ r ↦_g v) ∼ r ↦_{f+g} u ∗ pure (u==v))
```

Monotonic references. Whereas we have used preorders and monotonic state within our memory model to support the dynamic allocation of invariants, here we aim to expose preorders to describe state transitions on individual references, in the style of Ahman et al.'s monotonic references. Pleasantly, we find that our PCM-based memory model layered above the monotonic state effect can precisely capture Ahman et al.'s construction in a generic manner.

Our goal is to provide the following interface on an abstract type mref a p of references indexed by a preorder. The main point of interest is the signature of write, which requires proving that the new value v is related to the old value by the preorder p.

```
val mref (a:Type) (p:preorder a) : Type
val (⟼_f) (x:mref a p) (v:a) : slprop
val read (r:mref a p) (v0:erased a) : action a (r ⟼_f v0) (λ v → r ⟼_f v)
val write (r:mref a p) (v0:erased a) (v:a{p v0 v}) : action unit (r ⟼_1.0 v0) (λ _ → r ⟼_1.0 v)
val observed (r:mref a p) (q:a → prop) : prop
val witness_mref (r:mref a p) (q:stable_prop p) (v:erased a{q v})
    : action unit (pts_to r f v) (λ _ → pts_to r f v ∗ pure (observed r q))
val recall_mref (r:mref a p) (q:stable_prop a p) (v:erased a)
    : action unit (pts_to r f v ∗ pure (observed r q) (λ _ → pts_to r f v ∗ pure (q v))
```

[3] Iris also internalizes Hoare triples, but in SteelCore, we rely on the computation types of the host language to express Hoare triples outside the logic.

In return for respecting the preorder at each update, we provide two new operations to witness and recall properties that are invariant under the preorder. The operation witness returns a pure, abstract predicate observed r q when the current value of r satisfies a stable property q; and recall eliminates observed r q into q v, for v the current value of r. These operations are the analogue of the MST actions witness and recall exposed to SteelCore programs at the granularity of a single reference, rather than the entire state. For instance, one could define a monotonically increasing counter as r:mref int (≤), and having observed that r contains the value 17 one can recall later that r's value is at least 17.

From PCMs to preorders. We observe that every PCM induces a preorder and, dually, every preorder can be encoded as a PCM. To interpret a PCM as a preorder, we take the infinite conjunction of all preorders refined by the frame_preserving relation: in other words, since all updates must be frame-preserving, we take the preorder of a PCM to be the strongest preorder entailed by frame-preservation.

```
let induces (p:pcm a) (q:preorder a) = ∀(x y:a). frame_preserving p x y ⟹ (∀ (z:a). p.compatible x z ⟹ q z y)
let preorder_of_pcm (#a: Type u#a) (p:pcm a) : preorder a = λx y → ∀q. p `induces` q ⟹ q x y
```

With this notion in hand, we can finally define the heap evolution relation (part of the global memory preorder shown in §4.1) stating that (1) unused heap cells can change arbitrarily; (2) used heap cells remain used; and, most importantly, (3), the type and PCM associated with a ref cell does not change and its value evolves according to the preorder of the PCM. In other words, heaps evolve by the pointwise conjunction of the PCMs at each ref cell.

```
let h_evolves h0 h1 = ∀(a:addr). match h0 a, h1 a with | None, _ → ⊤| Some _, None → ⊥
                | Some (Ref a0 p0 v0), Some (Ref a1 p1 v1) → a0 == a1 ∧ p0 == p1 ∧ preorder_of_pcm p0 v0 v1
```

From preorders to PCMs. Conversely, to interpret a preorder q:preorder a as a PCM, we define a PCM over hist q, the type of histories over a, sequences of a-values where adjacent values are related by q, with composability demanding one history to be an extension of the other; composition being history extension; and the unit being the empty history. The full construction is available in our online code repository—we show its main signature below, including a round-trip property, showing that the PCM built by the construction induces the preorder corresponding to extension of q-respecting histories.

```
val pcm_of_preorder (q:preorder a) : p:pcm (hist q) {p `induces` history_extension}
```

This construction enables constructing a PCM frac_hist q to support the type mref a q, combining fractional permissions with the hist q PCM, with the property that for any property f:a → prop stable with respect to q, its lifting lift f : hist q → prop (that applies to the most recent value in a history) is stable with respect to preorder_of_pcm (frac_hist q). As such, the underlying witness and recall operations of the monotonic-state effect suffice to provide a model for witness_mref and recall_mref. In §5.4 we use mrefs to encode a trace of messages exchanged on a channel, proving that those traces respect a preorder induced by a user-provided state machine.

5 STEELCORE AT WORK: LOCKS, FORK/JOIN, CHANNELS, TRACES

In this section, we make use of the SteelCore program logic to build a few libraries of verified synchronization primitives. We start with a spin lock, built using invariants accessed by an atomic CAS instruction. Using a spin lock, we build a library for fork/join concurrency on top of structural parallelism (par) and general recursion. Finally, we present a library of synchronous, simplex channels whose use is controlled by a specification-level state machine. All of our examples are programmed directly within F*, making use of all its abstraction and specification features, including mixing dependently typed specifications and effects with SteelCore's CSL. That said, while our proofs already rely on both SMT solving and dependent typechecking, they are still quite manual. As such, the code in this section is simplified to make it easier to explain: the full implementation together with the proofs are much more verbose and are in the supplement. Better proof automation is left as future work.

5.1 Example: Spin Locks

We illustrate how to use invariants with atomic commands to build a spin lock, building on a primitive compare-and-set atomic action with the signature shown below. It states that given a (fractional permission) reference r to a word-sized integer for which we have full permission, and old and new values, cas updates the reference to new if its current value is old and otherwise leaves r unchanged. Importantly, cas is parametric in the set of opened invariants u. Note, cas takes an additional ghost parameter, v:erased uint32, which represents the value stored in the reference in the initial state.

```
val cas (#u:set inv_name) (r:ref uint32) (old new:uint32) (v:erased uint32) :
    atomic (b:bool{b=(v=old)}) u false (r ↦1.0 v) (λ b → r ↦1.0 (if b then new else v))
```

A lock is represented as a pair of a reference and an invariant stating that the reference is in one of two states: either it holds the value available and the lock invariant p, an slprop, is true separately; or it holds the value locked.

```
let available, locked = false, true
let lockinv (r:ref bool) (p:slprop) = (pts_to r 1.0 available ∗ p) ∨ (pts_to r 1.0 locked)
let lock_t = ref bool & inv_name        let protects (l:lock_t) (p:slprop) : prop = snd l ⇝lockinv (fst l) p
let lock p = l:lock_t { l `protects` p }
```

For convenience, similarly to the NST effect in §2.1, we package the ctree trees into a Steel effect having the form Steel a fp0 fp1. Using this Steel effect, allocating a lock is straightforward:

```
let new_lock p : Steel (lock p) p (λ _ → emp) =
    let r = alloc available in let i = new_invariant (lockinv r p) in (| r, i |)
```

Releasing a lock requires opening the invariant to gain permission to the reference—we add comments to the code to show the relevant Hoare triples in the term using the notation {p} e {q}. Within the invariant, we use a ghost read to fetch the current value of the reference, then do a cas and can prove that it sets the reference to available. In the case where the reference was already set to available, we use the affinity of our separation logic to forget the assertion (b=false wand p) before closing the invariant. We could also return the resulting boolean to avoid losing information.

```
let release ((| r, i |):lock p) : Steel unit p (λ _ → emp) =
    let _ = with_invariant i
(∗ {lockinv r p ∗ p} ∗)
    (∗ {((pts_to r 1.0 available ∗ p) ∨ pts_to r 1.0 locked) ∗ p} ∗)
        (let v = ghost_read r in cas r locked available v)
    (∗ {λ b → pts_to r 1.0 available ∗ (b=false wand p) ∗ p} ∗)
(∗ {lockinv r p ∗ emp} ∗) in ()
```

Acquiring a lock is similar to releasing it: We try to set the lock reference to locked within the invariant using an atomic cas. If cas fails, we "spin" by repeatedly calling acquire until the lock becomes available. The function terminates once the reference has been set to locked and we successfully acquired the corresponding slprop.

```
let rec acquire ((| r, i |):lock p) : Steel unit emp (λ _ → p) =
  let b = with_invariant i
    (* {lockinv r p} *) (let v = ghost_read r in cas r available locked v) (* {λ b → lockinv r p ∗ (b=true wand p)} *)
  in if b then () else acquire (| r, i |)
```

Although SteelCore's logic is predicative, since the host language supports abstraction in arbitrary universes, we can build libraries whose specifications are generic in separation logic assertions—this allows us to use our spin lock library to protect any slprop.

5.2 Fork/Join

SteelCore's only concurrency primitive is the Par combinator for structured parallelism shown in Figure 2, that we expose as a stateful par in the Steel effect. However, having just built a library for locks, we can code up a library for fork/join concurrency without too much trouble. As with locks, since the host language is higher order, we can easily abstract over computations and their specifications, although Hoare triples are not part of SteelCore's logic itself.

The interface we provide for forking and joining threads is shown below. The type thread p represents a handle to a thread which guarantees p upon termination. The combinator fork f g runs the thread f and continues with g in parallel, passing to g a handle to the thread running f. The join t combinator waits until the thread t completes and guarantees its postcondition.

```
val thread (p:slprop) : Type
val fork #p #q #r #s (f: (unit → Steel unit p (λ _ → q))) (g: (thread q → Steel unit r (λ _ → s)))
  : Steel unit (p ∗ r) (λ _ → s)
val join #p (t:thread p) : Steel unit emp (λ _ → p)
```

To implement this interface, we represent a thread handle as a boolean reference protected by a lock that guarantees the thread's postcondition p when the reference is set. Allocating a thread handle is easy, since the reference can initially be set to false.

```
let thread p = { r:ref bool; l:lock (∃ b. pts_to r 1.0 b ∗ (if b then p else emp))}
val new_thread (p:slprop) : Steel (thread p) emp (λ _ → emp)
```

To fork a thread, we create a new thread handle t, then in parallel, run g t and in the thread for f, we acquire the lock, run f(); then set the reference and release the lock.

```
let fork #p #q #r #s f g =
  let t = new_thread q in let _ = par (λ _ → acquire t.l; f(); t.r := true; release t.l) (λ _ → g t) in ()
```

Finally, to join we repeatedly acquire the lock, and if the reference is set, we can free the reference and return the postcondition p; otherwise we release the lock and loop—F*'s existing support for general recursion makes it relatively easy.

```
let rec join #p (t:thread p) = acquire t.l; let b = !t.r in if b then free t.r else (release t.l; join t)
```

Note, to provide a C-style fork/join on top of our API requires a CPS-like transform, since fork expects separate continuations for the parent and child threads. We hope to address the usability of fork in the future, perhaps layering another effect for continuations above the Steel effect to support fork/join in direct style.

5.3 Local State and Lock-Coupling Lists: Higher Order Assertions and Invariants

Being embedded in a dependent type theory allows Steel programs to enjoy all the abstraction facilities of the host language. In this section, we provide two classic examples further illustrating the abstraction facilities available, while pointing out some limitations.

Counters with local state. The type ctr_t below represents a counter as a closure paired with an abstract invariant p over its local state. The invariant is indexed by the current value of the counter, and each application of the closure expects the invariant, returns a value one greater than its previous index, and restores the invariant at the returned value. To allocate a new counter, new_ctr returns a ctr_t and provides the initial invariant at the index 0. As mentioned in §4.2, since slprops cannot be stored in the heap, a ctr_t cannot be stored in a ref cell.

```
let ctr_t = (p:(int → slprop) & (x:erased int → Steel (y:int{y==x+1}) (p x) p))
val new_ctr (_:unit) : Steel ctr_t emp (λ (| p, _|) → p 0)
```

Lock-coupling lists. Spatial assertions and invariants can be defined by recursion too, e.g., to describe the representation invariant of a linked list, each of whose nodes is protected by its own lock, a so-called lock-coupling list. A challenge here is to support locks that can be dynamically allocated and stored in the heap in alongside each node that it protects, but this is easily expressed in our system, since invariants and locks (on which they are based), are dynamically allocated and, unlike slprop, are storable. Proceeding along the lines of Gotsman et al. (2007), we show the representation invariant for a lock-coupling list below. The predicate llist_inv repr n grants ownership to the head of the list (if any); whose value v validates p; and states that the lock stored at the head recursively grants the representation predicate for the tail of the list.

```
type llist (a:Type0) : Type0 = { v : a; next : ref (llist a); lock : lock_t }
let rec llist_inv (repr:list (a → slprop)) (n:ref (llist a)) =
  match repr with | [] → emp | p::tl → ∃c. p c ∗ n ↦1.0 c ∗ pure (c.lock `protects` llist_inv tl c.next)
```

5.4 Channel Types: From Indexed Action Trees to Action Tree Indexes

As a final example, we present a library for synchronous communication among threads. We draw on inspiration from the long line of work on session types (Honda et al. 1998) to enforce a typing discipline that associates with each channel a state machine that describes the sequence of permissible operations on that channel. For this paper, we focus only on the simplest scenario of synchronous simplex channels (channels with unidirectional communication)—towards the end, we remark on how to generalize this construction to 2-party session types and duplex channels. As such, our work shows how to use SteelCore as a platform on which to model higher-level constructs for reasoning about concurrent and distributed programs, while mixing various concurrency idioms (e.g., channels, locks and atomics).

Our model of channels proceeds in three steps. First, we define a small language for describing protocols as action trees. Next, we define a notion of partial traces of protocols, sequences of messages that are accepted by the protocol state machine. And, finally, we define the type of channels indexed by protocols with an interface that supports sending and receiving messages on channels, together with an internalized proof (done once and for all protocol-indexed channels) that the sequence of messages received on a channel are a partial trace of the protocol.

5.4.1 Protocols as Action Trees. The type protocol below expresses a small embedded language to express the sequence of messages that can be sent on a channel—the erasable annotation causes $F^\star$ to check that the type is never used in a computationally relevant context and all protocol values are erased to () during extraction.

```
[@erasable] type protocol : Type → Type =
| Ret : #a:Type → v:a → protocol a
| Msg : a:Type → #b:Type → k:(a → protocol b) → protocol b
| DoWhile : p:protocol bool{p≠Ret _} → #a:Type → k:protocol a → protocol a
```

The type has the classic structure of an infinitely branching tree of actions similar to the one described in §2. For example, the term below describes a small two message protocol, where the first message is an integer x and the second message y is an integer one greater than the first message, i.e., the continuations of a message depend on the values exchanged in the history of the protocol.

```
let xy = Msg int (λ x → Msg (y:int{y = x + 1}) (λ y → Ret ()))
```

It should be straightforward to see that protocol is a monad, with an easily definable bind. The DoWhile construct allows specifying infinite protocols. For example, DoWhile (xy `bind`(λ_ → Ret true)) (Ret ()) is a channel on which one can repeatedly send related pairs of successive integers.

5.4.2 Traces of a Protocol. One may wonder how we give semantics to infinite reductions with DoWhile: as it turns out, we only consider finite partial traces defined as the reflexive, transitive closure of a single step relation, as we show next.

The function hnf below puts a protocol into a form that begins with either a Ret (in case the protocol has finished) or a Msg node, indicating the next action to be performed.

```
let rec hnf (p:protocol a): (q:protocol a{(Ret? q ∨ Msg? q) ∧ (¬(DoWhile? p) ⟹ (p == q))})
  = match p with | DoWhile p k → bind (hnf p) (λ b → if b then DoWhile p k else k) | _ → p
```

Using it, we can define a notion of a single step of reduction of a protocol: if a protocol p has more actions, then given a message x whose type matches the type of the next action, p steps according to its continuation.

```
let more (p:protocol a) : bool = Msg? (hnf p)
let msg_t (p:protocol a) : Type = match hnf p with | Msg a _ → a | Ret #a _ → a
let step (p:protocol a{more p}) (x:msg_t p) : protocol a = Msg?.k (hnf p) x
```

The type of traces below, trace from to, represents a sequence of messages that are related by stepping the protocol from until the protocol to, and trace_of p is the type of traces accepted by zero or more steps of p.

```
let prot = protocol unit
[@erasable] type trace : prot → prot → Type = | Waiting : p:prot → trace p p
  | Message : from:prot{more from} → x:msg_t from → to:prot → trace (step from x) to → trace from to
type trace_of p = { until:prot; tr:trace p until }
```

It's easy to extend a trace by one message; then to define a relation next on t; and finally to define trace extension as the closure of next.

```
val extend₁ (#from #to:protocol unit) (t:trace from to{more to}) (m:msg_t to) : trace from (step to m)
let next p t0 t1 = more t0.to ∧ (∃ msg. t1.to == step t0.to msg ∧ t1.tr == extend₁ t0.tr msg)
let (↪) (#p:protocol unit) : preorder (trace_of p) = ReflexiveTransitiveClosure.closure (next p)
```

We will use the preorder ↪ to maintain a ghost monotonic reference storing a log of messages associated with a channel and to prove that the trace of messages on a channel are always accepted by that channel's protocol.

5.4.3 Channel Types. We aim to provide the following (hopefully idiomatic) interface to work with channels. The abstract type chan i is the type of channels created for use with the *initial protocol* i. We have two abstract predicates, sender and receiver, both indexed by a protocol that describes the *current state* of the channel from the sender's and receiver's perspective, respectively

```
val chan (i:prot) : Type
val sender #i (c:chan i) (cur:prot) : slprop
val receiver #i (c:chan i) (cur:prot) : slprop
```

To create a channel, we use new_chan i, we return a new chan and the sender's and receiver's state separately initialized to the given initial protocol i.

```
val new_chan (i:prot) : Steel (chan i) emp (λ c → sender c i ∗ receiver c i)
```

To send a message, one presents a channel c in a state cur where more messages are expected; a value x, whose type is the type of the next message in the protocol: as a result, the sender's state transitions by a single step, which depends on the value x provided. The recv is dual to the send.

```
val send #i (#cur:prot{more cur}) (c:chan i) (x:msg_t cur) : Steel unit (sender c cur) (λ _ → sender c (step cur x))
val recv #i (#cur:prot{more cur}) (c:chan i) : Steel (msg_t cur) (receiver c cur) (λ x → receiver c (step cur x))
```

In addition, we provide further operations that internalize the guarantee that the trace of messages on a channel respects its protocol. The duplicable abstract predicate history c t states that t is a partial trace of messages received on c. The operation trace allows a client to extract the current trace. Most importantly, extend_trace ensures that traces are monotonic: if history c p witnesses that p was a trace of c, then if the receiver's current state is cur, one can prove that the current trace t of the protocol is an extension of p until cur. That is, all well-typed channel programs respect the channel's protocol.

```
val history #i (c:chan i) (t:trace_of i) : slprop
val history_duplicable #i (c:chan i) (t:trace_of i) : Steel unit (history c t) (λ _ → history c t ∗ history c t)
val trace #i (c:chan i) : Steel (trace_of i) emp (λ tr → history c tr)
val extend_trace #i (#cur:prot) (c:chan i) (p:trace_of i): Steel (t:trace_of i{p `extended_to` t})
            (receiver c cur ∗ history c p) (λ t → receiver c cur ∗ history c t ∗ until t == cur)
```

Implementing this interface will take a few steps, and will exercise nearly all elements of SteelCore presented so far, including fractional permissions, ghost state, locks, and monotonic references.

Representing channels. We'll represent channels by a pair of concrete references, each writable by only one side of the channel, though readable by both (via a lock), and a ghost reference maintaining a trace of the protocol. The concrete references contain a triple: an erased field prot, which we'll use to state our invariants; the last value sent or received on the channel (respectively); and a counter nat which counts the number of messages sent or received so far, which we'll use to determine if a message is available to be received or not.

```
type chan_val = { prot : prot{more prot}; msg : msg_t prot; ctr : nat}
type chan_t i = { send : ref chan_val; recv : ref chan_val; trace : mref (trace_of i) (↪) }
```

The main invariant chan_inv retains half a permission on the concrete references and full permission on the trace. It states that the trace is a partial trace of i until the state of the protocol on the receiver's side. Finally, it states that either the last sent message has already been received, in which case, the contents of the two references agree. Or, the sender is exactly one step of the protocol ahead of the receiving reference—its counter is one greater, and its protocol is a single step ahead of the receiver.

```
let chan_inv #i (c:chan_t i) = ∃vs vr tr. pts_to c.send 0.5 vs ∗ pts_to c.recv 0.5 vr ∗ pts_to c.trace 1.0 tr ∗
  (tr.until == step vr.prot vr.msg) ∗
  (if vs.chan_ctr = vr.chan_ctr then vs==vr else vs.ctr==vr.ctr+1 ∧ vs.prot == step vr.prot vr.msg )
```

Our channel type packages the two references with a lock that protects the chan_inv invariant. This makes channels fully first-class: channels can be stored in the memory and channels can even be passed on channels.

```
let chan i = { chan : chan_t i; lock : lock (chan_inv chan) }
```

The sender c p predicate is a permission to transition the protocol by one step to state p: it retains half a permission to the c.chan.send reference, together with an assertion that p is exactly the successor state of the protocol stored in the reference. The receiver predicate is similar. In both cases, by retaining half a permission to the reference, acquiring the chan_inv lock gives a full permission to the reference in question allowing, but only allowing the other reference to be read.

```
let sender (c:chan) (p:prot) = ∃vs. p == step vs.prot vs.msg ∧ pts_to c.chan.send 0.5 vs
let receiver (c:chan) (p:prot) = ∃vr. p == step vr.prot vr.msg ∧ pts_to c.chan.recv 0.5 vr
```

Implementing channels. With these invariants in place, the implementation is nearly determined. For space reasons, we only sketch the implementations of recv and extend_trace.

Receiving a message involves acquiring the lock, reading both references, and if a message is not available, releasing the lock and looping; otherwise, we update the receiver's state, extending their trace (ghostly), releasing the lock and returning the received message.

```
let rec recv #i #cur c = acquire c.lock; let vs, vr = !c.chan.send, !c.chan.recv in
    if vs.ctr=vr.ctr then release c.lock; recv c
    else c.chan.recv := vs; c.chan.trace := extend1 c.chan.trace vs.msg; release c.lock; vs.msg
```

Witnessing the monotonicity of the trace makes use of the monotonic references from §4.5. We define the history predicate in terms of the observations on monotonic references. Extending a trace then involves acquiring the lock, reading the trace, recalling the prior observation to learn that the current trace is an extension of the previous one, then making another observation using witness_mref; and finally releasing the lock and returning.

```
let history c tr = observed c.chan.trace (tr ↪ _)
let extend_trace #i #cur c prev = acquire c.lock; let tr = !c.chan.trace in recall_mref c (prev ↪ _);
                                  witness_mref c.chan.trace (tr ↪ _) tr; release c.lock; tr
```

2-party sessions. Channel types are already a useful abstraction, but they would be even more so when generalized to support duplex, asynchronous channels. We do not foresee any major difficulties in doing so. To support asynchrony, rather than holding just a single message in the sender's reference, we can buffer messages in the sender's state and the receiver can dequeue them. To support duplex channels, we anticipate extending the language of protocols to support directed message actions between principals and to derive mutually dual protocols for each participant by inverting the polarities of each messaging action. We leave both of these extensions to future work.

Discussion. In a sense, we've come full circle: we started in §3 by representing infinite concurrent computations as indexed effectful action trees. Now, we specify concurrent programs using indexed types, where the indexes themselves are action trees with infinite traces, with a proof within SteelCore that the message traces are partial traces of the index state machines. To prove safety properties of concurrent, channel-using programs, one can reason by induction on the partial traces. Alternatively, rather than reason directly on traces, one might even replay the methodology of this paper "one level up" and derive a program logic to reason about action tree indexes.

6 RELATED WORK

Throughout the paper, we have discussed connections to many strands of related work on CSL. In particular, we have drawn inspiration from, and contrasted our work with, Iris (Jung et al. 2018). The most significant point of contrast with Iris, perhaps, is our differing goals. Iris is a powerful impredicative logical framework into which other logics and programming languages can be embedded and studied. This has allowed researchers to use Iris as a foundation on which to investigate languages and language features as different as unsafe blocks in Rust (Jung et al. 2017) and state-hiding via rank-2 polymorphism in Haskell (Timany et al. 2018). In contrast, with SteelCore, we aim not to provide a general logical framework but instead to extend a proof assistant's programming language with an effect for concurrency and to reason about effectful, dependently typed concurrent programs in a CSL. This allows us to keep the embedded logic relatively simple: unlike Iris, it does not internalize Hoare triples, does not support storing slprops, and does not make use of step indexing (since our model internalizes monotonicity) or any of Iris' several modalities. However, many lacking features in the logic are recovered using the facilities of the host language, $F^\star$. E.g., we support rules for manipulating dynamically created, named invariants $i \rightsquigarrow p$ in a style inspired by Iris, using the underlying effect of monotonic state. Further, the logic supports quantification over slprops while remaining in $F^\star$'s predicative universe hierarchy, and also supports abstraction at the level of $F^\star$'s programming language (§5), e.g., we can abstract over slprops, computations with Hoare triples, etc.

In developing a shallow embedding of CSL in a dependent type theory, SteelCore is similar to FCSL (Nanevski et al. 2014, 2019; Sergey et al. 2015). FCSL is shallowly embedded in Coq and relies on Coq's abstraction facilities for some of its expressive power. Their logic (like ours and unlike Iris') applies directly to Coq programs, rather than to embedded programs. FCSL's semantic model is also similar to ours, in that they also represent computations as action trees. However, rather than using indexed action trees and directly interpreting the trees as the proof rules of a Hoare logic, Nanevski et al. (like Brookes's (2004) original proof of soundness of CSL) instead go via the indirection of action traces. In principle, it might be possible to define something like FCSL's logic using indexed action trees and to interpret those trees directly into the Hoare Type Theory underlying FCSL. However, modeling partiality in this way within Coq may be difficult, if not impossible—the action traces approach provides a way around this. In contrast, working in $F^\star$'s effectful type theory, we can directly model partiality and avoid the indirection of traces. FCSL's action traces also resemble the trace semantics we developed for our action trace indexes for channel types (§5.4.3). Another notable point of distinction with FCSL is SteelCore's treatment of invariants. Since FCSL's model is predicative, it does not provide a way to dynamically allocate an invariant, making it impossible to model certain kinds of synchronization primitives, e.g., our generic interface for locks does not seem to be expressible in FCSL. On the other hand, FCSL provides several constructs for reasoning about concurrent programs mixing styles of reasoning from CSL with rely-guarantee reasoning, something which we haven't explored much: our use of monotonic references may play a role in this direction, particularly in connection with other related work on rely-guarantee references (Gordon et al. 2013).

Many researchers have explored using action trees in modeling effectful constructs, including building indexed representations and interpreting them into effectful computations (Brady 2013; McBride 2011). Allowing effects in constructors of inductive types has been separately studied for simple, non-indexed types by Filinski and Støvring (2007) and Atkey and Johann (2015). We are the first to consider indexed effectful action trees that mix data and effectful computations, while interpreting the trees into another indexed effect, allowing us to layer effects—in our case, layering concurrency over divergence, monotonic state, and nondeterminism—while also deriving a program logic to reason about the new effect layer, based on the indexing structure. Our action trees from Section 3 have one parameter (the state type class) and four logical specification (including implicit dynamic frames) indexes in addition to the result type. This layering of effects also allows us to support infinite computations, without needing coinduction. In contrast, Xia et al. (2019) build coinductive, non-indexed action trees and give them an extrinsic, equational semantics. It would be interesting to study whether our style of intrinsically defined program logics can be developed using indexed versions of Xia et al.'s coinductive action trees.

We prove the soundness of our semantics by building an intrinsically typed definitional interpreter for our indexed effectful action trees. Intrinsically typed definitional interpreters have been investigated before by Bach Poulsen et al. (2018), who give several instances for languages ranging from the simply typed lambda calculus to middleweight Java embedded in Agda. The typing guarantees from these definitional interpreters ensure syntactic type safety of reduction with respect to the classic type systems for the languages they study. More recently, Rouvoet et al. (2020) give an intrinsically typed definitional interpreter for a linearly typed language, which bears some resemblance (owing to its linearity) to the structure of our interpreter for CSL. Our approach is similar to both these works in spirit, with two notable differences. First, rather than defining interpreters for deeply embedded languages, our representation allows the interpretation of host-language terms in an extended effectful semantics including concurrency (following the free monads methodology of Swierstra (2008) and others). Second, rather than proving syntactic type safety for embedded programs, we derive the soundness of a generic concurrent separation logic applied to host programs.

Embedding session types in CSL has also been investigated before—the Actris system embedded in Iris explores this in depth (Hinrichsen et al. 2019). Our channel types explore a similar direction, though we only scratch the surface, in that as a proof concept of the expressiveness of the underlying logic, we only give an encoding for synchronous simplex channels. One technical difference is that we embrace the use of state transition systems structured as action trees and prove trace inclusion within the system once and for all. Actris instead provides impredicative dependent separation protocols, which due to the use of higher-order ghost state that can depend on the type of propositions, appear to be strictly more expressive than our predicative state machines. Nevertheless, both systems can describe data-dependent protocols. Prior work on F* (Swamy et al. 2011a), which then included support for affine types, also developed a model for data-dependent affinely typed sessions for sequential, pure F* programs. Nearly a decade later, we show how to encode channel types for concurrent, stateful F* programs with CSL.

7 CONCLUSION

We have demonstrated how a full-fledged CSL can be embedded in an effectful dependent type theory, relying on an underlying semantics of monotonic state to model features that have otherwise required impredicative logics. In doing so, we have brought together two strands of work pioneered by John Reynolds: definitional interpreters and separation logic—we hope that he would at least have been intrigued by our work. Going forward, we plan to make use of SteelCore as the foundation of a higher level DSL embedded in F*, aiming to use a combination of tactics for manipulating slprops and SMT solving for implicit dynamic frames to help ease proofs of concurrent programs.

ACKNOWLEDGMENTS

Aymeric Fromherz's and Denis Merigoux's work was supported in part by internships at Microsoft Research. Aymeric Fromherz was also funded by the Department of the Navy, Office of Naval Research under Grant no. N00014-18-1-2892. Denis Merigoux was also funded by ERC Consolidator Grant CIRCUS no. 683032. Danel Ahman's work was supported in part by the Microsoft Research Visiting Researcher program. He has also received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no. 834146. We thank Robbert Krebbers, the shepherd of this paper; the anonymous reviewers; Matt Parkinson, Derek Dreyer, Ralf Jung, Aleks Nanevksi and all the members of Project Everest for their feedback, answering questions and many useful discussions.

REFERENCES

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

R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19:335–376, 2009.

R. Atkey and P. Johann. Interleaving data and effects. *Journal of Functional Programming*, 25, 2015.

C. Bach Poulsen, A. Rouvoet, A. Tolmach, R. Krebbers, and E. Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL), 2018.

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

E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. 2013.

S. Brookes. A semantics for concurrent separation logic. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory*. 2004.

A. Buisse, L. Birkedal, and K. Støvring. Step-indexed kripke model of separation logic for storable locks. *Electronic Notes in Theoretical Computer Science*, 276:121 – 143, 2011. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019.

T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2013.

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

A. Filinski and K. Støvring. Inductive reasoning about effectful data types. In R. Hinze and N. Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*. 2007.

C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2013.

A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*. 2007.

P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. G. Clote and H. Schwichtenberg, editors, *Computer Science Logic*. 2000.

J. K. Hinrichsen, J. Bengtson, and R. Krebbers. Actris: Session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), 2019.

A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In S. Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 2008.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Programming Languages and Systems*. 1998.

J. B. Jensen and L. Birkedal. Fictional separation logic. In H. Seidl, editor, *Programming Languages and Systems*. 2012.

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

R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), 2017.

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

O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. 2015.

R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017.

M. Krogh-Jespersen, A. Timany, M. E. Ohlenbusch, S. O. Gregersen, and L. Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. *Submitted for publication*, 2019.

G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hriţcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. *ESOP*. 2019.

C. McBride. Kleisli arrows of outrageous fortune, 2011. Unpublished draft.

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

A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. *POPL*. 2010.

A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. 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*, 2014.

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

P. W. O'Hearn. Resources, concurrency and local reasoning. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory*. 2004.

M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.

M. Piróg, T. Schrijvers, N. Wu, and M. Jaskelioff. Syntax and semantics for operations with scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*. 2018.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. 2002.

A. Rouvoet, C. B. Poulsen, R. Krebbers, and E. Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In J. Blanchette and C. Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*. 2020.

I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015.

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

N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *ICFP*. 2011a.

N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. *ICFP*, 2011b.

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

W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

The Coq development team. *The Coq proof assistant*.

A. Timany, L. Stefanesco, M. Krogh-Jespersen, and L. Birkedal. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runst. *PACMPL*, 2(POPL):64:1–64:28, 2018.

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