Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic

AYMERIC FROMHERZ, Carnegie Mellon University, USA
ASEEM RASTOGI, Microsoft Research, India
NIKHIL SWAMY, Microsoft Research, USA
SYDNEY GIBSON, Carnegie Mellon University, USA
GUIDO MARTÍNEZ, CIFASIS-CONICET, Argentina
DENIS MERIGOUX, Inria Paris, France
TAHINA RAMANANANDRO, Microsoft Research, USA

Steel is a language for developing and proving concurrent programs embedded in F*, a dependently typed programming language and proof assistant. Based on SteelCore, a concurrent separation logic (CSL) formalized in F*, our work focuses on exposing the proof rules of the logic in a form that enables programs and proofs to be effectively co-developed.

Our main contributions include a new formulation of a Hoare logic of *quintuples* involving both separation logic and first-order logic, enabling efficient verification condition (VC) generation and proof discharge using a combination of tactics and SMT solving. We relate the VCs produced by our quintuple system to solving a system of associativity-commutativity (AC) unification constraints and develop tactics to (partially) solve these constraints using AC-matching modulo SMT-dischargeable equations.

Our system is fully mechanized and implemented in F*. We evaluate it by developing several verified programs and libraries, including various sequential and concurrent linked data structures, proof libraries, and a library for 2-party session types. Our experience leads us to conclude that our system enables a mixture of automated and interactive proof, making it productive to build programs foundationally verified against a highly expressive, state-of-the-art CSL.

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

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

1 INTRODUCTION

Structuring programs with proofs in mind is a promising way to reduce the effort of building high-assurance software. There are many benefits: the program structure can simplify proofs, while proofs can simplify programming too by, for example, eliminating checks and unnecessary cases. Programming languages of many different flavors embrace this view and we use the phrase *proof-oriented programming* to describe the paradigm of co-developing proofs and programs.

Dependently typed programming languages, like Agda and Idris, are great examples of languages that promote proof-oriented programming. As Brady (2016) argues, the iterative "type-define-refine" style of type-driven development allows programs to follow the structure of their dependently typed specifications, simplifying both programming and proving. From a different community, languages like Dafny (Leino 2010), Chalice (Leino et al. 2009), and Viper (Müller et al. 2016) enrich imperative languages with first-order program logics for program correctness, driving program development from Floyd-Hoare triples, with proof automation using SMT solvers. Hoare Type Theory (Nanevski et al. 2008) combines these approaches, embedding Hoare logic in dependently

This work is licensed under a Creative Commons Attribution 4.0 International License
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/8-ART85
https://doi.org/10.1145/3473590

```
let swap (#v1 #v2:ghost int) (r1 r2:ref int)
  : SteelCore unit
      (pts_to r1 #v1 ∗ pts_to r2 #v2)
      (λ _ → pts_to r1 #v2 ∗ pts_to r2 #v1)
  = let x1 = frame (read r1) (pts_to r2 #v2) in
    commute_star (pts_to r1 #v1) (pts_to r2 #v2);
    let x2 = frame (read r2) (pts_to r1 #v1) in
    frame (write v2 x1) (pts_to r1 #x2);
    commute_star (pts_to r2 #x1) (pts_to r1 #v1);
    frame (write r1 x2) (pts_to r2 #v2);
```

```
let swap (r1 r2:ref int) : Steel unit
   (ptr r1 ∗ ptr r2) (λ _ → ptr r1 ∗ ptr r2)
   (requires λ_ → ⊤)
   (ensures λs _ s' →
      s'.[r1]=s.[r2] ∧ s'.[r2]=s.[r1])
  = let x1 = read r1 in
    let x2 = read r2 in
    write r2 x1;
    write r1 x2
```

Fig. 1. SteelCore implementation of swap (left); Steel version (right)

typed Coq with tactic-based proofs, while F⋆ (Swamy et al. 2016) follows a similar approach but, like Dafny, Chalice and Viper, uses an SMT solver for proof automation.

In this paper, we aim to design a proof-oriented programming language based on SteelCore (Swamy et al. 2020), a recent concurrent separation logic (CSL) (O'Hearn 2004; Reynolds 2002) for dependently typed programs formalized in F⋆. Our goal is to integrate the expressive power of the SteelCore logic within a higher-order, dependently typed programming language with shared-memory and message-passing concurrency, with proof automation approaching what is offered by Dafny, Chalice, and Viper, but with soundness ensured by construction upon the foundations of SteelCore.

We have our work cut out: SteelCore, despite providing many features as a logic, including an impredicative, dependently typed CSL for partial-correctness proofs, with a user-defined partial commutative monoid (PCM)-based memory model, monotonic state, atomic and ghost computations, and dynamically allocated invariants, all of whose soundness is derived from a proof-oriented, intrinsically typed definitional interpreter in F⋆, is far from being usable directly to build correct programs.

SteelCore's main typing construct is a Hoare type, SteelCore a p q, describing potentially divergent, concurrent, stateful computations returning a value of type a and whose pre- and postconditions are p:slprop and q:a → slprop, where slprop is the type of separation logic propositions. Figure 1 shows, on the left, a SteelCore program that swaps the content of two references. Calls to frame wrapping each action combined with re-arrangements of slprops with commute_star overwhelm the program—the pain is perceptible.

Through several steps, we offer instead Steel, an embedded domain-specific language (DSL) within F⋆ based on SteelCore, which enables writing the swap program on the right of Figure 1. We briefly call out some of its salient features. First, we introduce a new "quintuple" computation type, shown below:

```
Steel a (p:slprop) (q:a → slprop) (requires (r:pre p)) (ensures (s:post p a q))
```

The additional indices r and s are *selector predicates*, that depend only on the p-fragment of the initial memory and q-fragment of the final memory, i.e., they are self-framing, in the terminology of Parkinson and Summers (2012). These predicates are SMT encodeable and allow a useful interplay between tactic-based proofs on slprops and SMT reasoning on the content of memory. In swap, these predicates also remove the need for existentially quantified ghost variables to reason about the values stored in the two references (i.e., the function arguments v1 and v2). Next, through the use of F⋆'s effect system, we encode a syntax-directed algorithm to automatically insert applications

of the frame rule at the leaves of a computation, while a tactic integrated with the DSL solves for frames using a variant of AC-matching (Kapur and Narendran 1987).

Freed from the burden of framing, the program's intent is evident once more. The swap program in Steel is perhaps close to what one would expect in Chalice or Viper, but we emphasize that Steel is a shallow embedding in dependently typed F★ and the full SteelCore logic is available within Steel. So, while Viper-style program proofs are possible and encouraged, richer, dependently typed idioms are also possible and enjoy many of the same benefits, e.g., automated framing and partial automation via SMT. Indeed, our approach seeks only to automate the most mundane aspects of proofs, focusing primarily on framing. For the rest, including introducing and eliminating quantifiers, rolling and unrolling recursive predicates, writing invariants, and manipulating ghost state, the programmer can develop lemmas in F★'s underlying type theory and invoke those lemmas at strategic points in their code—the Steel library provides many generic building blocks for such lemmas. The result is a style that Leino and Moskal (2010) have called *auto-active* verification, a mixture of automated and interactive proof that has been successful in several other languages, including in other large F★ developments, but now applied to SteelCore's expressive CSL.

1.1 Contributions

Quintuples: Selectively separating separation and first-order logic. We develop for Steel a verification condition generator and hybrid tactic- and SMT-based solver for a separation logic of quintuples whose main judgment involves a computation type $\{\ P \mid R\ \}\ x{:}t\ \{\ Q \mid S\ \}$, where P, Q are slprops as usual, but R, S are first-order logic encodeable self-framing selector predicates. Proof obligations in our formulation are in two classes: separation logic goals, relating the slprops in a judgment, and SMT encodeable goals relating the selector predicates. This allows us to write efficient reflective tactics that focus on the former, while the latter are encoded efficiently to SMT by F★, as usual. In a style reminiscent of Nelson and Oppen's (1979) cooperating decision procedures, we show how tactics and SMT share information through equalities on uninterpreted symbols. Of course, proof obligations remain undecidable and what automation we do provide is partial, but being embedded in F★, additional lemmas can always be developed interactively.

A type-and-effect directed frame rule. To control the placement of frames, we model the application of the frame rule as an effect. In particular, we formulate the system using a second related computation type that contains metavariables for an unsolved frame introduced by an application of the frame rule. This allows us to build a type-and-effect directed elaborator for Steel, inserting frames only at the leaves of a derivation, while proving that this strategy of leaf-framing is complete. Our approach enables interactions among several fragments of Steel modeled in an effect hierarchy, including atomic and ghost code.

Automating frame inference through AC-matching. Automatically applying the separation logic *frame* rule in the right places only solves half of the problem; generated frames must also be inferred. We prove that our type-and-effect system yields a unitriangular[1] system of constraints on the frame metavariables that can be solved by AC-matching (§4). Compared to standard AC-matching algorithms, we favor an incomplete but efficient and predictable non-backtracking approach that only solves problems with unique solutions, while accounting for equalities, theory reasoning, and existentially quantified ghost variables—the interaction with theory reasoning is enabled by the quintuple formulation.

[1]The term is borrowed from linear algebra: a unitriangular matrix is a triangular matrix such that all values on its main diagonal are 1.

Evaluation: Steel programs and libraries. To conclude, we evaluate how this work impacts the programmability and expressiveness of Steel programs. We first port several SteelCore libraries to Steel, and compare the quantity of proof annotations needed in both cases—the difference is marked, in some cases reducing the proof overhead by an order of magnitude. We then develop several new libraries, including linked structures of various flavors and highlight a verified implementation of mutable AVL trees, showing how quintuples enable reasoning independently about the tree shape and the balanced property of the mutable data structure. Next, we present proof idioms packaged as dependently typed libraries, including a library for disposable invariants. Using our invariant library, we verify a lock-free version of the standard Owicki-Gries parallel increment. We also show how locks and invariants can interoperate to verify the safety of an implementation of Michael and Scott's (1996) racy 2-locks concurrent queue. Finally, we develop a novel PCM-based encoding of 2-party dependently typed sessions, packaging our encoding as a library for message-passing concurrency on dependently typed channels.

Mechanization. Steel is entirely implemented in the F⋆ proof assistant, with proofs fully mechanized upon the SteelCore program logic. All of our code and proofs are open-source, and publicly available online[2].

2 BACKGROUND: F⋆, STEELCORE, AND VC GENERATION FOR SEPARATION LOGIC

Our goal is to shallowly embed the SteelCore logic as a DSL for programming in F⋆'s type theory. This involves exposing the proof rules of the logic as dependently typed combinators and instructing F⋆'s typechecker to apply those combinators during type inference and elaboration, coupling the program with a proof of its correctness. To make this process algorithmic we rely on F⋆'s user-defined effect system, which through our encoding, provides the needed structure. Rather than focus on the syntactic detail of our encoding, we present a more abstract view of the elaboration problem and our solution in standard mathematical notation, relating back to the specifics of F⋆ and SteelCore only when essential. Nevertheless, we start with a short primer on F⋆ and its syntax for the reader to refer back to periodically, if necessary.

2.1 A Primer on F⋆

F⋆ is a program verifier and a proof assistant based on a dependent type theory with a hierarchy of predicative universes (like Coq or Agda). F⋆ also has a dependently typed metaprogramming system inspired by Lean (Ebner et al. 2017) and Idris (called Meta-F⋆ (Martínez et al. 2019)) that allows using F⋆ itself to build and run tactics for constructing programs or proofs. More specific to F⋆ is its effectful type system, extensible with user-defined effects, and its use of SMT solving to automate some proofs. F⋆ syntax is roughly modeled on OCaml (val, let, match etc.) although there are many differences to account for the additional typing features.

Syntax: Binders, lambda, arrows, computation types. Binding occurrences b of variables take the form x:t, declaring a variable x at type t; or #x:t indicating that the binding is for an implicit argument. The syntax $\lambda(b_1) \ldots (b_n) \rightarrow t$ introduces a lambda abstraction, whereas $b_1 \rightarrow \ldots \rightarrow b_n \rightarrow c$ 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). We abbreviate _:unit{p} as squash p, the type inhabited by the unit value () only in a context where p is valid. 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. The c to the right of an arrow is a *computation type*. An example of a computation type is Tot bool, the type of total computations returning a boolean.

[2] https://zenodo.org/record/4768854

By default, function arrows have Tot co-domains, so, rather than decorating the right-hand side of every arrow with a Tot, the type of, say, the pure append function on vectors can be 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 '#'. We often omit implicit binders and write vec a m → vec a n → vec a (m + n) treating all unbound names as implicitly bound at the top.

Effectful computations. Users can define their own computation types, encapsulating various indexed-monad constructions to model computational effects. One example is the computation type from the introduction, SteelCore a p q, representing potentially divergent, concurrent, nondeterministic, and stateful computations. F*'s effect system isolates effectful code from its total core language to ensure that effects like divergence do not compromise soundness.

Notation for pairs. 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, and a dependent pair value is written (| e, f |). We use $x._1$ and $x._2$ for projecting the first and second components from both pairs and non-dependent pairs.

2.2 Verification Condition Generation for Separation Logic

Like any separation logic, SteelCore has rules for framing, sequential composition, and consequence, shown below in their first, most simple forms, where the type stc a p q represents a Hoare type with a:Type, p:slprop, and q:a → slprop. These proof rules are implemented in SteelCore as combinators with the following signatures:

```
let stc a p q = unit → SteelCore a p q (* represents { p } x:a { q x } *)
val frame (_:stc a p q) : stc a (p ∗ f) (λ x → q x ∗ f)
val bind (_:stc a1 p q') (_: (x:a1 → stc a2 (q' x) r)) : stc a2 p r
val conseq (_:stc a p' q') (_:squash (p −∗ p' ∧ q' −∗ q)) : stc a p q
```

Our goal is to shallowly embed Steel as a DSL[3] in F*, whereby Steel user programs are constructed by repeated applications of combinators like frame, bind and conseq. The result is a program whose inferred type is a judgment in the SteelCore logic, subject to verification conditions (VCs) that must be discharged, e.g., the second argument of conseq, squash (p −∗ p' ∧ q' −∗ q), is a proof obligation.

For this process to work, we need to make the elaboration of a Steel program into the underlying combinator language algorithmic, resolving the inherent nondeterminism in rules like Frame and Consequence by deciding the following: first, where exactly should Frame and Consequence be applied; second, how should existentially bound variables in the rules be chosen, notably the frame f; and, finally, how should the proof obligations be discharged.

The standard approach to this problem is to define a form of weakest precondition (WP) calculus for separation logic that strategically integrates the use of frame and consequence into the other rules in the system. Starting with Ishtiaq and O'Hearn's (2001) "backwards" rules, weakest precondition readings of separation logic have been customary. Hobor and Villard (2013) propose a ramified frame rule that integrates the rule of consequence with framing, while Iris' (Jung et al. 2018) "Texan triples" combine both ideas, integrating a form of ramified framing in the WP-Wand rule of its WP calculus. In the setting of interactive proofs, Texan triples are convenient in that every command is

[3]A note on terminology: From one perspective, Steel is not domain-specific—it is a general-purpose, Turing complete language, with many kinds of computational effects. But, from the perspective of its host language F*, Steel is a domain-specific language for proof-oriented stateful and concurrent programming.

constant	T	::=	$\mathsf{unit} \mid () \mid \mathsf{Type} \mid \mathsf{prop} \mid \mathsf{slprop} \mid \ldots$
term	e, t	::=	$x \mid T \mid \lambda x{:}t.\ e \mid e_1\ e_2 \mid x{:}t \to C \mid \mathsf{ret}\ e \mid \mathsf{bind}\ e_1\ x.e_2$
		|	$e_1 * e_2 \mid e_1 \mathrel{-\!\!*} e_2 \mid e_1 \wedge e_2 \mid \forall x.e \mid \ldots$
computation type	C	::=	$\mathsf{Tot}\ t \mid \{\ P \mid R\ \}\ y{:}t\ \{\ Q \mid S\ \}$
program	d	::=	$\mathsf{val}\ f\ (x : t)\ :\ C\ =\ e$

Fig. 2. Simplified syntax for Steel

always specified with respect to a parametric postcondition, enabling it to be easily applied to a framed and weakened (if necessary) postcondition.

Prior attempts at encoding separation logic in F* (Martínez et al. 2019) followed a similar approach, whereby a Dijkstra monad (Swamy et al. 2013) for separation logic computes weakest preconditions while automatically inserting frames around every function call or primitive action. However, Martínez et al. (2019) have not scaled their prototype to verify larger programs and we have, to date, failed to scale their WP-based approach to a mostly-automated verifier for Steel.

The main difficulty is that a WP-calculus for separation logic computes a single (often quite large) VC for a program in, naturally, separation logic. F* aims to encode such VCs to an SMT solver. However, encoding a separation logic VC to an SMT solver is non-trivial. SMT solvers like Z3 (de Moura and Bjørner 2008) do not handle separation logic well, in part because slprops are equivalent up to Associativity-Commutativity (AC) rewriting of $*$, and AC-rewriting is hard to encode efficiently in SMT. Besides, WP-based VCs heavily use magic wand and computing frames involves solving for existential quantifiers over AC terms, which again is hard to automate in SMT. Viper (the underlying engine of Chalice) does provide an SMT-encoding for a permission system with implicit dynamic frames that is equivalent to a fragment of separation logic (Parkinson and Summers 2012), however, we do not have such an encoding for SteelCore's more expressive logic. While some other off-the-shelf solvers for various fragments of separation logic exist (Brotherston et al. 2012; Iosif et al. 2014), using them for a logic like SteelCore's dependently typed, impredicative CSL is an open challenge.

Martínez et al. (2019) confront this problem and develop tactics to process a separation logic VC computed by their Dijkstra monad, AC-rewriting terms and solving for frame variables, and finally feeding a first-order logic goal to an SMT solver. However, this scales poorly even on their simpler logic, with the verification time of a program dominated by the tactic simply discovering fragments of a VC that involve non-trivial separation logic reasoning, introducing existentially bound variables for frames, solving them and rewriting the remainder of the VC iteratively.

Our solution over the next several sections addresses these difficulties by developing a verification condition generator for quintuples, and automatically discharging the computation of frames using a combination of AC-matching tactics and SMT solving, while requiring the programmer to write invariants and to provide lemmas in the form of imperative ghost procedures.

3 A TYPE-AND-EFFECT SYSTEM FOR SEPARATION LOGIC QUINTUPLES

In this section, we present our elaboration and VC generation strategy for Steel as a small idealized calculus. We transcribe the rules omitting some side conditions (e.g., on the well-typedness of some terms) when they add clutter—such conditions are all captured formally in our mechanization. As such, these rules are implemented as combinators in F*'s effect system and mechanically proven sound against SteelCore's logic in F*. In §4, we study the metatheory of the system from the perspective of completeness and the solvability of the constraints it produces.

3.1 Syntax

Figure 2 presents the syntax of a subset of the internal desugared, monadic language of Steel in F*. Our implementation supports the full F* language, including full dependent types, inductive types, pattern matching, recursive definitions, local let bindings, universes, implicit arguments, a module system, typeclasses, etc. This is the advantage of a shallow embedding: Steel inherits the full type system of F*. For the purposes of our minimalistic presentation, the main constructs of interest are slprops and computation types, though an essential preliminary notion is a memory, which we describe first.

Memories. A memory mem represents the mutable heap of a program and SteelCore provides an abstract memory model of mutable higher-order typed references, where each memory cell stores an element in the carrier type of a user-chosen PCM. We return to the specifics of the memory model in our case studies (§5). For now, it suffices to note that mem supports two main operations:

- disjoint (m0 m1: mem) : prop, indicating that the domains of the two memory maps are disjoint
- join (m0:mem) (m1:mem{disjoint m0 m1}) : mem, the disjoint union of two memories

What is an slprop? An slprop is a typeclass that encapsulates two things: an interpretation as a separation logic proposition and a self-framing memory representation called a *selector*. Specifically, it supports the following operations:

- An interpretation as an affine predicate on memories, namely interp (_: slprop) : mem ⟶ prop such that interp p m ∧ disjoint m m' ⟹ interp p (join m m'). We write fpmem (p:slprop) for a memory validating p, i.e., m:mem { interp p m }.
- A selector type, type_of (p:slprop) : Type
- A selector, sel (p:slprop) (m:fpmem p) : type_of p, with the property that sel depends only on the p fragment of m, i.e., (∀(m0:fpmem p) m1. disjoint m0 m1 ⟹ sel p m0 = sel p (join m0 m1)).
- slprops have all the usual connectives, including $*$, $-\!*$, ∧, ∨, ∀, ∃ etc. We observe that the selectors provide a form of linear logic over memory fragments as resources. For instance, the selector type for p $*$ q corresponds to a linear pair type_of p $*$ type_of q, while the selector type for p $-*$ q is a map from memories validating p $*$ (p $-*$ q) to the type of q. However, we do not yet exploit this connection deeply, except to build typeclass instances for $*$ and $-*$ and to derive from the double implication p $*\!-\!*$ q a bidirectional coercion on selectors.

It is trivial to give a degenerate selector for any slprop simply by picking the selector type to be unit. But, more interesting instances can be provided by the programmer depending on their needs. For example, given a reference r:ref a, the interpretation of ptr r : slprop could be that r is present in a given memory; type_of (ptr r) = a; and sel (ptr r) m : a could return the value of the reference r in m.

Computation types. The type Tot t is the type of total computations and is not particularly interesting. The main computation type is the quintuple $\{ P \mid R \}\ x{:}t\ \{ Q \mid S \}$, where

- P : slprop is a separation logic precondition
- R : fppred P, where R is a predicate on P's selector, i.e. fppred p = type_of p ⟶ prop, where the predicate is applied to sel p on the underlying memory.
- x : t binds the name x to the t-typed return value of the computation.
- Q : slprop is a postcondition, with x:t in scope.
- S: fppost P Q is an additional postcondition, relating the selector of P in the initial memory, to the result and the selector of Q in the final memory. It also has x:t in scope.
  fppost (p:slprop) (q:slprop) = type_of p ⟶ type_of q ⟶ prop

SteelCore's logic also provides support for a form of quintuples, but with one major difference: instead of operating on selectors, SteelCore uses memory predicates with proof obligations that

App

$$\frac{\Gamma \vdash e : \mathsf{Tot}\ t \qquad \Gamma \vdash f : x{:}t \to C}{\Gamma \vdash f\ e : C[e/x]}$$

Frame

$$\frac{\Gamma \vdash e : \{\ P \mid R\ \}\ y{:}t\ \{\ Q \mid S\ \}}{\Gamma \vdash_F e : \{\ P * ?F \mid \lambda(s_{p_0}, s_{f_0}).R\ s_{p_0}\ \}\ y{:}t\ \{\ Q * ?F \mid \lambda(s_{p_0}, s_{f_0})\ (s_{q_1}, s_{f_1}).S\ s_{p_0}\ s_{q_1} \wedge \mathit{seleq}\ ?F\ s_{f_0}\ s_{f_1}\ \}}$$

$$\text{let}\ \mathit{pre}\ \chi\ R_1\ S_1\ R_2\ ?a\ ?b = \lambda s_{p_1}.\ R_1\ s_{p_1} \wedge \forall x\ s_{p_2}.\ ?a \wedge (S_1[x/y]\ s_{p_1}\ (\chi\ s_{p_2}) \implies R_2\ s_{p_2} \wedge \forall z.\ ?b)$$
$$\text{let}\ \mathit{post}\ \chi_1\ \chi_2\ S_1\ S_2 = \lambda s_{p_1}\ s_q.\ \exists x\ s_{p_2}.\ S_1\ s_{p_1}\ s_{p_2} \wedge S_2\ (\chi_1\ s_{p_2})\ (\chi_2\ s_q)$$

Bind

$$\frac{\begin{array}{c}\Gamma \vdash_F e_1 : \{\ P_1 \mid R_1\ \}\ y{:}t_1\ \{\ Q_1 \mid S_1\ \} \qquad \Gamma, x{:}t_1 \vdash_F e_2 : \{\ P_2 \mid R_2\ \}\ z{:}t_2\ \{\ Q_2 \mid S_2\ \} \\ \Gamma, x{:}t_1, ?a \vDash_{tac} Q_1[x/y] \ast\!\!-\!\!\ast P_2 : \chi_1 \qquad \Gamma, x{:}t_1, z{:}t_2, ?b \vDash_{tac} Q_2 \ast\!\!-\!\!\ast ?Q : \chi_2 \qquad x \notin FV(t_2, ?Q)\end{array}}{\Gamma \vdash_F \mathsf{bind}\ e_1\ x.e_2 : \{\ P_1 \mid \mathit{pre}\ \chi_1\ R_1\ S_1\ R_2\ ?a\ ?b\ \}\ z{:}t_2\ \{\ ?Q \mid \mathit{post}\ \chi_1\ \chi_2\ S_1\ S_2\ \}}$$

Val

$$\frac{\begin{array}{c}\Gamma, x : t_1 \vdash_F e : \{\ P' \mid R'\ \}\ y{:}t_2\ \{\ Q' \mid S'\ \} \\ \Gamma, x{:}t_1, ?a \vDash_{tac} P \ast\!\!-\!\!\ast P' : \chi_p \qquad \Gamma, x{:}t_1, y{:}t_2, ?b \vDash_{tac} Q' \ast\!\!-\!\!\ast Q : \chi_q \\ \Gamma \vDash_{smt} \forall x\ s_p.\ (R\ s_p \implies ?a \wedge R'\ (\chi_p\ s_p)) \wedge (\forall y\ s_q.\ S'\ (\chi_p\ s_p)\ (\chi_q\ s_q) \implies ?b \wedge S\ s_p\ s_q)\end{array}}{\Gamma \vdash \mathsf{val}\ f\ (x : t_1)\ :\ \{\ P \mid R\ \}\ y{:}t_2\ \{\ Q \mid S\ \}\ =\ e}$$

Fig. 3. Core rules of Steel's type and effect system

they depend only on the appropriate part of memory. Steel's quintuples with selectors are proven sound in the model of SteelCore's "raw" quintuples, and the abstraction they provide yields useful algebraic structure while freeing the user from proof-obligations on the framing of memory predicates—proof-oriented programming at work!

3.2 VC Generation for Steel

Figure 3 presents selected rules for typechecking Steel programs. There are 3 main ideas in the structure of the rules.

First, there are two kinds of judgments $\vdash$ and $\vdash_F$. The $\vdash$ judgment applies to terms on which no top-level occurrence of framing has been applied. The $\vdash_F$ judgment marks terms that have been framed. We use this modality to ensure that frames are applied at the leaves, to effectful function calls only, and nowhere else. The application of framing introduces metavariables to be solved and introduces equalities among framed selector terms.

Second, the rule of consequence together with a form of framing is folded into sequential composition. Both consequence and framing can also be triggered by a user annotation in a val. Although Steel's separation logic is affine, Steel aims at representing and modeling a variety of concurrent programs, including programs implemented in a language with manual memory management, such as C. To this end, we need to ensure that separation logic predicates do not implicitly disappear. As such, our VC generator uses equivalence $\ast\!\!-\!\!\ast$ where otherwise a reader might expect to see implications ($-\!\!\ast$). Programmers are expected to explicitly *drop* separation logic predicates by either freeing memory or calling ghost functions to drop ghost resources.

Finally, the proof obligations corresponding to the VCs in the rules appear in the premises in two forms, $\vDash_{tac}$ and $\vDash_{smt}$. The former involves solving separation logic goals using a tactic, which

can produce auxiliary propositional goals to interact with SMT. The latter are SMT-encodeable goals—all non-separation logic reasoning is collected in the other rules eventually dispatched to SMT at the use of consequence triggered by a user annotation.

We now describe each of the rules in turn.

App. This is a straightforward dependent function application rule. F* internal syntax is already desugared into a monadic form, so we need only consider the case where both the function f and the argument e are total terms. Of course, the application may itself have an effect, depending on C. The important aspect of this rule is that it is a $\vdash$ judgment, indicating that this is a raw application—no frame has been added.

Frame. This rule introduces a frame. Its premise requires a $\vdash$ judgment to ensure that no repeated frames are added, while the conclusion is, of course, in $\vdash_F$, since a frame has just been applied. The rule involves picking a fresh metavariable $?F$ and framing it across the pre- and postconditions. The effect of framing of the memory postcondition S is particularly interesting: we strengthen the postcondition with seleq ?F s_{f_0} s_{f_1}, which is equivalent to sel ?F s_{f_0} = sel ?F s_{f_1}. We'll present this predicate in detail in §3.5.

Bind. The most interesting rule is Bind, with several subtle elements. First, in order to sequentially compose e_1 and e_2, in the first two premises we require $\vdash_F$ judgments, to ensure that those computations have already been framed. The third premise encodes an application of consequence, to relate the slprop-postcondition Q_1 of e_1 to the slprop-precondition P_2 of e_2. Strictly speaking, we do not need a double implication here, but we generate equivalence constraints to ensure that our constraint solving heuristics do not implicitly drop resources. The premise $\Gamma, x{:}t_1, ?a \vDash_{tac} Q_1[x/y] *\!\!-\!\!* P_2 : \chi_1$ is a VC that is discharged by a tactic and, importantly, $?a$ is a propositional metavariable that the tactic must solve. For example, in order to prove $\Gamma, x{:}t_1, ?a \vDash_{tac} (r \rightarrow u) *\!\!-\!\!* (r \rightarrow v)$, where the interpretation of $r \rightarrow u$ is that the reference r points to u, a tactic could instantiate $?a := (u = v)$, i.e., the tactic is free to pick a hypothesis $?a$ under which the entailment is true. The fourth premise is similar, representing a use of consequence relating the postcondition of e_2 to a freshly picked metavariable $?Q$ for the entire postcondition, again not dropping resources implicitly. A technicality is the use of the selector coercions χ_1, χ_2 witnessing the equivalences, which are needed to ensure that the generated pre- and postconditions are well-typed. Notice that this rule does not have an SMT proof obligation. Instead, we gather in the precondition the initial precondition R_1 and the relation between the intermediate post- and preconditions, S_1 and R_2. Importantly, we also include the tactic-computed hypotheses $?a$ and $?b$, enabling facts to be proved by the SMT solver to be used in the separation logic tactic. Finally, in the postcondition, we gather the intermediate and final postconditions.

Val. The last rule checks that the inferred computation type for a Steel program matches a user-provided annotation, and is similar to most elements of Bind. As shown by the use of the entailment $\vdash_F$, it requires its premise to be framed. The next two premises are tactic VCs for relating the slprop-pre- and postconditions, with the same flavor as before, allowing the tactic to abduct a hypothesis under which the goal is validated. Finally, the last premise is an SMT goal, which includes the freshly abducted hypotheses, and a rule of consequence relating the annotated pre- and postconditon to what was computed. Annotated computation types are considered to not have any implicit frames, hence the use of $\vdash$ in the conclusion.

As an example, typechecking the swap program presented in Fig. 1 proceeds as follows: The App rule is applied to each of the read and write function applications. Each application of App is followed by an application of Frame; this enables the composition of the function applications using

the Bind rule, whose premises require $\vdash_F$ judgments. Finally, an application of the Val rule ensures that the annotated Steel computation type is admissible for this swap program.

We prove in the supplement that the addition of the $\vdash_F$ modalities and the removal of a nondeterministic frame and consequence rule do not compromise completeness—we can still build the same derivations, but with the additional structure, we have set the stage for tactics to focus on efficiently solving slprop goals, while building carefully crafted SMT-friendly VCs that can be fed as is to F*'s existing, heavily used SMT backend.

3.3 Why it works: Proof-oriented Programming

In §2, we claimed that prior attempts at using a WP-based VC generator for separation logic in F* did not scale. Here, we discuss some reasons why, and why the design we present here fares better. As a general remark, recall that we want the trusted computed base (TCB) of Steel to be the same as SteelCore, i.e., we trust F* and its TCB, but nothing more. As such, our considerations for the scalability of one design over another will be based, in part, on the difficulty of writing efficient, untrusted tactics to solve various kinds of goals. Further, we aim for a Steel verifier to process an entire procedure in a single go and respond in no more than a few seconds or risk losing the user's attention. In contrast, in fully interactive verifiers, users analyze just a few commands at a time and requirements on interactive performance may be somewhat less demanding.

WP-based VCs are large and require non-reflective tactics. Separation logic provides a modular way to reason about memory, but properties about memory are only one of several concerns when proving a program. VCs for programs in F* contain many other elements: exhaustiveness checks for case analysis, refinement subtyping checks, termination checks, hypotheses encoding the definitions of let-bound names, and several other facts. In many existing F* developments a VC for a single procedure can contain several thousand logical connectives and the VC itself includes arbitrary pure F* terms. Martínez et al.'s (2019) tactics for separation logic process this large term, applying verifiable but slow proof steps just to traverse the formula—think repeated application of inspecting the head symbol of the goal, introducing a binder, splitting a conjunction, introducing an existential variable—even these simple steps are not cheap, since they incur a call to the unifier on very large terms—until, finally an slprop-specific part of a VC is found, split from the rest and solved, while the rest of the VC is rewritten into propositional form and fed to the SMT solver. Although F*'s relatively fresh and unoptimized tactic system bears some of the blame, tactics like this are inherently inefficient. Anecdotally, in conversations with some Iris users, we are told that running its WP-computations on large terms would follow a similar strategy to Martinez et al.'s tactics and can also be quite slow. Instead, high-performance tactics usually make use of techniques like proof-by-reflection (Gonthier et al. 2016), but a reflective tactic for processing WP-based VCs is hard, since one would need to reflect the entire abstract syntax of pure F* terms and write certified transformations over it—effectively building a certified solver for separation logic.

Structured VCs separate concerns. A proof-oriented programming mindset suggests that producing a large unstructured VC and trying to write tactics to recover structure from it is the wrong way to go about things. Instead, our proof rules are designed to produce VCs that have the right structure from the start, separating slprop reasoning and other VCs by construction. The expensive unification-based tactics to process large VCs are no longer needed. We only need to run tactics on very specific, well-identified sub-goals and the large SMT goals can be fed as is by F* to the SMT solver, once the tactics have completed.

Reflective tactics for slprop goals. Our tactics that focus on slprop implications are efficient because we use proof-by-reflection. Rather than reflect the entire syntax of F*, we only reflect the slprop

skeleton of a term, and then can use certified, natively compiled decision procedures for rewriting in commutative monoids and AC-matching to (partially) decide slprop equivalence and solve for frames. What calls we make to the unifier are only on relatively small terms.

3.4 Correspondence to our implementation

The two judgments $\vdash$ and $\vdash_F$ correspond to two new user-defined computation types in F⋆, namely Steel and SteelF. F⋆'s effect system provides hooks to allow us to elaborate terms written in direct style, let x = e in e' to bind_M_M' [[e]] (λx → [[e']]) when e elaborates to [[e]] with a computation type whose head constructor is M, and when the elaboration [[e']] has a type with a head constructor M'. This allows us to compose, say, un-framed Steel computations with framed SteelF computations by first applying the frame around the first computation and then calling the Bind rule. As such, we provide 4 binds for each of the combinations, rather than a single bind and a factored frame. The precise details of how this works in F⋆ are beyond the scope of this paper—Rastogi et al. (2020) describe the facilities offered by F⋆'s effect system which we use in this work.

Steel has two other kinds of computation types, for atomic computations and for ghost (proof-only) computation steps. We apply the same recipe to generate VCs for them, inserting frames at the leaves, and including consequence and framing in copies of Bind and Val used for these kinds of computations. Ultimately, we have six computation types, three of which are user-facing: Steel, SteelAtomic and SteelGhost. Behind the scenes, each of these has a framed counterpart introduced by the elaborator and eliminated whenever the user adds an annotation. These computation types are ordered in an effect hierarchy, allowing smooth interoperation between different kinds of computations; SteelAtomic computations are implicitly lifted to Steel computations when needed, while SteelGhost can be used transparently as either SteelAtomic or Steel.

3.5 An SMT-friendly Encoding of Selectors

Soundness of quintuples. We prove the soundness of our quintuples with selectors by reducing them to raw quintuples in SteelCore. In SteelCore quintuples $\{\ P \mid R\ \}\ x{:}t\ \{\ Q \mid S\ \}_{raw}$, we have R:mempred P and S:mempost P Q, capturing that R and S depend only on the P and Q fragments of the initial and final memories, respectively.

```
mempred (p:slprop) = f:(fpmem p → prop){∀ (m:fpmem p) (m':mem{disjoint m m'}). f m ⟺ f (join m m')
mempost (p:slprop) (q:slprop) = f:(fpmem p → mempred q){∀ (m0:fpmem p) (m0':mem{disjoint m0 m0'})
  (m1:fpmem q). f m0 m1 ⟺ f (join m0 m0') m1}
```

Thus every user annotation in SteelCore's raw quintuples comes with an obligation to show that the R and S terms depend only on their specified footprint—these relational proofs on specifications can be overwhelming, and require reasoning about disjoint and joined memories, breaking the abstractions that separation logic offers. In comparison, selector predicates are self-framing by construction: the predicates R and S can only access the selectors of P and Q instead of the underlying memory, which are themselves self-framing. By defining selector predicates as an abstraction on top of the SteelCore program logic, we thus hide the complexity of the self-framing property from both the user and the SMT solver.

A more efficient encoding of seleq. To preserve the modularity inherent to separation logic reasoning when using selector predicates, the postcondition of the Frame rule previously presented contains the proposition seleq ?F s_{f_0} s_{f_1}, capturing that sel ?F s_{f_0} = sel ?F s_{f_1}.

Using this predicate, the SMT solver can derive that the selector of any slprop contained in the frame is the same in the initial and final memories, leveraging the fact that, for any m:fpmem (p ∗ q),

sel (p ∗ q) m = (sel p m, sel q m). But as the size of the frame grows, this becomes expensive; the SMT solver needs to deconstruct and reconstruct increasingly large tuples.

Instead we encode seleq as the conjunction of equalities of the *atomic* slprops selectors contained in the frame, where an *atomic* slprop does not contain a ∗. For instance, p and q are the atomic slprops contained in p ∗ q Our observation is that most specifications pertain to atomic slprops; the swap function presented in the introduction for instance is specified using the selectors of ptr r1 and ptr r2, instead of (ptr r1 ∗ ptr r2).

Once the frame has been resolved using the approach presented in §4, generating these equalities is straightforward using metaprogramming; we can flatten the frame according to the star operator, and generate the conjunction of equalities to pass to the SMT solver.

Limitations of selectors. Selectors can alleviate the need for existentially quantified ghost variables; the value stored in a reference for instance can be expressed as a selector, decluttering specifications. But, not all slprops have meaningful selectors, nor do we expect that they should. For example, when using constructions like PCMs to encode sharing disciplines, it is not always possible to define a selector that returns the partial knowledge of a resource. However, as illustrated §5.2, when applicable, selectors can significantly simplify specifications and proofs.

4 AUTOMATICALLY DISCHARGING STEEL VERIFICATION CONDITIONS

In this section, we show how to discharge separation logic VCs generated during the elaboration of Steel programs; namely, how to solve frame metavariables $?F$ and separation logic entailments $\vDash_{tac}$.

We start by presenting a quick overview of our methodology on the simple (though artificial) example below:

```
val write (r:ref a) (x:a) : Steel unit (ptr r) (λ _ → ptr r)
let two_writes (r1 r2:ref int) : Steel unit (ptr r1 ∗ ptr r2) (λ _ → ptr r1 ∗ ptr r2) = write r1 0; write r1 1
```

For clarity, we will omit the $\vDash_{smt}$ constraints when typechecking this program; they are irrelevant to this example. First, the App rule from Fig. 3 is applied to both writes. The function applications are then sequentially composed using the Bind rule. This rule requires $\vdash_F$ computations as premises; frames $?F_1$ and $?F_2$ are automatically inserted by applying the Frame rule to each application. When composing sequentially, a fresh metavariable ?Q as well as the constraints ptr r1 ∗ $?F_1$ ∗−∗ ptr r2 ∗ $?F_2$ and ptr r2 ∗ $?F_2$ ∗−∗ ?Q are generated. Finally, the Val rule ensures the inferred type matches the user's signature, generating the constraints ptr r1 ∗ ptr r2 ∗−∗ ptr r1 ∗ $?F_1$ and ?Q ∗−∗ ptr r1 ∗ ptr r2.

Determining whether two terms containing an arbitrary number of metavariables can be unified up to associative-commutative rewriting is a hard problem (Fages 1984). Constraints for Steel programs fit this description, using only one AC-function: the separation logic ∗. Our observation is that we can instead reduce constraint solving in Steel to a simpler problem, namely, AC-matching. By solving constraints in a particular order, it is possible to only consider constraints that contain at most one metavariable. This happens when considering constraints generated by a linear traversal, in either forward or backward program order. In our example, going forward we would first solve $?F_1$ in ptr r1 ∗ ptr r2 ∗−∗ ptr r1 ∗ $?F_1$, then $?F_2$ in ptr r1 ∗ $?F_1$ ∗−∗ ptr r2 ∗ $?F_2$ ($?F_1$ having been solved previously), and finally ?Q through ptr r2 ∗ $?F_2$ ∗−∗ ?Q. Once we reach the last constraint ?Q ∗−∗ ptr r1 ∗ ptr r2, ?Q has already been solved and checking AC-equivalence is straightforward.

We first formalize this intuition in §4.1, proving that the rules presented in Fig. 3 ensure that a scheduling suitable for AC-matching exists for any well-typed Steel program. We then present our approach to AC-matching in the context of our auto-active prover. Compared to complete, but expensive algorithms for AC-matching previously proposed (Kapur and Narendran 1987),

our algorithm is efficient, providing quick feedback to the programmer when unification fails. In exchange, it is incomplete; when results are inconclusive, the programmer can provide hints and annotations to help the unifier make progress.

4.1 A Unitriangular AC-Matching Problem

In this section, we show that the constraints generated in Fig. 3 can be split into a *unitriangular* set and an unrestricted set of equations. This property ensures that a scheduler can always pick an AC-matching constraint, while guaranteeing progress and termination.

For notational purposes, we write the metavariables ?F, ?Q etc. as variables ?u in this section. We also write $\Gamma \vdash e : C \mid U; \mathcal{X}$ to mean that a typing judgment $\Gamma \vdash e : C$ from Fig. 3 generates the set of metavariables U and the set of $\vDash_{tac}$ constraints over them denoted by $\mathcal{X}$ (similarly for $\vdash_F$ judgments). We begin by defining a unitriangular system of equations. Such systems of equations can be represented using unitriangular matrices, i.e. triangular matrices where all elements in the main diagonal are 1.

Definition 4.1 (Unitriangular system of equations). An ordered set of metavariables $U = \{?u_i\}_{i\in[1,n]}$ and an ordered set of equations $\mathcal{X} = \{\mathcal{X}_i\}_{i\in[1,n]}$ form a unitriangular system of equations if

(1) $\forall i \in [1, n]$. $?u_i$ occurs exactly once in $\mathcal{X}_i$, and
(2) $\forall i, j \in [1, n]$. $?u_j$ does not occur in $\mathcal{X}_i$ if $j > i$.

Our main theorem is then as follows:

Theorem 4.2. *If* $\Gamma \vdash e : \{ P \mid R \}\ z{:}t\ \{ Q \mid S \} \mid U; \mathcal{X}$ *then* $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ *and there exists an ordering of* U *and* $\mathcal{X}_1$ *s.t.* $(U, \mathcal{X}_1)$ *is unitriangular.*

To prove theorem 4.2, we need to reason about the metavariables and constraints generated by the framed computations. To that end, we work with a notion of *once-removed*-unitriangular system of equations. Intuitively, given a unitriangular system of equations, we can obtain a once-removed-unitriangular system of equations by removing the first constraint.

Theorem 4.3. *If* $\Gamma \vdash_F e : \{ P \mid R \}\ z{:}t\ \{ Q \mid S \} \mid U; \mathcal{X}$ *then* $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ *and there exists an ordering of* U *and* $\mathcal{X}_1$ *s.t.* $(U, \mathcal{X}_1)$ *is once-removed-unitriangular with exactly one occurrence of* $?u_1$ *in* Q.

The proofs for these theorems rely on the following auxiliary lemmas.

Lemma 4.4. *If* $\Gamma \vdash e : \{ P \mid R \}\ z{:}t\ \{ Q \mid S \}$, *then* P *and* Q *do not contain any metavariables.*

Lemma 4.5. *If* $\Gamma \vdash_F e : \{ P \mid R \}\ z{:}t\ \{ Q \mid S \}$, *then* P *and* Q *each contain exactly one occurrence of a metavariable.*

The proofs for Theorem 4.2 and Theorem 4.3 proceed by simultaneous induction on the two typing derivations. The intuition behind this proof is that we can construct the unitriangular system by traversing the derivation backwards, starting from the postcondition. The choice is arbitrary but convenient for the proof—one could also structure the proof to go forwards instead.

A constraint scheduler for Steel. The unitriangular shape of the set of equations allows us to solve the $?u_i$ sequentially while finally verifying that the solutions are consistent with the equations C_i. In practice, when typechecking a Steel program, we do not reorganize the generated constraints to extract a unitriangular system. We instead implement a simpler scheduling, solving the first remaining constraint which contains only one occurence of a metavariable. The existence of the unitriangular system ensures the progress and termination of this scheduling.

4.2 Solving AC-Matching Instances

In the previous sections, we showed how we could schedule equations to be solved sequentially, so that each scheduled equation contains at most one frame metavariable. In this section, we present the last missing piece of the puzzle: how to actually solve such an equation.

Consider below a simplified grammar for a subset of slprops.

$$v \quad ::= \quad c \mid ?v \qquad t \quad ::= \quad \text{emp} \mid f\ v \mid ?u \mid t * t$$

We assume the existence of a set of constants c and uninterpreted function symbols f. Terms t can be the unit emp, a function f whose argument is either a constant or metavariable $?v$, an slprop-metavariable $?u$ (typically the frame variable), or a separating conjunction of two terms. The other connectives are uninterpreted by our AC-matching solver. One main point of interest here is that we have two sorts of metavariables, $?v$ variables may arise due to implicit arguments in a program and, unlike the $?u$ variables, may have a type other than slprop.

When considering a scheduled equation, we will denote the frame metavariable $?u$, and assume without loss of generality that it is on the left-hand-side of the equality. Similarly to AC-matching algorithms in the literature, we consider flattened representations of both sides of the equation.

Our algorithm proceeds by trying to match each symbol on the left with a rigid head symbol (c or f) with a term on the right, and finally sets the metavariable $?u$ to the conjunction of the remaining terms on the right once matching is over, or the unit emp if no such term is left. Let us first present the simple case where all terms different from $?u$ are constants. If a constant on the left has no counterpart on the right, then the equation cannot be solved, raising a unification error. We present below some simple examples to illustrate—constants c_i, c_j are distinct and non-matchable.

- $c_1 * ?u = c_2 * c_1 * c_3$. The c_1 on both sides match and are removed. $?u$ is set to the conjunction of leftover terms on the right side, i.e. $c_2 * c_3$
- $c_1 * ?u = c_1$. The c_1 on both sides match and are removed. There is no leftover term on the right side, so $?u$ is set to the unit emp
- $c_1 * ?u = c_2$. A unification error is raised, since c_1 cannot be matched with any term on the right side.

Now consider the equation f ?v * f v2 * ?u = f v2 * f v1. By applying naively the matching algorithm presented previously, the first term on the left, f ?v, would be matched with the first term on the right, f v2. This would prevent the second term on the left, f v2 from being matched with the remaining terms on the right, which in turn would return a unification error to the programmer, pointing to f v2 not being matched, and leading to confusion since a matching term does exist on the right side.

A natural solution to this problem would be backtracking, attempting to match f ?v with a different term on the right side. As previously stated, this is a solution we wish to avoid; the cost of backtracking can become prohibitive, and hinder the interactivity required for program verification. We instead only match a left-hand-side term t_l if there is a *unique* term on the right that it can be unified with. If this is not the case, we delay the matching of t_l and attempt to match the rest of the terms on the left side. If no progress was made once we retry matching t_l, an error message prompts the programmer to instantiate more implicit arguments.

Again, we present below several examples illustrating the behaviour of our algorithm.

- f v1 * f ?v * ?u = f v2 * f v1. We first attempt to match f v1 with a term on the right. There is a unique solution, so matching is performed. We then attempt to match the second term, f ?v. Only f v2 is left on the right, so there is a unique solution and we can set ?u to emp.
- f ?v * f v1 * ?u = f v2 * f v1. We first attempt to match f ?v with a term on the right. Both terms on the right are valid solution, so we delay this matching. We then attempt to match f v1,

which has a unique solution. Matching on all terms on the left different from ?u has been attempted and progress has been made, we retry with delayed term. f ?v now has a unique solution on the right side. We finally set ?u to emp.

- f ?v * f ?v' * ?u = f v1 * f v2. Both f ?v and f ?v' have several solutions on the right side. Since no progress is made after attempting matching for terms on the left side, an error is raised.

4.3 Cooperating with the SMT solver

Our AC-matching algorithm is entirely implemented as an F⋆ tactic. It relies on the F⋆ unifier to determine whether two terms can be matched, thus solving relevant metavariables when matching occurs. As in other systems, F⋆ tactics are not trusted—the terms they generate are guaranteed to be well-typed, thus ensuring soundness of the decision procedure.

Since the decision procedure for AC-matching is not trusted, there is no need to restrict its complexity. As such, our decision procedure is designed to be easily extensible with additional heuristics and user customization. In this section, we present one such extension, which enables equality rewriting during AC-matching by querying the SMT solver.

Consider the simple case where we wish to solve the equation f b = f true. Here, the unification procedure presented so far would fail even if the equality b = true is valid, due to, say, a control flow hypothesis. The only solution would be for a programmer to manually trigger a rewrite by calling a ghost procedure. Instead, we implement heuristic abduction of equalities (the ?a and ?b in $\vDash_{tac}$ in Fig. 3) in our tactic that allows us to match f b = f true via a SMT provable equality. This is powerful and allows an interplay between arbitrary theories and AC-matching, allowing, for example, our algorithm to match f (x - x) = f 0 or f (2*x) = f (x + x), i.e., a kind of AC-matching modulo theories.

However, this is in tension with the basic structure of our AC-matching algorithm. If ever $c_i = c_j$ is possible with theory reasoning, even the most basic steps of our matching algorithm will always fail, since there is no unique solution even to a simple problem like $c_i * ?u = c_i * c_j$. Hence, deciding which equalities to abduct and delegate to SMT requires program-specific knowledge, which we allow the programmer to configure.

When defining a separation logic predicate, the programmer can annotate some arguments to mark them as candidates for SMT-based rewritings. Consider for instance the standard separation logic predicate pts_to r v, which indicates that the reference r stores the value v. When reasoning about the functional correctness of a program using this predicate, equalities on the value v are common while equalities on the reference r itself are rarer. As such, a programmer could decide to mark values v as candidates for SMT-based rewriting using the smt_rewrite attribute as follows, but not references r in pts_to predicates.

```
val pts_to (r:ref a) (#[@@ smt_rewrite] v:a) : slprop
```

Thus, unifying pts_to r v1 = pts_to r v2 would automatically succeed if the SMT solver can prove v1 = v2, but unifying pts_to r1 v = pts_to r2 v would require a manual rewrite when r1 = r2. A library designer can make some of these choices once and for all, so that all clients benefit from smart equality abduction and our library for references with fractional permissions does indeed mark the value and the permission as abduction candidates.

When the AC unifier cannot make progress anymore, ideally after having matched some of the left-hand-side terms that had a unique solution on the right-hand-side, it retries a similar unification while generating equalities to be discharged by SMT. These equalities propagate to the SMT solver using the $?a$ and $?b$ abduction variables from Fig. 3. When falling back on the SMT solver is not necessary, these metavariables are set to $\top$, ensuring that no metavariable is left unsolved. Similarly to the main algorithm, solutions that are not unique are not accepted, so as to provide accurate error reporting to the programmer.

- pts_to r v1 ∗ pts_to r v2 = pts_to r v1 ∗ pts_to r v3. The decision procedure first attempts exact matching, and removes pts_to r v1 from both sides. It is then left with two terms that cannot be unified, and falls back on SMT rewritings. Since the value argument of pts_to has been marked as candidate for SMT-based rewriting, the AC unifier queries the SMT solver to check whether v2 = v3.
- pts_to r v1 ∗ pts_to r v2 = pts_to r v3 ∗ pts_to r v4. No exact matching is possible, the AC unifier fallbacks directly on SMT rewritings. Since both v1 and v2 could possibly be rewritten into v3 or v4, the unicity of the solution cannot be guaranteed. The AC unifier fails, and asks the programmer to provide manual rewrites to disambiguate.

5 STEEL IN ACTION

In this section, we present several verified libraries in Steel, evaluating the various styles of program proof it enables and the level of proof automation and control it provides.

As regards expressiveness, being embedded in F★, Steel offers a large variety of styles of programming with a dependently typed CSL, ranging from Viper-style permission accounting with implicit dynamic frames, to more dependently typed libraries for invariants, concurrent data structures and message-passing protocols.

As regards automation, we find that our hybrid tactic- and SMT-based program verifier eliminates *all* mundane proofs step related to framing. Equality abduction in our tactics automatically delegates extensional conversions of slprops to SMT in many though not all cases. Programmers must still invoke lemmas to roll and unroll recursive predicates, to trigger certain rewritings, and also call ghost procedures for proof steps and to maintain stateful invariants. We believe that the overhead is comparable to other SMT-based program verifiers, though in comparison to prior developments in F★ (verifying sequential imperative programs), Steel proofs are significantly more abstract due to the more powerful logic and the automated support for framing.

To be fair, not all is rosy: we still have work to do to improve error reporting and there are many opportunities for better integration with F★'s emacs-based IDE—the verifier has lots of information about the slprops available at every program point, and surfacing this to the user as a tooltip would help boost productivity significantly. Yet, all the main technical pieces are in place for us to address these user issues.

5.1 Basic Concepts

Most of the prior sections do not depend much on the specifics of the SteelCore logic itself, but here, we need to introduce several concepts before the case studies themselves.

Steel inherits from SteelCore several basic concepts, however we have re-packaged SteelCore's libraries for easier use. In the process, we were able to simplify the proofs of many of these libraries, reducing in some cases the proof overhead by an order of magnitude. We review the key concepts here and, where relevant, also report on our experience upgrading the libraries.

Reference and executable semantics. Steel's semantics is formally based on a definitional interpreter that provides a non-deterministic interleaving of actions. An *action* is a single step of stateful computation, reading or writing to a memory mem and returning a result. While our interpreter provides a reference semantics, we do not intend to actually run Steel programs on it. Although we have yet to do so, we intend to reuse F★'s existing backends to OCaml, F#, and C, to execute Steel programs efficiently. As such, these backends are also part of F★'s and Steel's TCB. All our code is written with extraction in mind, e.g., all code that should be erased is indeed marked for erasure and checked as such by F★.

Erased types. The type erased (t:Type) in F⋆ describes a computationally irrelevant value which will be extracted by F⋆ to (). These types are useful to describe ghost values for use in specifications only, while ensuring that passing such values around does not incur any runtime cost. The function hide (v:t) : erased t builds an erased version of t, while its inverse, reveal (erased t) : t, is checked by F⋆ to be used only in code that is computationally irrelevant.

Three kinds of computations: SteelAtomic, SteelGhost and Steel. As mentioned in §3.4, we have three related user-facing computation types with the signatures below, where pre p = type_of p ⟶ prop and post p a q = type_of p ⟶ x:a ⟶ type_of (q x) ⟶ prop:

- SteelAtomic (a:Type) (i:inames) (p:slprop) (q:a ⟶ slprop) (r:pre p) (s:post p a q)
- SteelGhost (a:Type) (i:inames) (p:slprop) (q:a ⟶ slprop) (r:pre p) (s:post p a q)
- Steel (a:Type) (p:slprop) (q:a ⟶ slprop) (r:pre p) (s:post p a q)

SteelAtomic and SteelGhost have the same signature and carry an additional index (i:names) which we will explain shortly. SteelAtomic is used to classify computationally relevant code whose effects on memory are atomic, e.g., an atomic compare-and-set (CAS) instruction would have type SteelAtomic. SteelGhost describes code that has no observable computational effect, e.g., this could involve a proof step such as unrolling a recursive predicate, calling a lemma, or reading, writing or allocating to ghost state. Steel is the general purpose computation type for Steel code, and involves a mixture of pure computations, multiple atomic steps composed in sequence or parallel, and ghost code—SteelAtomic and SteelGhost are implicitly lifted to Steel. Parallel composition in Steel is implemented by the following combinator

```
par (f : unit ⟶ Steel a p q r s) (g: unit ⟶ Steel a' p' q' r' s') : Steel (a & a') (p ∗ p') (λ (x,x') ⟶ q x ∗ q' x')
    (requires λ(t,t') ⟶ r t ∧ r' t')
    (ensures λ(t,t') (x,x') (u,u') ⟶ s t x u ∧ s' t' x' u')
```

In case the r:pre p or q:post p a q are trivial we simply omit it; otherwise, as in par above, we tag the selector predicates with requires and ensures to improve readability.

Memory. In its most basic form, the memory mem of a Steel program contains a map from abstract typed references pref (a:Type) (p:pcm a) to values of type a, where p:pcm a is some partial commutative monoid (PCM) over the carrier type a. By choosing suitable PCMs, Steel's libraries provide various flavors of derived reference types, the most commonly used of which are references with fractional permissions, with the signatures below.

- ref t, is a reference to a t-typed value
- f : frac is an erased, real-valued fraction between 0 and 1.
- $r \overset{f}{\mapsto} v$: slprop asserts ownership of an f-fraction of r pointing to v, while $r \mapsto v = r \overset{1.0}{\mapsto} v$.
- $r \overset{f}{\mapsto}$ is equivalent to $\exists v.\ r \overset{f}{\mapsto} v$, with the selector type type_of (ptr r f) = t, when r:ref t.
- pure p is equivalent to emp in a context where the proposition p is valid.

Additionally, we have ghost references, ghost_ref (t:Type) which refer to erased t values in memory—both the references and the values they point to are computationally irrelevant. The slprop for ghost references is written $r \overset{f}{\dashrightarrow} v$, but is otherwise identical to $\mapsto$. In §5.6 we show how to use references with other PCMs.

Invariants. Any slprop can at some point in a computation be designated an *invariant*, and is from then on enforced by the logic to be maintained by all susbequent computation steps. Invariants in Steel are closely related to invariants in Iris. The main constructs are shown below:

	SteelCore	Steel	Total file size (LoC, SteelCore implementation)
SpinLock	34	13	150
Fork/Join	33	9	130
Simplex	340	70	933

Table 1. Comparison of the number of separation logic lemma calls in SteelCore and Steel.

- inv (p:slprop) : Type, is the type of an invariant enforcing p. Note, inv p is a value that can be freely duplicated and shared among threads. It represents a kind of token witnessing the validity of p.
- Invariants are named, with name (i:inv p) : iname, and inames = set iname.
- new_inv (p:slprop) : SteelGhost (inv p) i p (λ_ → emp), consumes the initially valid p:slprop and returns a new token for it.
- Invariants can be opened and restored in atomic code using the following combinator, which states that f can assume p_i and restore it in an atomic step and return x, while also transforming p to q x. The index u:inames is used to ensure that f does not itself internally open i, which would be unsound. A similar combinator, with_inv_ghost, allows using and restoring an invariant in SteelGhost code.

  ```
  with_inv (i:inv p_i) (f: unit → SteelAtomic a u (p_i ∗ p) (λ x → p_i ∗ q x)) : SteelAtomic a (name i ⊎ u) p q
  ```

Improving other SteelCore libraries. Based on the core constructs above, SteelCore provided libraries for spin locks, fork/join parallelism, and message passing on unidirectional channels, or simplex channels. We reimplemented the proofs of those libraries while retaining their specifications. As presented in Table 1, the improved automation in SteelCore shrunk the proofs dramatically, e.g., our proof of simplex channels is several times shorter than the previous proof, which, like swap from §1, was previously utterly overwhelmed by manual proof steps for framing and slprop rewriting. The new proofs use the same invariants as in SteelCore, but are thankfully significantly more maintainable. In §5.6, we present a new PCM-based construction for 2-party session types on bidirectional channels, subsuming SteelCore's simplex channel library. The spin lock library provides the following idiomatic interface, which we use in several examples later in the section.

```
val lock (p:slprop) : Type (* type of a lock, a firstclass value, locks can even be stored in the mem *)
val new_lock (p:slprop) : Steel (lock p) p (λ _ → emp) (* give up the slprop p to create a new lock *)
val acquire (l:lock p) : Steel unit emp (λ _ → p) (* acquire the lock and gain p *)
val release (l:lock p) : Steel unit p (λ _ → emp) (* release the lock and give up p *)
```

5.2 Balanced trees: Selectors at work

As a first case study, we present a verified implementation of self-balancing AVL trees. To specify this implementation, our first step is to define a tree : slprop capturing the essence of a mutable tree. In the following code, Spec is the name of the F⋆ module containing a standard, pure specification of binary trees, represented as an inductive datatype whose constructors are Leaf and Node data left right.

We begin by setting up the various types and representation invariants. The definitions of tree nodes and trees are mutually recursive: a tree node is a record containing a value data, as well the left and right subtrees, while a tree is a pointer to a tree node.

```
type node (a: Type) = { data: a; left: t a; right: t a }
and t (a: Type) = ref (node a) (* The type of the binary linked trees *)
```

```
(* A recursive predicate for binary trees *)
let rec tree_interp' (ptr: t a) (n: Spec.tree (node a)) = match n with
  | Spec.Leaf → pure (ptr = null) (* Leaves are represented by null pointers *)
  | Spec.Node data left right → tree_interp' data.left left ∗ tree_interp' data.right right ∗ ptr ↦ data
let tree_interp (ptr:t a) = ∃n. tree_interp' ptr n (* We existentially quantify over the spec tree *)
(* The selector only keeps the data in the nodes, returning a Spec.tree a *)
val tree_sel (ptr:t a) (m:fpmem (tree_interp ptr)) : Spec.tree a
(* We finally collect the different components to define the slprop tree, indexed by the pointer to the root *)
let tree (ptr:t a) : slprop = { interp = tree_interp ptr; type_of = Spec.tree a; sel = tree_sel ptr }
```

Note that the interpretation of tree is an existentially quantified, recursive predicate. As mentioned previously, we expect to make ghost procedure calls to manipulate quantifiers and to roll and unroll recursive predicates. The signature of roll_tree is below.

```
val roll_tree (root: t a) (left: t a) (right: t a) : SteelGhost unit u
  (tree left ∗ tree right ∗ ptr ↦ root) (λ _ → tree root)
  (requires (λ s → s.[ptr].left == left ∧ s.[ptr].right == right))
  (ensures (λ s _ s' → s'.[ptr] == Spec.Node s.[ptr].data s.[left] s.[right]))
```

Proofs of such lemmas are a bit mechanical—we open the existentials for tree, instruct the F⋆ normalizer to reduce the recursive function tree_interp' and then fold it back to introduce tree_interp, then pack the existential, and return. In the future, we believe some of this boilerplate can be reduced through metaprogramming. Now, with our roll and unroll lemmas in hand, we can turn to the code itself.

Using tree selectors, we can define concise specifications operating on pure F⋆ trees. For example, the specification for height relates the returned value x to the height of the F⋆ tree returned by tree_sel, while ensuring that the function did not modify the tree.

```
let rec height (ptr:t a) : Steel int (tree ptr) (λ _ → tree ptr) (requires λ_ → ⊤)
    (ensures λs x s' → s.[ptr] == s'.[ptr] ∧ Spec.height s.[ptr] == x)
  = if is_null ptr then (unroll_leaf ptr; 0) else (
      let node = unroll_tree ptr in
      let hleft = height node.left in
      let hright = height node.right in
      roll_tree ptr node.left node.right;
      if hleft > hright then (hleft + 1) else (hright + 1))
```

With the exception of three ghost calls that roll and unroll the definition of the slprop, the code is fairly canonical and the proof is automated by the hybrid of tactics and SMT in about a second.

One of the main benefits of this approach is that tree can be used as a basis to define more involved notions of trees at a minimal cost. For instance, one can define mutable AVL trees as a pure predicate over the selector of a tree, without needing to modify the representation invariant. Under this requirement, we can specify a self-balancing insertion operation, insert_avl, which is parameterized by a comparison function cmp needed to perform a binary search. This operation builds on the same memory layout for the tree, but with a different logical layer over it (e.g AVL balancing) . The Spec.is_avl function checks that the tree meets our specification for an AVL, e.g., every subtree is balanced and the tree is a binary search tree.

The full Steel implementation and proof of insert_avl, shown below, follows the flow of a texbook binary search tree insertion; it creates a new node if the tree is empty, or recursively inserts v in

the correct subtree if not before finally rebalancing the tree. All verification conditions related to the shape of the tree are discharged automatically by SMT, while separation logic VCs only require minimal user interaction; calling stateful lemmas such as roll_tree in a few specific, predictable places is sufficient. The procedure is checked in about 6 seconds.

```
let rec insert_avl (cmp:Spec.cmp a) (ptr: t a) (v: a) : Steel (t a) (tree ptr) (λ ptr' → tree ptr')
  (requires λs → Spec.is_avl cmp s.[ptr])
  (ensures λs ptr' s' → Spec.insert_avl cmp s.[ptr] v == s'.[ptr'])
  = if is_null ptr then (unroll_leaf ptr; (* unroll the tree slprop *)
      let node = {data = v; left = ptr; right = null} in let new_tree = alloc node in
      roll_leaf (); roll_tree new_tree ptr null; new_tree (* roll the tree slprop and return tree *)
    ) else (
      let node = unroll_tree ptr in if cmp node.data v ≥ 0 then (
        let new_left = insert_avl cmp node.left v in
        let new_node = {data = node.data; left = new_left; right = node.right} in
        write ptr new_node; roll_tree ptr new_left node.right; rebalance_avl cmp ptr
    ) else (
      let new_right = insert_avl cmp node.right v in
      let new_node = {data = node.data; left = node.left; right = new_right} in
      write ptr new_node; roll_tree ptr node.left new_right; rebalance_avl cmp ptr))
```

5.3 A Library of Disposable Invariants

To illustrate how common proof idioms can be packaged as dependently typed libraries in Steel, we present a library for disposable invariants, which are very similar to Iris' cancellable invariants (Jung et al. 2018). Disposable invariants, like invariants, package an slprop and provide a similar with_invariant combinator to work with the slprop in the atomic code.

The main novelty of disposable invariants is that, similar to locks, they may be reclaimed, thereby returning the underlying slprop back to the context. But since disposable invariants are still computationally irrelevant, unlike locks, they don't have a computational overhead.

We present the library interface below. The main slprop provided by the library is active perm i, where perm is a permission over the disposable invariant i. share and gather may be used to split and collect the invariant permissions, while dispose enforces that the caller must have full permission over the invariant. We elide the with_invariant combinator, its signature is similar to the signature shown in Section 5.1 with active perm i in the pre- and postcondition of the combinator.

```
val inv (p:slprop) : Type (* the type of disposable invariants *)
val name (i:inv p) : iname
val active (f:perm) (i:inv p) : slprop
val new_inv (p:slprop) : SteelGhost (inv p) _ p (λ i → active 1.0 i) (* consumes p *)
val share (i:inv p) : SteelGhost unit _ (active perm i) (λ _ → active perm/2 i ∗ active perm/2 i)
val gather (i:inv p) : SteelGhost unit _ (active perm0 i ∗ active perm1 i) (λ _ → active (perm0 + perm1) i)
val dispose (i:inv p{not (name i ∈ u)}) : SteelGhost unit u (active 1.0 i) (λ _ → p) (* destroys i, recovers p *)
```

The implementation of the library packages a normal invariant with a ghost_ref bool. Depending on the value that the reference points to (true or false resp.), this invariant either encapsulates the underlying slprop p or emp. Thus, a disposable invariant starts with the ghost ref pointing to true, while disposing it sets the value of the ref to false, returning the slprop p back to the context.

```
let inv p = r:ghost_ref bool & Steel.Memory.inv (∃ (b:bool). r -0.5-> b ∗ (if b then p else emp))
let active perm i = i.1 -perm/2-> true
```

5.4 Parallel Increment *à la* Owicki-Gries with Disposable Invariants

For our next case study, we present a Steel implementation of the Owicki-Gries counter (Owicki and Gries 1976) using disposable invariants. In this example, the main thread spawns two worker threads, both of which increment a shared counter by 1. The goal is to prove in the main thread that once the worker threads finish, the value of the shared counter is incremented by 2. Owicki and Gries's solution is for each thread to use a ghost reference to track its contribution to the counter, with an invariant that the value of the shared counter is equal to the sum of the values of the two contribution variables. Since each ghost reference is incremented by 1, the main thread can now prove that the assertion about the counter value holds. Here's the invariant:

```
let og' (ctr:ref int) (r1 r2:ghost_ref int) w → r1 -0.5-> (fst w) ∗ r2 -0.5-> (snd w) ∗ ctr ↦ (fst w + snd w)
let og ctr r1 r2 : slprop = ∃w. og' ctr r1 r2 w
```

Since the invariant needs to only be in place while the threads are active, this is a good candidate for a disposable invariants, so long as the threads use only atomic instructions to increment the counter. The main thread creates the two ghost references and splits their permissions. It then creates the disposable invariant sealing og with it. Next, it splits the disposable invariant and spawns the two worker threads passing them the invariant and the remaining half permission to their respective ghost ref. Once the worker threads finish, the main thread gathers the permissions of the invariant and the ghost refs, and disposes them. With og back in the context, it is able to prove the required assertion about the counter.

```
let incr_main (v:erased int) (ctr:ref int) : Steel unit (ctr ↦ v) (λ _ → ctr ↦ (v + 2)) =
  let r1 = ghost_alloc 0 in let r2 = ghost_alloc v in (∗ allocate the ghost refs:: r1 --> 0 ∗ r2 --> v ∗ ctr ↦ v ∗)
  ghost_share r1; ghost_share r2; (∗ split permissions ∗)
  intro_∃ (hide 0, v) (og' ctr r1 r2); (∗ r1 -0.5-> 0 ∗ r2 -0.5-> v ∗ og ctr r1 r2 ∗)
  let i = new_inv (og ctr r1 r2) in (∗ allocate the disposable invariant sealing og:: ... ∗ active 1.0 i ∗)
  share i; (∗ split the invariant permission:: active 0.5 i ∗ r1 -0.5-> 0 ∗ active 0.5 i ∗ r2 -0.5-> v ∗)
  let _ = par (incr_with_invariant ctr r1 r2 0 true i) (incr_with_invariant ctr r2 r1 v false i) in (∗ workers ∗)
  gather_invariant i; dispose i; (∗ dispose the invariant:: r1 -0.5-> 1 ∗ r2 -0.5-> v + 1 ∗ og ctr r1 r2 ∗)
  let w = open_∃ () in (∗ r1 -0.5-> 1 ∗ r2 -0.5-> v + 1 ∗ og' ctr r1 r2 w ∗)
  ghost_gather (incr 0) r1; ghost_gather (incr v) r2; (∗ r1 --> 1 ∗ r2 --> v + 1 ∗ ctr ↦ v + 2 ∗)
  drop () (∗ drop the ghost refs:: ctr ↦ v + 2 ∗)
```

Imperative lemmas as ghost code. We show some of the relevant triples in comments to highlight how the use of ghost code manipulates ghost state as well as the logical context. Note, for instance, the use of intro_∃ and open_∃ to manipulate quantifiers. Getting "your hands on" an erased witness is possible due to frame-passing semantics of SteelCore's definitional interpreter together with a classical axiom from F⋆'s library.[4] The drop function call at the end drops the frame, which in our case consists of the ghost_pts_to predicates for the two ghost refs. Since our separation logic is

[4] SteelCore's logic is classical anyway, for several reasons, not least the use of SMT. So, this isn't particuarly onerous.

affine, we can implement such a combinator. It is possible to restrict it so that drop is allowed only for certain predicates, e.g., those that describe ghost state only.

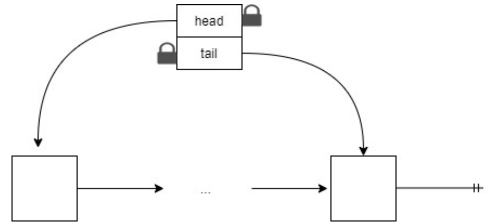
Finally, the worker threads open the invariant, thereby getting full permission to the counter and to their respective ghost ref. They increment the counter and the ghost ref, repackage the invariant, and return. Given an atomic operation to increment a ref (or a CAS), we implement the worker threads with the following signature.

```
let incr_with_invariant (ctr:ref int) (mine other:ghost_ref int) (n:erased int) (b:bool) (i:inv _)
  : Steel unit (active 0.5 i ∗ mine -0.5-> n) (λ _ → active 0.5 i ∗ mine -0.5-> (incr n))
  = with_invariant i (incr ctr mine other n b (name i))
```

5.5 Michael-Scott 2-Lock Queues

In this section, we present a proof of safety of a more realistic concurrent data structure, a queue by Michael and Scott (1996) which enables enqueuers and dequeuers to proceed in parallel. We prove the main invariants of the algorithm, including that the queue is always connected and that the head and tail point to first and last elements of the queue.

The main idea of the data struture is illustrated by the diagram alongside. A queue is implemented as a linked list that *always contains at least one element* (the last element cannot be dequeued) and a pair of pointers to the head and tail of the list. These head and tail pointers are each protected by a lock. Enqueuers take the tail lock, add a node at the end of the list, update the tail pointer, and release the lock. Dequeuers take the head lock, try to dequeue from the head of the list, and if successful, swing the head pointer to the next node, and release the lock. The interesting case is when the queue has only one element in it. In this case, the head and tail pointer point to the same node. Enqueing and dequeueing threads race on the next pointer of this node, with the enqueuer trying to update it while the dequeuer tries to read it. However, so long as reading or writing the next pointer is atomic, the algorithm correctly maintains the queue invariants.



To prove this in Steel, we follow a style similar in spirit to the Owicki-Gries parallel increment from the previous section, though this time we relate the invariants of the two locks with an atomic invariant on the queue itself by using two pieces of ghost state. Here's the invariant:

```
let lock_inv ptr ghost = ∃v. ptr ↦ v ∗ ghost -0.5-> v
let queue_invariant hd tl = ∃h t. hd.ghost -0.5-> h ∗ tl.ghost -0.5-> t ∗ Q.queue h t
type q_ptr a = { ptr : ref (Q.t a); ghost: ghost_ref (Q.t a); lock: lock (lock_inv ptr ghost) }
type t a = { head : q_ptr a; tail : q_ptr a; inv : inv (queue_invariant head tail) }
```

The type t a represents the structure of two fields at the top of the picture. The head and tail pointers are q_ptrs, holding the concrete pointer ptr to a queue node Q.t a, a ghost pointer, and a lock relating the two. The queue itself bundles the head and tail q_ptrs with an invariant token inv. The lock_inv holds full permission to the concrete pointer but only half the ghost pointer, while synchronizing them to hold the same value v. Meanwhile, the queue_invariant holds the other half of the ghost pointers together with the invariant Q.queue h t, which states that we have a valid non-empty linked list from h to t. These types and invariants drive the code that follows.

Creating a queue. To allocate a new queue, we allocate the underlying linked list with Q.new_queue and an initial element x. Then, we allocate the two queue pointers for head and tail, introduce the queue invariant, package it and return. The main proof effort involved is in allocating and sharing the ghost state, and introducing the existential quantifers. This is mostly done inside the auxiliary new_qptr function, which allocates a concrete and a ghost pointer to a queue node q:Q.t a and relates them by creating a lock: lock (lock_inv ptr ghost), finally returning a queue pointer alongside the remaining half share of the ghost pointer.

```
let new_queue (x:a) : Steel (t a) emp (λ _ → emp)
  = let new_qptr (q:Q.t a) : Steel (q_ptr a) emp (λ qp → qp.ghost -0.5-> q) =
        let ptr = alloc q in
        let ghost = ghost_alloc q in
        ghost_share ghost;
        intro_∃ _ (λ q → ptr ↦ q ∗ ghost -0.5-> q); (* need to introduce ∃, explicitly *)
        let lock = Steel.SpinLock.new_lock _ in
        { ptr; ghost; lock}
    in
    let hd = Q.new_queue x in
    let head = new_qptr hd in
    let tail = new_qptr hd in
    pack_queue_invariant _ _ head tail; (* need to package the invariant, 2 intro_∃ *)
    let inv = new_invariant _ _ in
    { head; tail; inv }
```

Enqueueing. The enqueue procedure below is also fairly clean. We start by acquiring a lock on the tail pointer. Then, we call a ghost computation from the library, open_∃ to return a witness for the existentially quantified lock_inv as an erased value. We then read the tail pointer and allocate a new cell with its next pointer properly initialized to null and ready for enqueuing.

```
let enqueue (hdl:t a) (x:a) : Steel unit emp (λ _ → emp)
  = Steel.SpinLock.acquire hdl.tail.lock;
    let v = open_∃ () in
    let tl = read hdl.tail.ptr in
    let cell = Q.({ data = x; next = null}) in
    let node = alloc cell in
    let enqueue_core #inames () : SteelAtomic unit inames
          (queue_invariant hdl.head hdl.tail ∗ (hdl.tail.ghost -0.5-> tl ∗ node ↦ cell))
          (λ _ → queue_invariant hdl.head hdl.tail ∗ hdl.tail.ghost -0.5-> node)
      = let h = open_∃ () in
        let t = open_∃ () in
        ghost_gather tl hdl.tail.ghost; (* fuse the two half permissions and get t=tl *)
        Q.enqueue tl node;
        ghost_write hdl.tail.ghost node; (* update the ghost state *)
        ghost_share hdl.tail.ghost;
        pack_queue_invariant _ _ _ _
    in
    with_invariant hdl.inv enqueue_core;
    write hdl.tail.ptr node;
```

```
intro_∃ _ (λ n → hdl.tail.ptr ↦ n ∗ hdl.tail.ghost -0.5-> n);
Steel.SpinLock.release hdl.tail.lock
```

The main work is done by the atomic computation enqueue_core which opens and restores the queue invariant, by calling Q.enqueue, which itself is an atomic update of tl→ next := node, but the proof involves exploiting the synchronization of the ghost and concrete state, updating it and restoring the invariant. Once we exit the atomic block, we update the tail pointer, introduce the lock invariant's existential, and release the lock. Dequeue is similar, we leave it out of the paper for lack of space.

Overall, with some carefully chosen types and invariants, the code mostly just writes itself, echoing Brady's type-define-refine slogan but with SteelCore's CSL specifications. The proof overhead compares favorably with other automated $F^\star$ developments—the framing is entirely automated, quantifier instantiation requires some manual intervention but the style we have here is very predictable, rather than relying on E-matching triggers for SMT. Yet, the interplay between SMT and tactics is profitable, with many small proofs done automatically behind the scenes. The whole procedure verifies in around 2 seconds including solving 25 SMT goals due to equality abduction in around 300 milliseconds.

5.6 PCMs for 2-party Session Types

As a final example, we illustrate how Steel can be used to build dependently typed libraries that provide both a foundational semantics as well as usable abstractions for embedded session-typed programming. SteelCore provided some steps in this direction, but only provided a library for unidirectional channels, which are of limited utility in comparison to the duplex channels we build here. Actris (Hinrichsen et al. 2019) is a full system that provides duplex channels with more features than we do here, but being a library in Iris, they stop short of providing dependently typed libraries for programming. On the other hand, while Actris provides implementations of the channel API using low-level operational primitives, our implementation is just a model of duplex channels showing that they can be realized by designing an appropriate PCM for 2-party dependent session types. In summary, what is proved by our PCM model is that the channel is represented by a trace of messages, where the trace is a word in the language accepted by the protocol state machine; and that the participants' knowledge of the channel are mutually compatible and represents agreement on a prefix of the trace. As much as demonstrating the applicability of Steel to message-passing programs, this example is meant to illustrate Steel's hybrid tactic/SMT-based automation at work with other features of SteelCore's logic, including user-defined PCMs.

To set the goal posts, we offer the following interface for duplex channels.

```
val ch : Type
val ep (name:party) (c:ch) (p:prot) : slprop
val new (p:prot) : Steel (ch & ch) emp (λ (cA, cB) → ep A cA p ∗ ep B cB p)
let msg_t (p:prot) : Type = match hnf p with | Msg _ a _ → a | Return #a _ → a
val send (c:ch { is_send_next n next }) (x:msg_t next) : Steel unit (ep n c next) (λ _ → ep n c (step next x))
val recv (c:ch { is_recv_next n next }) : Steel (msg_t next) (ep n c next) (λ x → ep n c (step next x))
val close (c:ch) : Steel unit (ep n c done) (λ _ → emp)
```

A channel is associated with a protocol p:prot via ep n c p, an slprop governing the use of one of the channel's named endpoints. A protocol is a free monad over basic actions to send and receive messages:

```
type tag = | Send | Recv
type prot : Type → Type =
```

```
| Return : #a:Type → v:a → prot a
| Msg : tag → a:Type → #b:Type → k:(a → prot b) → prot b
```

For example, a simple two-message protocol could be

```
reply_larger = Msg Send int (λ x → Msg Recv (y:int{y>x}) (λ _ → Return ()))
```

The type msg_t p computes the type of next message of the protocol p, by reducing a protocol to its head-normal form and returning the type of the next Msg or Return node.

To allocate a channel, one calls new and obtains two separate endpoints A and B—B interprets the protocol dually to A, flipping sends and receives.

If an endpoint's protocol p is Msg Send t k, then send c x can be called with x:t, and the endpoint transitions to the next state of the protocol, step p x = k x. Dually, recv c blocks until it can return a x:t when the protocol is currently Msg Recv t k and the protocol continues as k x. For instance, the following code typechecks, since the protocol type guarantees that B must reply with a value larger than what it received from A.

```
let pingpong (c:ch) : Steel unit (ep A c reply_larger) (ep A c done) = send c 17; let y = recv c in assert (y > 17)
```

This interface to channels is simple, intuitive and also quite powerful—protocols are monadic terms over the basic actions, and so support arbitary dependence on the *values* exchanged, including branching for internal and external choice, and recursion. The question that remains is how to implement this interface.

Background on PCMs. Our main insight is that one can design a PCM for protocols to orchestrate the temporal sharing of resources. As explained in §5.1, Steel's memory model includes PCM references, pref a p, a reference to a value of type a, where p:pcm a. The slprop associated with a pref is (r:pref a p) ⇝(v:a), where r ⇝v ∗ r ⇝u is equivalent to r ⇝v ⊕ u, where ⊕ is the composition operator of the PCM p. Further, r ⇝v is validated by a memory m only when there exists a frame f that is composable with v and m(r) = f ⊕ v. In other words, PCMs offer a form of rely-guarantee reasoning: a thread can rely on r ⇝v being stable, but must in turn guarantee that its actions on r preserve other threads' assertions on r.

A PCM for temporal sharing of protocol endpoints. At the core of our model of 2-party sessions is a PCM on t p, a type that captures each participant's knowledge of the partial traces of a protocol p. A channel allocated with protocol p is a pref (t p) q, where q: pcm (t p) is to be defined shortly.

```
type t (p:prot) = | Nil (* unit of the PCM: no knowledge *) | V : partial_trace_of p → t p (* full knowledge *)
  | A_W, B_R : q:prot {is_send q} → trace p q → t p (* A_W: A to send next, B_R: B to recv next *)
  | A_R, B_W : q:prot {is_recv q} → trace p q → t p (* A_R: A to recv next, B_W: B to send next *)
  | A_Fin, B_Fin : q:prot{is_ret q} → trace p q → t p (* A is finished, B is finished *)
```

Each case in t p is intended to represent some knowledge of the state of the channel from the perspective of some participant. For example, given a channel reference c : pref (t p) q, the assertion c ⇝A_W p tr is intended to model the knowledge that c is currently in a state where the protocol trace so far is tr and the next action on the trace is for A to send a message. To complete the construction, we need to define which elements of t p are composable and how to compose them.

```
let composable #p : symrel (t p) = λt0 t1 →
  match t0, t1 with | _, Nil | Nil, _ → ⊤(* unit composes with everything *) | V, _ | V, _ → ⊥(* V with nothing *)
  | A_Fin q s, B_Fin q' s' | B_Fin q s, A_Fin q' s' → q == q' ∧ s == s' (* both sides finished, traces agree *)
  | A_Fin q s, B_R q' s' | B_R q' s', A_Fin q s → ahead A q q' s s' (* A is finished, B still has to read *)
  | A_W q s, B_R q' s' | B_R q' s', A_W q s → ahead A q q' s s' (* A is writing, B is reading: A is ahead *)
  | A_R q s, B_R q' s' | B_R q' s', A_R q s → ahead A q q' s s' ∨ ahead B q' q s' s (* Both reading, either ahead *)
```

```
  | A_R q' s', B_Fin q s | B_Fin q s, A_R q' s' → ahead B q q' s s' (* B is finished, A still has to read *)
  | B_W q s, A_R q' s' | A_R q' s', B_W q s → ahead B q q' s s' (* B is writing, A is reading: B is ahead *)
```

The composable (symmetric) relation captures that the reader's knowledge of the state of the channel is a prefix of the messages that have been written so far; while the writer's knowledge is the entire partial trace so far. When the protocol is finished, both participants agree on the entire trace.

The final step in defining our PCM is to compose the participants knowledge of the traces—since the traces are composable, one of the traces is always a prefix of the other, and so composition takes the longer of the two traces.

```
let compose (s0:t p) (s1:t p{composable s0 s1}) = match s0, s1 with
  | a, Nil | Nil, a → a
(* Just build V with the longer of the two traces *)
  | A_Fin q s, _ | _, A_Fin q s | B_Fin q s, _ | _, B_Fin q s → V (mk_trace q s)
  | A_W q s, B_R q' s' | B_R q' s', A_W q s | B_W q s, A_R q' s' | A_R q' s', B_W q s → V (mk_trace q s)
  | A_R q s, B_R q' s' | B_R q' s', A_R q s → if len s ≥ len s' then V (mk_trace q s) else V (mk_trace q' s')
```

Taking these definitions as the basis of q:pcm (t p), the essence of our 2-party session typed channels is done. With the knowledge that, say, c ⇝A_W p tr, an endpoint can only advance the channel to an extension of the trace tr. Conversely, with the knowledge that, say, c ⇝A_R p tr, an endpoint can rely on the fact that either the current value of the channel is already or will be extended to be ahead of the protocol state p, and the value expected by the receiver can be read from the trace.

Representing a channel. This PCM-based rely-guarantee reasoning enables a fairly straightforward implementation of the main API we presented at the start, though with several levels of abstraction. We start with our representation of channels, the type ch below.

```
let chan p = ref (t p) (pcm p)
type ch = { p:prot; chan:chan p; tr: ref (until:prot & trace p until) }
```

A channel ch is a triple of a protocol index p; a reference chan holding the current state of the channel; and a reference tr containing a partial trace of the messages exchanged on the channel so far, i.e., from the start state of p until the current state of the protocol, until. This trace will allow us to state our main invariant and will also serve, operationally, as the queue of messages transmitted so far and to be dispatched to the other endpoint.

The key representation predicate, ep name c next (defined below) states that there exists a partial trace (| until, tr |), such that (1) the protocol's next state is until; (2) c.chan carries knowledge (from the perspective of the given endpoint) that the protocol's current state is an extension of the trace tr; and, (3) relating the two, the trace reference c.tr points to (|until, tr|).

```
let k_of name (next:prot) (tr:trace p next) = match name with
| A → if is_send next then A_W next tr else if is_recv next then A_R next tr else A_Fin next tr
| B → if is_send next then B_R next tr else if is_recv next then B_W next tr else B_Fin next tr
let ep name c next = ∃(| until, tr |). until == next ∗ c.chan ↦ k_of name next tr ∗ c.tr ↦ (|until, tr|)
```

Writing to and reading from a channel. Finally, the core of the top-level API to send and recv messages is implemented by the two functions shown below, write_a and read_a—similar functions exist for B. The top-level send and recv simply multiplex between these functions (and update the trace references) to present a single API for both participants.

To write a message on the channel, write_a expects the channel to be in the A_W next tr state. The main work here is calling a generic action for frame-preserving updates, upd_gen_action. Operationally, upd_gen_action c x y f reads the memory cell corresponding to c obtaining the current complete value for the channel v, which is provably equal to compose frame x, for some frame. The total function f updates v to v' = compose frame y, i.e., updating the local knowledge of the channel from x to y, while preserving frames.[5] In this case, write_a updates A's knowledge of the channel r from A_W next tr to v. The proof of write_a_f (the total, frame-preserving update function) is non-trivial and takes around 100 lines, but only involves pure reasoning about the PCM, and does not use any Steel-specific features.

```
let write_a (r:chan p { is_send next }) (tr:trace p next) (x:msg_t next)
  : Steel unit (r ↦ A_W next tr) (λ _ → r ↦ k_of A (step next x) (extend tr x))
  = let v = k_of A (step next x) (extend tr x) in
    upd_gen_action r (A_W next tr) v (write_a_f tr x)
```

Reading a message from a channel is the dual operation shown below. In this case, we expect the channel to be in the A_R next tr state, indicating that it's A's turn to read. If when reading the channel we find that it has no new message to deliver yet (the current trace is not strictly longer than our static knowledge of the protocol state), we just loop and retry. Otherwise, by calling another lemma about the PCM (compatible_a_r_v_is_ahead) we learn that the next message in the queue has a type compatible with the next state of the protocol (i.e., msg_t next) and can refine A's knowledge of the current state of the channel by advancing the state machine.

```
let rec read_a (c:chan p { is_recv next }) (tr:trace p next)
  : Steel (msg_t next) (c ↦ A_R next tr) (λ x → c ↦ k_of A (step next x) (extend tr x))
  = let tr' = get_a_r c next tr in
    if trace_length tr ≥ trace_length tr'.tr then read_a c next tr
    else (compatible_a_r_v_is_ahead tr tr'; let x = next_message tr tr'.tr in return x)
```

It is worth repeating that while our library provides a semantic basis for dependent session types, and although it is executable, it is not a particularly realistic implementation of a channel in shared memory, e.g., it maintains the entire trace of interactions and operations on the channel have to protected by a lock. In the future, we plan to integrate our channel API with the concurrent queue from §5.5, to only hold the messages that are yet to be delivered and to erase the traces by holding them in ghost references. Since with the session-typed API, write privilege on the channel is only ever held by one endpoint at time, we speculate that for this particular usage, we may actually be able to eliminate the use of locks in our two-lock queue. More pragmatically, to focus on programming and proving distributed systems, we plan to use our session-typed channel API, proven here to be semantically justified in Steel, and to link it to a native implementation of sockets provided by the underlying execution platform.

6 RELATED WORK

We have discussed several strands of related work throughout the paper, especially in §2, and provide a few more perspectives here. Steel's impredicative CSL of PCMs and dynamically allocated invariants is a descendent of Iris (Jung et al. 2018). Whereas Iris provides a logical framework in which users can embed and reason about programming languages, Steel's approach is the converse:

[5] upd_gen_action is expected to execute atomically. To execute it safely at runtime, one might use a lock to protect all accesses to memory cells that support this form of generic frame-preserving update.

we embed a program logic into a full-fledged dependently typed programming language, providing an effectful foundation for CSL for partial correctness, as opposed to Iris' step-indexed models. Our goals are also perhaps complementary: Iris' expressive power and interactive proof environment make it very suitable for formalizing programming language semantics and tricky concurrent algorithms, while for Steel we aim to provide pragmatic automation for simpler code through dependently typed proof-oriented libraries. As a case in point, consider Vindum and Birkedal's (2021) impressive recent proof of contextual refinement of the Michael-Scott lock-free queue based on Iris: contextual refinement proofs are beyond what is possible in Steel, though we offer instead a simple proof of Michael-Scott's two-lock queue with a high level of automation.

Steel's shallow embedding of a CSL in a dependently typed language makes it also closely related to HTT (Nanevski et al. 2008) and its more recent variants like FCSL (Nanevski et al. 2019). The main differences lie, first, in the underlying logics—unlike FCSL, Steel is impredicative and supports dynamically allocated invariants—and in the proof automation provided—FCSL proofs are interactive tactics in Coq, whereas Steel integrates a variety of automation techniques as part of a DSL package. While others provide frameworks to verify sequential (Charguéraud 2011; Chlipala 2011) and concurrent imperative programs (Appel 2011; Sammler et al. 2021) in Coq, their work applies to languages embedded in Coq rather than Coq itself. As far as we are aware, Steel and HTT/FCSL, are the only systems that offer to develop dependently typed programs in CSL.

Many tools and solvers aim to automate various fragments of separation logic, including tools like Smallfoot (Berdine et al. 2005) or Cyclist (Brotherston et al. 2012) and frameworks for heap shape analysis (Yang et al. 2008). Much of the work on automation has focused on *full automation*, including handling recursive predicates and quantifiers while aiming to scale lightweight analyses and bug finding tools to large codebases. In contrast, we focus on simple automation for user-assisted proofs of functional correctness. In that sense, our work is closer to frameworks like Viper (Müller et al. 2016) or VeriFast (Jacobs et al. 2011). Some of our specifications closely mimic Viper's implicit dynamic frames style, with an access permission and a heap-fragment refinement. For instance, in our verified implementation of balanced trees in §5.2, the validity of the tree ptr separation logic predicate represents permission to access the mutable tree at reference ptr, while the functional correctness of the data structure is specified using selector predicates. But compared to Viper, Steel also offers different verification styles, for instance based on PCMs or invariants, which we use in several other examples. To verify C and Java programs, VeriFast automatically splits verification conditions between separation logic assertions and pure predicates during its symbolic execution. But while this automatic splitting is a nice feature, pure assertions in VeriFast do not depend on the heap, compared to Viper's heap-fragment refinements or to Steel's selector predicates. Furthermore, while Viper, VeriFast and Steel all rely on an SMT solver to discharge verification conditions, another distinction with Viper and VeriFast is that Steel is built on top of SteelCore's foundational semantics for CSL in F*. As such, the TCB of the Steel framework corresponds exactly to that of F* itself. In particular, Steel leverages F*'s generic support for SMT solving—there is no specialized, trusted SMT encoding for Steel verification conditions.

7 CONCLUSION

At last year's ICFP we defined SteelCore, a foundational CSL developed in F*. Although expressive, proofs in SteelCore were painfully manual. With Steel, we have developed a full-fledged language embedded in F* for concurrent programming with semi-automated proofs, without extending the trust assumptions of SteelCore and while preserving its expressive power. A key enabler has been our use of a mixture of tactics and SMT to solve AC-unification constraints modulo theories, a design point that has been understudied in the literature seeking to automate separation logic.

ACKNOWLEDGMENTS

Aymeric Fromherz's and Sydney Gibson's work was supported in part by grants from the Intel Corporation, the Alfred P. Sloan Foundation, and the Department of the Navy, Office of Naval Research under Grant No. N00014-18-1-2892. Aymeric Fromherz was also supported in part by an internship at Microsoft Research. Sydney Gibson was also funded by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE1745016. Denis Merigoux's work was partially supported by the European Research Council under the CIRCUS Consolidator Grant Agreement (683032). We thank Ralf Jung, the shepherd of this paper; the anonymous reviewers; Deepak Kapur for insightful discussions about AC-unification, and all the members of Project Everest for their feedback and many useful discussions.

REFERENCES

Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software (ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17.

Josh Berdine, Cristiano Calcagno, and Peter W. O'hearn. 2005. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*. Springer, 115–137.

Edwin Brady. 2016. *Type-driven Development With Idris*. Manning. http://www.worldcat.org/isbn/9781617293023

James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. 2012. A Generic Cyclic Theorem Prover. In *Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 350–367.

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

Adam Chlipala. 2011. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1993498.1993526

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 34 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110278

François Fages. 1984. Associative-Commutative Unification. In *7th International Conference on Automated Deduction*, R. E. Shostak (Ed.). Springer New York, New York, NY, 194–208.

Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2016. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. Inria Saclay Ile de France. https://hal.inria.fr/inria-00258384

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2019. Actris: Session-Type Based Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article Article 6 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371074

Aquinas Hobor and Jules Villard. 2013. The Ramifications of Sharing in Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2429069.2429131

Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. 2014. Deciding Entailments in Inductive Separation Logic with Tree Automata. In *Automated Technology for Verification and Analysis*, Franck Cassez and Jean-François Raskin (Eds.). Springer International Publishing, Cham, 201–218.

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 14–26. http://dl.acm.org/citation.cfm?id=360204

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*.

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

Deepak Kapur and Paliath Narendran. 1987. Matching, Unification and Complexity. *SIGSAM Bull.* 21, 4 (Nov. 1987), 6–9. https://doi.org/10.1145/36330.36332

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370. http://dl.acm.org/citation.cfm?id=1939141.1939161

K. Rustan M. Leino and Michał Moskal. 2010. Usable Auto-Active Verification.

K. R. M. Leino, P. Müller, and J. Smans. 2009. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V (Lecture Notes in Computer Science)*, A. Aldini, G. Barthe, and R. Gorrieri (Eds.), Vol. 5705. Springer-Verlag, 195–222.

Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *28th European Symposium on Programming (ESOP)*. Springer, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS)*, B. Jobstmann and K. R. M. Leino (Eds.), Vol. 9583. Springer-Verlag, 41–62.

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

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

Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (Oct. 1979), 245–257. https://doi.org/10.1145/357073.357079

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

Susan Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* (1976).

M. J. Parkinson and A. J. Summers. 2012. The Relationship Between Separation Logic and Implicit Dynamic Frames. *Logical Methods in Computer Science* 8, 3:01 (2012), 1–54.

Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. 2020. Layered Indexed Effects: Foundations and Applications of Effectful Dependently Typed Programming. https://www.fstar-lang.org/papers/layeredeffects/ In submission.

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

M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *to appear in Programming Languages Design and Implementation (PLDI)*. ACM.

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

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

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '13)*. 387–398. https://www.microsoft.com/en-us/research/publication/verifying-higher-order-programs-with-the-dijkstra-monad/

Simon Friis Vindum and Lars Birkedal. 2021. Contextual Refinement of the Michael-Scott Queue (Proof Pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 76–90. https://doi.org/10.1145/3437992.3439930

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, and Dino Distefano. 2008. Scalable shape analysis for systems code. In *In CAV*.