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\textsuperscript{*} proof assistant in a manner that enables dependently typed, effectful F\textsuperscript{*} 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.

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.

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 affordances 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.

This work is licensed under a Creative Commons Attribution 4.0 International License
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/1-ART1
https://doi.org/
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\(^{\star}\) 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\(^{\star}\), we develop a semantics for concurrent F\(^{\star}\) programs while simultaneously deriving a CSL to reason about F\(^{\star}\) 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\(^{\star}\) programs and prove their partial correctness using SteelCore.

### 1.1 SteelCore: A Concurrent Separation Logic Embedded in F\(^{\star}\)

SteelCore is the core semantics of Steel, a DSL under development in F\(^{\star}\) 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\(^{\star}\) 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
An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs

In 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. 2017) 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.

\[
\text{type } \text{ctree} (\text{st}: \text{st:state}) : \text{a:Type } \rightarrow \text{pre:st:slprop } \rightarrow \text{post:(a } \rightarrow \text{st:slprop)} \rightarrow \text{Type } = \\
| \text{Ret} : x:a \rightarrow \text{ctree} \text{a (post x) post} \\
| \text{Act} : \text{action pre post } \rightarrow \text{ctree} \text{a pre post} \\
| \text{Par} : \text{ctree} \text{a p q } \rightarrow \text{ctree} \text{a' p' q' } \rightarrow \text{ctree} \text{a} \& \text{a'} (p \& \text{st:star} \& p') (\lambda (x, x') \rightarrow q \& \text{x'st:star} \& q' \& x') \\
| \text{Bind} : \text{ctree} \text{a p q } \rightarrow ((x:a) \rightarrow \text{Dv} (\text{ctree} \text{b} (q x) r)) \rightarrow \text{ctree} \text{b p r}
\]

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 expressible in a primitive way within F*, and are soundly isolated from its pure, 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:

\[
\text{val run (e:ctree st a p q) : NMST a st:evolves } (\lambda m \rightarrow \text{st:interp} \text{ p m}) (\lambda _ x m' \rightarrow \text{st:interp} \text{ q x m'})
\]

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; 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 explicitly 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 unit of allocation. 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 supplementary material includes a snapshot of our current development, totaling around 11,000 lines of code and proof. Building the core semantics on an effectful foundation has been pleasingly compact: all the definitions of the core semantics and its proof of soundness fit in only 1,400 lines of documented F* code.

2 F* BACKGROUND AND BASIC INDEXED ACTION TREES

F* is a program verifier and a proof assistant based on a dependent type theory (like Coq or Agda) and a hierarchy of predicative universes. F* also has a dependently typed metaprogramming system inspired by Lean and Idris (called Meta-F*) that allows using F* itself to build and run tactics for proving or program construction. More specific to F* is its effectful type system, extensible with user-defined effects, and its support for SMT solving to help 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 λ(b₁)...(bₙ) → t introduces a lambda abstraction, whereas b₁ → ... → bₙ → 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 x₁ and x₂ for the first and second dependent projection maps.
2.1 A total semantics of concurrency

We start by presenting a total semantics for concurrency, introducing \( F^\star \) while also emphasizing that some of our core ideas should transfer to type theories like Agda or Coq that only have total functions. Later, we extend our representation with support for divergence and other effects.

Our first step is to define a type of state-passing atomic actions, action\_tot a = state \rightarrow \text{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 \( \text{Tot} \) 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 \( F^\star \). All unannotated arrows are \( \text{Tot} \) 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.

\[
\text{type ctree\_total : nat } \rightarrow \text{Type } \rightarrow \text{Type } = \\
\text{| Ret : #a_ \rightarrow x:a } \rightarrow \text{ctree\_total } 0 \ a \\
\text{| Act : #a_ \rightarrow act:action\_tot a } \rightarrow \text{ctree\_total } 1 \ a \\
\text{| Par : (#aL #aR #nL #nR :_) } \rightarrow \text{ctree\_total } \text{nl aL } \rightarrow \text{ctree\_total } \text{nr aR } \rightarrow \text{ctree\_total } (\text{nl+nr+1}) (\text{aL } & \text{ aR}) \\
\text{| Bind : (#a b #n1 #n2 :_) } \rightarrow \text{f:ctree\_total } n1 \ a \rightarrow g(x:a ) \rightarrow \text{ctree\_total } n2 \ b ) \rightarrow \text{ctree\_total } (n1+n2+1) b \\
\text{type nctree\_total (a:Type) = n:nat } & \text{ctree\_total } n a}
\]

The type ctree\_total induces a monad 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, making ctree\_total a graded monad (Katsumata 2014). 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:

\[
\text{type tape } = \text{nat } \rightarrow \text{bool} \\
\text{type nst (a:Type) = tape } & \text{nat } & \text{state } \rightarrow \text{a } & \text{nat } & \text{state} \\
\text{let return (a:Type) (x:a) : nst } a = \lambda(_, \ n, s) \rightarrow x, \ n, \ s \\
\text{let bind (a b:Type) (f:nst a) (g:a } \rightarrow \text{nst } b) : \text{nst } b = \lambda(t, \ n, \ s) \rightarrow \text{let } x, \ n1, \ s1 = f (t, \ n, \ s) \text{ in } (g x) (t, \ n1, \ s1) \\
\text{let sample () : nst } \text{bool } = \lambda(t, \ n, \ s) \rightarrow t \ n, \ n+1, \ s \\
\text{let get () : nst } \text{state } = \lambda(_, \ n, \ s) \rightarrow s, \ n, \ s \\
\text{let put (s:state) : nst } \text{unit } = \lambda(_, \ n, _.) \rightarrow () , \ n, \ s}
\]

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 \( F^\star \), making use of its effect system to package the nst monad as an user\-defined effect.

A user-defined effect in \( F^\star \) introduces a new abstract computation type backed by an existing \( F^\star \) 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 \( F^\star \). In particular, \( F^\star \) 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.

\[
\text{total new_effect \{ NST : a Type \rightarrow Effect with repl=nst; return=return; bind=bind\}}
\]

let sample () = NST?.reflect (sample()) let get () = NST?.reflect (get()) let put s = NST?.reflect (put s)

The type of sample is unit \rightarrow NST bool, where the computation type at the right of the arrow indicates
that sample has NST effect—using 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 recursively evaluating to return a step on either the left or the right.

\[
\text{let rec run \#a (r:nctrree_total a) = r:nctrree_total a \{ Return? r' \land r' < r, 1 \}}
\]

let rec step \#a (redex:nctrree_total a) : NST (reduce redex) (decreases redex,1)

\[
\text{match redex with}
\]

| Ret \_ \rightarrow redex
| Act act \rightarrow let s0 = get () in let x, s1 = act s0 in put s1; (| \_ Ret x |
| Bind (Ret x) g \rightarrow (| \_, g x |
| Bind f g \rightarrow let (| \_, f \_ |) = step (| \_, f |) in (| \_, Bind f g |
| Par (Ret x) (Ret y) \rightarrow (| \_, Ret (x, y) |
| Par l (Ret y) \rightarrow let (| \_, l \_ |) = step (| \_, l |) in (| \_, Par l' (Ret y) |
| Par (Ret x) r \rightarrow let (| \_, r \_ |) = step (| \_, r |) in (| \_, Par (Ret x) r' |
| Par l r \rightarrow
| if sample () \text{ then let (| \_, l' \_ |) = step (| \_, l |) in (| \_, Par l' r' |) else let (| \_, r' \_ |) = step (| \_, r |) in (| \_, Par l r' |)
|}
\]

\[
\text{let rec run \#a (p:nctrree_total a) : NST (nctrree_total a) (decreases p,1) = if Return? p \text{ then p else run (step p)}
\]

The other interesting element is proving that these definitions are well-founded. For that, we
enrich the type of step redex to return a refinement type reduct redex which states that the result
is either a Return node or its index is strictly less that the index of the redex. This, together with
the decreases annotations, is sufficient for F* to prove (using an SMT solver) that step and run are
terminating. Similar proofs could be done in other proofs assistants, though the specifics would
differ, e.g., in Agda one might use sized types (Abel 2007).

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

\textbf{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
properness 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 \rightarrow Dv unit = \text{\_}() \rightarrow loop (). From
the perspective of F*’s logical core, \text{\_} \rightarrow Dv b is an abstract, un-eliminable type.

\textbf{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 $\lambda$-calculus with primitive state.

**effect MST (a:Type) (state:Type) (p:preorder state) (req:state $\to$ prop) (ens:state $\to$ a $\to$ state $\to$ 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 $s0 \times s1$. 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:
  
  \[
  \text{val get #state #p () : MST state state p (requires } \lambda s \to \top) (\text{ensures } \lambda s0 r s1 \to s0=\equiv s1 \land r=\equiv s0)\]

- **Put** the state, but only when the new state $s1$ is related to the old one $s$ by $p$:
  
  \[
  \text{val put #state #p (s1:state) : MST unit state state p (requires } \lambda s \to \top) (\text{ensures } \lambda _ _ s \to s=\equiv s1)\]

- **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.
  
  \[
  \text{let stable_sprop #state (p:preorder state) = q(state } \to \top)[\forall s0 s1. q s0 \land p s0 s1 \implies q s1] \\text{val witnessed #state #p (q:stable_sprop p) : prop} \\text{val witness #state #p (q:stable_sprop p) : MST unit state state p (requires } \lambda s \to \top) (\text{ensures } \lambda _ _ s \to s=\equiv s1)\]

- **Recall** stable predicates: Having witnessed $q$, one can use recall $q$ to re-establish it at any point.
  
  \[
  \text{val recall #state #p (q:stable_sprop p[witnessed q]) : MST unit state state p (requires } \lambda s \to \top) (\lambda _ s1 \to s0=\equiv s1 \land q s1)\]

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.

### 3 Indexed Action Trees and a Partial Correctness Separation Logic

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

**type ctree (st:state) (a:Type) (expects:st.slprop) (provides:a $\to$ 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.
Frame-preserving Actions

To define the type of action trees \( \text{ctree} \), let’s start by defining the type of atomic actions at the leaves of the tree:

\[
\text{let action \#st \#a (fp0:st.slprop) (fp1:a \rightarrow st.slprop) =}
\]
\[
\text{unit} \rightarrow \text{MST} \ a \ \text{st.mem st.evolves}
\]
\[
\text{(requires} \ \lambda m0 \rightarrow \text{st.interp (st.inv m0 \ast fp0) m0)}
\]
\[
\text{(ensures} \ \lambda m0 \times m1 \rightarrow \text{st.interp (st.inv m1 \ast fp1) m1 \ast preserves_frame fp0 (fp1) m0 m1)}
\]

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 \( fp1 \times \) on the final state \( m1 \). In both the pre- and postcondition, we expect \( st.inv \) to hold separately. Finally, and perhaps most importantly, the \( \text{preserves_frame} \) 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 \( \text{preserves_frame} \) (that an action must provide in its postcondition) states that all frames separate from \( st.inv m0 \ast pre \) and valid in the initial state \( m0 \) remain separate from \( st.inv m1 \ast post \) and are valid in \( m1 \).

\[
\text{let preserves_frame \#st (pre post:st.slprop) (m0 m1:st.mem) =}
\]
\[
\forall(\text{frame:st.slprop}). \text{st.interp (st.inv m0 \ast pre \ast frame) m0} \Rightarrow \text{st.interp (st.inv m1 \ast post \ast frame) m1}
\]
An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs

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 ctree type does not carry a natural number index as we did in §2.1. On the other hand, if one were interested in a total correctness semantics, we foresee no problems (either in F* or in some other type theory) in augmenting ctree’s with the size indexes from §2.1 and proving a definitional interpreter terminating.

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.

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 emp (\lambda_ → 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) (\(\lambda_\_ -> (Act a2))) P) (\(\lambda_\_ -> (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.

\[
\begin{align*}
\text{val} \quad \text{run} \quad #st \quad #a \quad #fp0 \quad #fp1 \quad (f:\text{ctree \ st a \ fp0 \ fp1}) : \text{NMST a \ st.mem \ st.evolves} \\
\quad \text{(requires) \ } \lambda m0 \rightarrow \text{st.interp (st.inv m0 \ fp0 \ m0)} \quad \text{(ensures) \ } \lambda m0 \times m1 \rightarrow \text{st.interp (st.inv m1 \ fp1 x) m1}
\end{align*}
\]

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.

\[
\begin{align*}
\text{type} \quad \text{reduct} \quad #st \quad a = \mid \text{Reduct:} \quad \text{fp0:}\_ \rightarrow \text{fp1:}\_ \rightarrow \text{ctree \ st \ a \ fp0 \ fp1} \rightarrow \text{reduct a} \\
\text{val} \quad \text{step} \quad (f:\text{ctree \ st a \ fp0 \ fp1}) : \text{NMST (reduct a) \ st.mem \ st.evolves} \\
\quad \text{(requires) \ } \lambda m0 \rightarrow \text{st.interp (st.inv m0 \ fp0 \ m0)} \quad \text{(ensures) \ } \lambda m0 \times (\text{Reduct fp0 \ fp1 \_}) \rightarrow \\
\quad \text{st.interp (st.inv m1 \ fp0 \ m1 \ \_preserves\_frame \ fp0 \ fp0' \ m0 \ m1 \ \_\_stronger\_than\_fp1} \\
\end{align*}
\]

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 fp’ 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
An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs

...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 \(c0 \text{ f}\) 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 empty footprint.

```ocaml
let rec step #st #a #fp0 #fp1 (cctree st a fp0 fp1) = match c with |... |
| Frame (Ret fp0’ x) f → Reduct (fp0’ x * f) (\(\lambda x → fp0’ x * f\)) (Ret (\(\lambda x → fp0’ x * f\)) 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) (\(\lambda x → fp1’ x * f\)) (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, \(\text{preserves_frame_star}\), which states \(\text{preserves_frame_fp0 fp0’ m0 m1} \implies \text{preserves_frame_fp0 fp0’ m0 m1}\).

**Subsumption.** Reductions of the other structural rule, Sub, is simpler, we just remove the Sub node, as shown below; the refinement \(\text{sub_ok}\) 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 in §3.4.

```ocaml
| 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:

```ocaml
| Bind #fp2 (Ret fp0 x) g → Reduct (fp0 x) fp2 (g x)
| Bind #fp0 #fp1 #fp2 g f → 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.

```ocaml
| Par (Ret fp0L xL) (Ret fp0R xR) → Reduct (fp0L xL * fp0R xR) (\(\lambda (xL, xR) → fp0L xL * fp0R xR\)) (Ret (\(\lambda (xL, xR) → fp0L xL * fp0R xR\)) (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) (\(\lambda (xL, xR) → fp1L’ xL * fp1R xR\)) (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.

```ocaml
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 _ 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.

### 3.4 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):

```ocaml
| 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

```ocaml
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 inter-leavein semantics for concurrent programs, we do not intend to run programs using run (although we could, very inefficiently). 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 soundness: 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.

```plaintext
let addr = nat
let heap = addr → option cell

type mem = { heap : heap; ctr : nat; istore : istore }

let inv m : slprop = (∀ i.i ≥ m.ctr ⇒ m.heap i == None) ∧ istore_inv m
```

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—we’ll defer discussing this until §4.3.

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 ≤_p for the partial order induced by p:pcm a.

```plaintext

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 } }
```

¹In 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.
let \((\leq)\) \((#a:\text{Type})\) \((#pcm:pcm\ a)\) \((x\ y : a) = \exists\text{frame}.\ pcm\ .\ op\ x\ \text{frame} \Rightarrow y\)

type\ cell = | \text{Cell:} a:\text{Type} \rightarrow pcm:pcm\ a \rightarrow v:a \rightarrow 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 separate and combine disjoint memories. Note, 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 \((h:\text{h’:heap})\) \((a:\text{addr})\) = \text{match} \text{h a, h’ a with}
| Some \((\text{Cell t pcm v})\), Some \((\text{Cell t’ pcm’ v’})\) = \text{t=\text{t’} \land pcm\ =\= pcm’} \land pcm.composable \(v\ v’\) \(\_\rightarrow \top\)
let disjoint \(h0\ h1 = \forall a.\ \text{disjoint_addr h0 h1 a}\)
let join \((h0:\text{heap})\) \((h1:\text{heap|(disjoint h0 h1)})\) = \(\lambda a\rightarrow \text{match h0 a, h1 a with}\)
| None, None → None | None, Some \(x\) | Some \(x\), None → Some \(x\)
| Some \((\text{Cell t pcm v0})\), Some \((\text{Cell } _ \_ v1)\) → Some \((\text{Cell t pcm (pcm.op v0 v1)})\)

To model references to \(t\)-typed values with fractional permissions, we store at each cell a value
\(\text{frac t = option (t \& r:real[0.0 < r \leq 1.0])}\) and define a \(pcm\) \((\text{frac t})\) as shown below. Through the use of the composable predicate, we have the flexibility to use undecidable relations like propositional equality in our notion of partiality.

let one : \(\text{frac t = None}\)
let composable \((f0 \ f1:\text{frac t})\) = \text{match} \(f0, f1\ with\)
| Some \((v0, r0)\), Some \((v1, r1)\) → \(v0 == v1 \land r0 + r1 \leq 1.0\) \(\_ \rightarrow \top\)
let \(\text{op} \ (f0 \ f1:\text{frac t})\) = \text{match} \(f0, f1\ with\)
| None, \(f\) | \(f\), None → \(f\) | Some \((v, r0)\), Some \((_,\ r1)\) → Some \((v, r0 + r1)\)

In what follows, for simplicity, we present examples that are specialized to use this fractional
permission \(PCM\) only.\(^3\)

4.2 Separation Logic Propositions

Next, we define a language of separation logic propositions and give it an interpretation on heaps.

let reference a pcm = addr

type\ slprop = | PointsTo : #a:\text{Type} \rightarrow #pcm:pcm\ a \rightarrow r:\text{reference a pcm} \rightarrow v:a \rightarrow slprop
| Emp : slprop | And, Or, Star, Wand : slprop \rightarrow slprop \rightarrow slprop | Ex, All : #a:\text{Type0} \rightarrow (a \rightarrow slprop) \rightarrow slprop
| Refine : p:slprop \rightarrow f:(heap \rightarrow prop)(v h0 h1.\ f h0 \land disjoint h0 h1 \Rightarrow f (join h0 h1)) \rightarrow slprop

This is mostly standard, except for three remarks. First, the PointsTo assertion relates a reference
(indexed by a type and a \(PCM\)) to a value, mirroring the contents of a cell. The existential and universal quantifiers,
\(Ex\) and \(All\), are predicative inasmuch as they support quantification over a

\(^2\)On universes and higher order stores: We define our memory model universe polymorphically, so that it can store values in higher universes, e.g., for storing values at existential types (§5.3.3). 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.

\(^3\)At the time of submission, while we provide a memory model with PCMs, the rest of our implementation is specialized with respect to the PCM of fractional permissions. Generalizing our full development to use generic PCMs is a work in progress expected to be straightforward.
universe strictly smaller than the universe of \( \text{slprop} \) itself. Finally, we include a form \text{Refine } p \ f \) that, in the spirit of implicit dynamic frames, supports refining \( p \) with an affine heap predicate.

Interpreting an \text{slprop} as a heap proposition is straightforward:

\[
\begin{align*}
\text{let rec interp (p:slprop) (h:heap) = match } p \text{ with } & | \text{Emp} \rightarrow \tau \\
| \text{Pts_to } # a \rightarrow (\text{match } h \text{ with } & \text{Some (Cell a’ p’ v’)) } \rightarrow a \equiv a’ \land p \equiv p’ \land v \leq p \lor v | _{\_} \rightarrow \perp \\
| \text{And } p1 p2 \rightarrow \text{interp } p1 h \land \text{interp } p2 h & \mid \text{Or } p1 p2 \rightarrow \text{interp } p1 h \lor \text{interp } p2 h \\
| \text{Star } p1 p2 \rightarrow \exists h1 h2. h1 \text{`disjoint’ } h2 \land h = join h1 h2 \land \text{interp } p1 h1 \land \text{interp } p2 h2 \\
| \text{Wand } p1 p2 \rightarrow \forall h1. h \text{`disjoint’ } h1 \land \text{interp } p1 h1 \implies \text{interp } p2 (join h1 h) \\
| \text{Ex } f \rightarrow \exists x. \text{interp } (f x) h & \mid \text{All } f \rightarrow \forall x. \text{interp } (f x) h \mid \text{Refine } p f \rightarrow \text{interp } p h \land f h
\end{align*}
\]

This interpretation also induces an equivalence relation on \text{slprop}, i.e., \( p \sim q \iff (\forall h. \text{interp } p h \iff \text{interp } q h) \) and it is easy to prove that Star and Emp form a (total) commutative monoid with respect to \( \sim \), as required by the state interface of our semantics.

In what follows, we overload the use of \( \mathcal{F}^* \)’s existing connectives \( \exists, \forall, \land, \lor \) for use with \text{slprop}. We write \text{emp} for \text{Emp}; \( \ast \) and \( \rightarrow \) for Star and Wand. Borrowing \( \mathcal{F}^* \)’s notation for refinement types, we also write \text{h:}p{f} for \text{Refine } p (\lambda h \rightarrow f) and \text{pure } p for _\text{emp}(p). Finally, specializing the PCM-based memory model to the use of references with fractional permissions, we write \text{ref } a for \text{ref } a (\text{pcm_frac } t) and \text{pts_to } r f v for PointsTo r (Some (v, f)), sometimes omitting some of the indexes when they are irrelevant (interpreting them as existentially quantified).

### 4.3 Introducing Invariants: Preorders and the istore

Beyond the traditional separation logic assertions, it is useful to also support a notion of \textit{invariant} that allows a non-duplicable \text{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).

\[
\begin{align*}
(1) \ P \Rightarrow_{\mathcal{E}} P^N & \quad (2) \ \text{persistent}(P^N) & \quad (3) \ \{ \ast P \ast Q \} e \{ \ast P \ast R \} \mathcal{E} \land N & \quad \text{atomic}(e) \quad N \in \mathcal{E} \\
\end{align*}
\]

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

The second rule states that an invariant is persistent or duplicable: i.e., \( P^N \implies P^N * P^N \). 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 \( P^N \), 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, \( P^N \) 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 \( \ast P \) 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.

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

\[
\text{val new_invariant (p:slprop) : action (ival p) p emp}
\]
Recall the action type from §3.1. The type above states that given possession of \( p \), \( \text{new}_\text{invariant} \) consumes \( p \), providing only \( \text{emp} \), but importantly, returning a value of type \( \text{ival} \ p \): our representation of an invariant—\( \text{new}_\text{invariant} \) models Iris’ update modality to allocate an invariant, i.e., the first of the three rules above. Being a value, \( \text{ival} \ p \) 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 \( p \ast q \) and provides \( p \ast r \) can be turned into a command that only expects \( q \) and provides \( r \), as long as an \( \text{ival} \ p \) value can be presented as evidence that \( p \) is an invariant.

\[
\text{val with}_\text{invariant} (\text{ival} p) (\text{atomic} a (p \ast q) (p \ast r)) : \text{atomic} a q r
\]

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

\[
\text{let \ text{istore} = list slprop \ let \ text{istore}_\text{inv} (i:istore) : slprop = List.fold_right (\_ \text{emp} i i) \ let \ text{inv}_\text{name} = nat \text{let \ text{evolves} :text{preorder} mem = \lambda m0 m1 \rightarrow } \\
\forall (i:inv\text{name}). \text{List.nth i m0.istore} == \text{None} \lor \text{List.nth i m1.istore} == \text{List.nth i m1.istore} \text{let \ inv}_\text{for}_p (i:inv\text{name}) (p:slprop) (m:mem) = \text{Some p == List.nth i m.istore} \text{let \ (\_ \_ \_ \_ i p = \text{witnessed} (inv}_\text{for}_p i p) \text{let \ ival p:slprop = i:inv\text{name}[i \rightarrow p]}
\]

Now, to define the \text{new}_\text{invariant} \( p \) action we simply extend the \text{istore}, witness that \( p \) is now an invariant, and return the address of the newly allocated invariant.

\[
\text{let \ new}_\text{invariant} (p:slprop) : \text{action (ival p) p emp} = \lambda () \rightarrow \\
\text{let \ m = get () in put (m with istore=m.istore@[p]); let \ i = List.length m.istore in witness (inv}_\text{for}_p i p); i
\]

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 \( \text{atomic}(e) \) 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 \text{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 type action a p q is equivalent to atomic a []_ p q.

\[
\text{let istore_inv'} \text{uses } p s = \text{List.fold_right_i} (\lambda i q \rightarrow \text{if } i \in \text{uses then } q \text{ else } p * q) \text{ ps emp}
\]

\[
\text{let inv' uses } m = \ldots \text{m.ctr} \ldots \land \text{istore_inv'} \text{uses } m \text{.istore}
\]

\[
\text{let atomic } (\text{a:Type}) \text{ (uses:set inv_name) (is_ghost:bool) (p:slprop) (q: a } \rightarrow \text{slprop) =}
\]

\[
\text{unit } \rightarrow \text{MST a mem evolves}
\]

\[
\begin{align*}
\text{(requires)} & \quad \lambda m \rightarrow \text{interp} \text{ (inv' uses } m * p) \text{ m) } \\
\text{(ensures)} & \quad \lambda m0 x m1 \rightarrow \text{interp} \text{ (inv' uses } m1 * q x) \text{ m1 } \land \text{preserves_frame } p \text{ (q x) m0 m1)
\end{align*}
\]

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

\[
\text{val bind_atomic } \text{#a } \#b \#u \#q \#r \#g1 (\#g2:bool(g1 || g2))
\]

\[
(e1: \text{atomic } a \text{ u } g1 \text{ p } q) \text{ (e2: } (x:a \rightarrow \text{atomic } b \text{ u } g2 \text{ (q x) } r)) : \text{atomic } b \text{ uses } (g1 && g2) \text{ p r}
\]

The type of CAS. An atomic compare-and-set instruction can be given the type below. It states that given a reference r to a word-sized integer for which we have full ownership, 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.

\[
\text{val cas } (\#u: \text{set inv_name}) \text{ (r:ref uint32) (old new:uint32) (v:erased uint32) :}
\]

\[
\text{atomic (b:bool(b=(v=old))) u false (pts_to r 1.0 v) (} \lambda b \rightarrow \text{if } b \text{ then } pts_to r 1.0 \text{ new else pts_to r 1.0 v})
\]

Opening and closing an invariant. The final piece of the puzzle is the withInvariant 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.

\[
\text{val withInvariant } \text{#a } \#p \#q \#u \#g (i:ival p) (e:atomic a (i } \lor u) g (p * q) (} \lambda x \rightarrow p * r x) : \text{atomic } a \text{ u } g \text{ q r}
\]

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 withInvariant 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 \sim p * i \sim q \vdash p \iff q is inconsistent—the conclusion of the rule must be guarded under a later, i.e.,
it should be \( \ast (p \iff q) \). 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 \( \perp \neq \perp \); instead, an explicit step of computation via the recall action is necessary to extract a contradiction from witnessed \( \perp \). As such, \( i \vdash p + i \vdash q \vdash (p \iff q) \) is not derivable in SteelCore, although with a step of computation, the Hoare triple \( [i \vdash p + i \vdash q] \) recall \( i \{ (p \iff q) \} \) 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, \( \Rightarrow \mathcal{E} \). 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 \( F^\ast \)’s existing support for erased types to model ghost state and updates within Hoare triples, rather than within the logic itself.\(^4\) For instance, the following action represents a ghost read: it dereferences \( x \), returning its contents only as an erased \( a \), an \( F^\ast \) type that is extracted to unit—values of type erased a cannot be eliminated in computationally-relevant contexts.

\[
\text{val ghost_read (x:ref a) : atomic (erased a) u true (pts_to r f) (} \lambda v \rightarrow pts_to r f v) \]

Other actions. Defining the canonical actions for allocating, reading, writing, sharing, and gathering references is relatively simple. We show their signatures below (as before, unbound variables are implicitly universally quantified):

\[
\begin{align*}
\text{val alloc (v:a) : action (ref a) emp (} \lambda r \rightarrow pts_to r \ 1.0 \ v) \\
\text{val (! (r:ref a) : action a (pts_to r f _) (} \lambda v \rightarrow pts_to r f v) \\
\text{val (=: (r:ref a) (v:a) : action unit (pts_to r 1.0 _ ) (} \lambda _ \rightarrow pts_to r 1.0 v) \\
\text{val free (r:ref a) : action unit (pts_to r 1.0 _) (} \lambda _ \rightarrow emp) \\
\text{val share (r:ref a) (v:erased a) : action unit (pts_to r f v) (} \lambda _ \rightarrow pts_to r (f/2) v \ast pts_to r (f/2) v) \\
\text{val gather (r:ref a) (v0 v1:erased a) : action (} (_\text{unit}[v0=\rightarrow v1]) (pts_to r f v0 \ast pts_to r g v1) (} \lambda _ \rightarrow pts_to r (f + g) v0) \\
\end{align*}
\]

4.5 Monotonic References for SteelCore

So far, we have used the monotonic state effect within SteelCore’s underlying semantics to soundly model stored invariants. One may wonder if, aside from invariants, there are other ways to exploit the use of the monotonic state effect in the SteelCore’s surface program logic. In this section, we show how to model monotonic references, i.e., references extended with preorders that constrain how their content is permitted to evolve. In §5.3, we make use of monotonic references to internalize a proof of correctness of a channel typing discipline.

PCMs vs Preorders. Briefly, let’s consider once again the general setting of SteelCore’s memory model with PCMs (rather than just fractional permissions): As has been observed before by the authors of Iris and others, since every \( p:pcm a \) induces a preorder \( \leq_p \), and since all state updates must be frame-preserving, the following triple is derivable: \( [pts_to r p v0] \) read \( r [\lambda v1 \rightarrow pts_r p v1 \ast pure (v0 \leq_p v1)] \). That is, knowing that \( r \) once pointed to \( v0 \), reading \( r \) now returns a value \( v1 \) that is at least \( v0 \) according to the preorder \( \leq_p \). Thus, by carefully choosing the PCM \( p \), one can design protocols that constrain how the contents of a reference evolves. On the other hand, given a particular state machine describing a protocol, it is not always immediately obvious how to encode the reachability relation of the state machine as a PCM. That is, while all PCMs induce preorders, designing a PCM to capture a particular preorder of interest is not always direct. As such, in SteelCore, we augment

\(^4\)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.
the memory model to support both PCMs and preorders, as we show below. This lends itself to a specification style where PCMs can be used to describe spatial separation and access permissions, while preorders can be used to express how memory evolves.

**Enriching the memory model with preorders.** At each cell in the memory, instead of storing just a pcm, we store a pcm_pre, i.e., a pcm extended with a preorder on the carrier type.

\[
\text{type } \text{pcm}_\text{pre}(a:\text{Type}) = \{ \text{pcm} : \text{pcm} a ; \text{pre} : \text{preorder} a \} \\
\text{type } \text{cell} = | \text{Cell} : a : \text{Type} \rightarrow \text{pcm} : \text{pcm}_\text{pre} a \rightarrow v : a \rightarrow \text{cell}
\]

Next, we extend the evolves relation to take into account the preorders at each heap cell: the global evolution relation now includes the pointwise conjunction of the preorders stored at each cell.

\[
\text{let } \text{evolves} : \text{preorder mem} = \lambda m0 m1 \rightarrow m0.\text{ctr} \leq m1.\text{ctr} \land \ldots \text{istore evolves as before} \ldots \land \forall (a: \text{addr}). \text{match } m0 a, m1 a \text{ with } | \text{None}, \text{None} \rightarrow \top \\
| \text{None}, \text{Some } _a \rightarrow a \geq m0.\text{ctr} (** a was allocated **) | \text{Some } _a, \text{None} \rightarrow a < m0.\text{ctr} (** a was freed **) \\
| \text{Some } (\text{Cell } t0 p0 v0), \text{Some } (\text{Cell } t1 p1 v1) \rightarrow t0 = t1 \land p0 = p1 \land p0.\text{pre } v0 v1 (** updated respecting preorder **) \\
\]

Given these definitions, we can build a type of monotonic references shared using fractional permissions, with the signature shown below. The main point of interest is the signature of write_mref, which requires proving that the new value \(v\) is related to the old value by the preorder \(p\).

\[
\text{val } \text{mref } (a: \text{Type}) (p: \text{preorder } a) : \text{Type} \\
\text{val } \text{alloc_mref } #a (p: \text{preorder } a) (v:a) : \text{action } (\text{mref } a p) \text{ emp } (\lambda r \rightarrow \text{pts_to } r 1.0 v) \\
\text{val } \text{read_mref } #a #f #p (r: \text{mref } a p) : \text{action } (\text{pts_to } r f _) (\lambda v \rightarrow \text{pts_to } r f v) \\
\text{val } \text{write_mref } #a ##p (r: \text{mref } a p) (\text{old:erasured } a) (v:a[p \text{ old } v]) : \text{action } \left(\text{pts_to } r 1.0 \text{ old} \right) (\lambda _a \rightarrow \text{pts_to } r 1.0 v) \\
\]

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_mref returns a pure, abstract predicate observed \(r q\) when the current value of \(r\) satisfies a stable property \(q\); and recall_mref eliminates observed \(r q\) into \(q v\), for \(v\) the current value of \(r\).

\[
\text{val } \text{observed } #a #p (r: \text{mref } a p) (q:a \rightarrow \text{prop}) : \text{prop} \\
\text{val } \text{witness_mref } #a ##p (r: \text{mref } a p) (q: \text{stable_prop } a p) (v: \text{erasured } a q v) : \text{action } \left(\text{pts_to } r f v \right) (\lambda _a \rightarrow \text{pts_to } r f v \ast \text{pure } (\text{observed } r q)) \\
\text{val } \text{recall_mref } #a ##p (r: \text{mref } a p) (q: \text{stable_prop } a p) (v: \text{erasured } a) : \text{action } \left(\text{pts_to } r f v \ast \text{pure } (\text{observed } r q) \right) (\lambda _a \rightarrow \text{pts_to } r f v \ast \text{pure } (q v)) \\
\]

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.

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. Higher 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. 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.

```plaintext
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 p = r:ref bool & ival (lockinv r p)
```

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

```plaintext
let new_lock p : Steel (lock p) p 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 decorate the relevant triples in the term using the notation \( \{ e \} \{ q \} \). 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 \( b=false \rightarrow p \) before closing the invariant. We could also return the resulting boolean to avoid losing information.

```plaintext
let release ((r, i):lock p) : Steel unit p 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 → 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.

```plaintext
let rec acquire ((r, i):lock p) : Steel unit emp 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 → 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 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 g = (Steel unit p (λ _ → q)) (g; (thread q → Steel unit 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)
```

5.3 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 channels 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.

5.3.1 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∗ to check that the type is never used in a computationally relevant context and all protocol values are erased to () during extraction.
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.

\[
\text{let } xy = \text{Msg int } (\lambda x \rightarrow \text{Msg } y: \text{int}(y = x + 1)) (\lambda y \rightarrow \text{Ret }())
\]

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 \((xy \text{ `bind` } (\lambda _ \rightarrow \text{Ret true}) \) \( \) \( )\) \( )\) is a channel on which one can repeatedly send related pairs of successive integers.

### 5.3.2 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.

\[
\text{let rec } \text{hnf } (p:\text{protocol }a): (q:\text{protocol }a)((\text{Ret } q \lor \text{Msg } q) \land (\neg(\text{DoWhile } p) \implies (p = q)))
\]

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.

\[
\text{let more } (p:\text{protocol }a): \text{bool } = \text{Msg? } (\text{hnf } p)
\]

\[
\text{let } \text{msg_t } (p:\text{protocol }a) : \text{Type } = \text{match } \text{hnf } p \text{ with } \text{Msg } a \_ \rightarrow a | \text{Ret } \_ \rightarrow \_ \\
\text{let step } (p:\text{protocol }a(\text{more } p)) (x:\text{msg_t } p) : \text{protocol }a = \text{Msg?} k (\text{hnf } p) x
\]

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 \( p \).

\[
\text{let } \text{prot } = \text{protocol unit} \\
[@\text{erasable}] \text{type } \text{trace } : \text{prot } \rightarrow \text{prot } \rightarrow \text{Type } = | \text{Waiting } p : \text{prot } \rightarrow \text{trace } p \\
| \text{Message } : \text{from:prot}[\text{more } p] \rightarrow x:\text{msg_t } f r o m \rightarrow \text{to:prot } \rightarrow \text{trace } (\text{step from } x) \rightarrow \text{trace from to} \\
\text{type } \text{trace_of } p = \{ \text{until:prot; tr:trace } p \text{ until } \}
\]

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.

\[
\text{val } \text{extend}1 (\#\text{from } \#\text{to:protocol unit}) (t:\text{trace from to}[\text{more } t]) (m:\text{msg_t } t) : \text{trace from (step to } m)
\]

\[
\text{let } \text{next } p t 0 t 1 = \text{more } t 0.0 .t \land (\exists \text{ msg. t 1 to } = \text{step } t 0.0 .t \text{ msg } \land t 1 .t .t r = \text{extend}1 t 0.0 .t .t r \text{ msg})
\]

\[
\text{let } (\rightarrow) (\#p:\text{protocol unit}) : \text{preorder } (\text{trace_of } p) = \text{ReflexiveTransitiveClosure.closure } (\text{next } p)
\]

We will use the preorder \( \rightarrow \) 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.
5.3.3 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.

```plaintext
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`.

```plaintext
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`.

```plaintext
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.

```plaintext
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 #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.

```plaintext
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 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 #: (c:chan_t i) = \vs\ 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.crt==vr.crt+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 (the full implementation with the proofs are much more verbose and are in the supplement).

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 #: cur c = acquire c.lock ; let vs, vr = !c.chan.send, !c.chan.recv in
  if vs.crt==vr.crt 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 \mapsto _) 
let extend_trace #: cur c prev = acquire c.lock ; let tr = !c.chan.trace in recall_mref c (prev \mapsto _);
  witness_mref c.chan.trace (tr \mapsto _) 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 is predicative, does not internalize Hoare triples, and does not make use of step indexing or any of Iris’ several modalities. However, many lacking features in the logic are recovered using the host language, F\textsuperscript{⋆}, facilities. E.g., we support rules for manipulating dynamically created, named invariants \( i \downarrow p \) in a style inspired by Iris, using the underlying effect of monotonic state. Further, we make up a bit for the lack of impredicativity by relying on abstraction in F\textsuperscript{⋆} (§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\textsuperscript{⋆}’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 (§5.3.3). 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;
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 3.4 have one parameter (the state type class) and four logical specification 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. (2017), 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 stprops 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 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 Matt Parkinson, Derek Dreyer, Ralf Jung, Aleks Nanevksi and all the members of Project Everest for answering questions and many useful discussions.

REFERENCES


