Recalling a Witness

Foundations and Applications of Monotonic State

DANEL AHMAN, Inria, France
CÉDRIC FOURNET, Microsoft Research, UK
CĂTĂLIN HRIŢCU, Inria, France
KENJI MAILLARD, Inria and ENS, France
ASEEM RASTOGI, Microsoft Research, India
NIKHIL SWAMY, Microsoft Research, USA

We provide a way to ease the verification of programs whose state evolves monotonically. The main idea is that a property *witnessed* in a prior state can be soundly *recalled* in the current state, provided (1) state evolves according to a given preorder, and (2) the property is preserved by this preorder. In many scenarios, such monotonic reasoning yields concise modular proofs, saving the need for explicit program invariants. We distill our approach into the *monotonic-state monad*, a general yet compact interface for Hoare-style reasoning about monotonic state in a dependently typed language. We prove the soundness of the monotonic-state monad and use it as a unified foundation for reasoning about monotonic state in the F* verification system. Based on this foundation, we build libraries for various mutable data structures like monotonic references and apply these libraries at scale to the verification of several distributed applications.

CCS Concepts: • **Theory of computation** → **Hoare logic**; **Program verification**; • **Software and its engineering** → **Formal software verification**; • **Security and privacy** → *Distributed systems security*;

Additional Key Words and Phrases: Program Verification, Hoare Logic, Modular Reasoning, Monotonic References, Monotonic-State Monad, Formal Foundations, Secure File Transfer, State Continuity

ACM Reference Format:
Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a Witness: Foundations and Applications of Monotonic State. *Proc. ACM Program. Lang.* 2, POPL, Article 65 (January 2018), 30 pages. https://doi.org/10.1145/3158153

1 INTRODUCTION

Functional programs are easier to reason about than stateful programs, inasmuch as properties proven on pure terms are preserved by evaluation. In contrast, properties of imperative programs generally depend on their evolving state, e.g., a counter initialized to zero may later contain a strictly positive number, requiring properties that depend on this counter to be revised.

To reign in the complexity of reasoning about ever-changing state, verification can be structured using *stateful invariants*, i.e., predicates capturing properties that hold in every program state. Defining invariants and proving their preservation are the bread and butter of verification; and many techniques and tools have been devised to aid these tasks. A prominent such example

Authors' addresses: Danel Ahman, Inria, Paris, France; Cédric Fournet, Microsoft Research, UK; Cătălin Hriţcu, Inria, Paris, France; Kenji Maillard, Inria and ENS, Paris, France; Aseem Rastogi, Microsoft Research, India; Nikhil Swamy, Microsoft Research, USA.

2475-1421/2018/1-ART65
https://doi.org/10.1145/3158153

is separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2002], which offers a way to compose invariants according to the shape of mutable data.

Whereas separation logic is concerned primarily with *spatial* properties,[1] program verification also makes use of *temporal* properties that control how the state may evolve. For instance, consider a program with a counter that can only be increased. Reading n as its current value should allow one to conclude that its value will remain at least n, irrespective of state updates. In turn, this may be used to reason about the generation of unique identifiers from fresh counter values. One may hope to recover some of the ease of reasoning about pure programs in this setting, at least for properties that are preserved by counter increments. Capturing this intuition formally and using it to simplify the verification of stateful programs are the main goals of this paper.

1.1 Stateful Invariants vs. Monotonic State and Stable Predicates

Consider a program written against a library that operates on a mutable set. The library provides abstract operations to read the current value of the set and test whether an element is in the set ($\in$). Importantly, its only operation to mutate the state is 'insert', which adds an element into the set. From this signature alone, one should be able to conclude that an element observed in the set will remain in it. However, proving this fact is not always easy in existing program logics. To illustrate our point, consider trying to prove the assertion at the end of the following program:

```
insert v; complex_procedure(); assert (v ∈ get())
```

In a Floyd-Hoare logic, one may prove that our code maintains a stateful invariant on the set, i.e., prove the Hoare triple {inv} complex_procedure() {inv}, where inv s is defined as v ∈ s. One may rely on separation logic to conduct this proof, for instance by framing inv across complex_procedure provided it does not operate on the set. However, when complex_procedure also inserts elements, one must carry inv through its proof and reason about the effect of these insertions to confirm that they preserve inv. This quickly becomes tedious, e.g., showing that the set also retains some other element w requires another adjustment to the proof of complex_procedure. Besides, although this Hoare triple suffices to prove our assertion, it does not by itself ensure that v remains in the mutable set throughout execution. For example, v could have been temporarily removed and then reinserted. While detailed stateful invariants are often unavoidable, they are needlessly expensive here: knowing that elements can only be inserted, we would like to conclude that v ∈ s is stable.

Our Solution: Monotonic State and Stable Predicates. Let $\leq$ be a preorder (that is, a reflexive-transitive relation) on program states, or a fragment thereof. We say that a program is *monotonic* when its state evolves according to the preorder (that is, we have $s_0 \leq s_1$ for any two successive states), and that a predicate on its state is *stable* when it is preserved by the preorder (that is, when for all states s_0 and s_1, we have $\mathsf{p}\ s_0 \wedge s_0 \leq s_1 \implies \mathsf{p}\ s_1$). We outline an extension of Hoare-style program logics in this setting: we restrict any state-changing operations so that they conform with $\leq$, and we extend the logic and the underlying language with the following new constructs:

(1) A logical capability, witnessed, that turns a predicate p on states into a *state-independent* proposition witnessed p, expressing both that i) p was observed in some past state and ii) it will hold in every future state, including the program's current state.
(2) A weakening principle, (∀s. p s ⟹ q s) ⟹ (witnessed p ⟹ witnessed q).
(3) Two actions: witness p, to establish witnessed p, given that p is stable and holds in the current state; and recall p, to (re)establish p in the current state, given that witnessed p holds.

[1] Recent variants of separation logic consider resources more abstract than just heap locations. In such settings, in addition to describing spatial properties of resources, one can encode certain kinds of temporal properties. We compare our work to these modern variants of separation logic in §7, focusing here on more familiar program logics for reasoning about heaps.

Continuing with our example, one may pick set inclusion as the preorder and check that the only mutating operation, insert, respects it. Then, preserving (i.e., framing) stable properties across state updates is provided by the logic whenever explicitly requested using witness and recall. For instance, we can revise our example program as shown below to prove the assertion:

```
insert v; witness inv; complex_procedure(); recall inv; assert (v ∈ get())
```

Crucially, witness inv yields the postcondition witnessed inv, which, being a pure, *state-independent* proposition, is trivially maintained across complex_procedure without the need to analyze its definition. With recall inv, we recover inv of the current state without having to prove that it is related to the state in which inv was witnessed, since this follows from the stability of inv with respect to $\subseteq$. We could also prepend insert w to this code, and still would not need to revisit the proof of complex_procedure to establish that the final state contains w—we only need to insert the corresponding witness ($\lambda s \rightarrow w \in s$) and recall ($\lambda s \rightarrow w \in s$) operations into our program.

1.2 Technical Overview and Contributions

A point of departure for our work is Swamy et al.'s [2013b] proposal to reason about monotonic state using a variant of the witness and recall primitives. Following their work, the F* programming language [Swamy et al. 2016] has embraced monotonicity in the design of many of its verification libraries. Although these libraries have been founded on ad hoc axioms and informal meta-arguments, they are used extensively in several large-scale projects, including verified efficient cryptographic libraries [Protzenko et al. 2017; Zinzindohoué et al. 2017] and a partially-verified implementation of TLS [Bhargavan et al. 2017a,b]. The theory of monotonic state developed here provides a new, unifying foundation accounting for all prior, ad hoc uses of monotonicity in F*, while also serving as a general basis to the verification of monotonic properties of imperative programs in other systems. Our contributions include the following main points.

The Monotonic-State Monad (§2.2). We propose MST, the *monotonic-state monad*, and its encoding within a sequential, dependently typed language like F*. Its interface is simpler and more general than Swamy et al.'s [2013b] with just four actions for programming and reasoning about global, monotonic state (get, put, witness, and recall), together with a new logical connective (witnessed) for turning predicates on states into state-independent propositions.

Formal Foundations of the Monotonic-State Monad (§5, §6). We investigate the foundations of the monotonic-state monad by designing a sequent calculus for a first-order logic with the witnessed connective, proving it consistent via cut admissibility. We use this to prove the soundness of a Hoare-style logic for a core dependently typed lambda calculus augmented with the MST interface. We present this in two stages. First, we focus on an abstract variant of MST and prove our Hoare logic sound in the sense of total correctness (§5). Next, we show how to soundly reveal the representation of MST computations as pure state-passing functions, while carefully ensuring that the preorder enforced by MST is not violated. This pure representation can be used to conduct relational proofs (e.g., showing noninterference) for programs with monotonic state, and to enable clients to safely extend interfaces based on MST with new, preorder-respecting actions (§6).

Typed Heaps and Three Flavors of References (§2.3 and §2.4). Using MST, we encode more convenient forms of mutable state, including heaps with dynamic allocation and deallocation. Our model supports both untyped references uref with strong (non-type-preserving) updates, as well as typed references (ref t) with only weak (type-preserving) updates. Going further, we use MST to program monotonic typed references (mref t rel) that allow us to select a preorder for each reference separately, with ref t just a special case of monotonic references with a trivial preorder.

As such, programmers can opt-in to monotonicity whenever they allocate a reference, and retain the generality of non-monotonic state and stateful invariants wherever needed.

Secure File-Transfer, a First Complete Example Application (§3). Based on monotonic references, we verify a secure file-transfer application, illustrating several practical uses of monotonicity: ensuring safe memory initialization; modeling ghost distributed state as an append-only log of messages; and the interplay between stateful invariants, refinement types, and stable predicates. This application provides a standalone illustration of the essential use of monotonicity in a larger scale F⋆ verification project targetting the main streaming authenticated encryption construction used in TLS 1.3 [Bhargavan et al. 2017b].

Ariadne, Another Application to State Continuity (§4). We also develop a new case study based on a recent protocol [Strackx and Piessens 2016] that ensures the state continuity of hardware-protected sub-systems (such as TPMs and SGX enclaves) in the presence of multiple crashes and restarts. Our proof consists of programming a "ghost state machine" to keep track of the progress of the protocol. It illustrates the combination of ghost state and monotonicity as an effective style for verifying distributed algorithms, following the intuition that in a well-designed protocol, the logical properties carried by every message must be stable. While we do not address concurrency in this paper, this case study also illustrates how monotonic state may be usefully composed with other computational effects, such as failures and exceptions.

In §7 we discuss related work and in §8 we outline future work and conclude. Additional materials associated with this paper are available from https://fstar-lang.org/papers/monotonicity. These include full definitions and proofs for the formal results in §5 and §6, as well as the code for all examples in the paper. The table alongside shows the number of lines of F⋆ code for these examples. Compared to their code, the listings in the paper are edited for clarity and sometimes omit uninteresting details.

Example	LOC
Memory models	2152
Array library	544
File transfer	630
Ariadne	232

2 THE MONOTONIC-STATE MONAD, BY EXAMPLE

Verifying stateful programs within a dependently typed programming language and proof assistant is attractive for the expressive power provided by dependent types, and for the foundational manner in which a program's semantics can be modeled. Concretely, we develop our approach in F⋆ but our ideas should transfer to other settings, e.g., Hoare Type Theory [Nanevski et al. 2008], which F⋆ resembles, and also to other Hoare-style program logics. Since dependent type theory provides little by way of stateful programming primitives, we will have to build support for stateful programming from scratch, starting from a language of pure functions. We focus primarily on *modeling* stateful programming faithfully, rather than efficiently *implementing* imperative state. For instance, we will represent mutable heaps as functions from natural numbers to values at some type, although an efficient implementation should use the mutable memory available primitively in hardware.

We start by defining a simple, canonical state monad, the basic method by which stateful programming is introduced in F⋆ (§2.1). We then present the monotonic-state monad, MST, in its simplest, yet most general form: a global state monad parameterized by a preorder which constrains state evolution (§2.2). We instantiate this generic version of MST, first, to model mutable heaps with allocation, deallocation, untyped and typed references (§2.3), and then generalize our heaps further to include per-reference preorders (§2.4). Throughout, we use small examples to illustrate the pervasive applicability of monotonic state in many verification scenarios.

2.1 F⋆: A Brief Review and a Basic State Monad

We start with a short primer on F⋆ and its syntax, showing how to extend it with a state effect based on a simple state monad. In particular, F⋆ is a programming language with full dependent types, refinement types, and a user-defined effect system. Its effect system includes an inference algorithm based on an adaptation of Dijkstra's weakest preconditions to higher-order programs. Programmers specify precise pre- and post-conditions for their programs using a notation similar to Hoare Type Theory and F⋆ checks that the inferred specification is subsumed by the programmer's annotations. This subsumption check is reduced to a logical validity problem that F⋆ discharges through various means, including a combination of SMT solving and user-provided lemmas.

Basic Syntax. F⋆ syntax is roughly modeled on OCaml (val, let, match etc.) although there are many differences to account for the additional typing features. Binding occurrences b of variables take the form x:t, declaring a variable x at type t; or #x:t indicating that the binding is for an implicit argument. The syntax $\lambda(b_1)\ ...\ (b_n) \rightarrow t$ introduces a lambda abstraction, whereas $b_1 \rightarrow ... \rightarrow b_n \rightarrow c$ is the shape of a curried function type—we emphasize the lack of enclosing parentheses on the b_i. Refinement types are written b{t}, e.g., x:int{x≥ 0} is the type of non-negative integers (ℕ). As usual, a bound variable is in scope to the right of its binding; we omit the type in a binding when it can be inferred; and for non-dependent function types, we omit the variable name. For example, the type of the pure append function on vectors is written #a:Type → #m:ℕ→ #n:ℕ→ 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 #.

Basic State Monad and Computation Types. Programmers can extend the core F⋆ language of pure, total functions to effectful programs, by providing monadic representations for the effects concerned. For example, the programmer can extend F⋆ for stateful programming by defining the standard state monad (with the type st a = state → a ∗ state) together with two actions, get and put. Given this monad, using a construction described by Ahman et al. [2017], F⋆ derives the corresponding *computation type* ST t (requires pre) (ensures post), describing computations that, when run in an initial state s_0 satisfying pre s_0, produce a result r:t and a final state s_1 satisfying post s_0 r s_1. On top of that, F⋆ also derives a get and a put action for ST, with the following types:

```
val get : unit → ST state (requires (λ _ → ⊤)) (ensures (λ s0 s s1 → s0 == s ∧ s == s1))
val put : s:state → ST unit (requires (λ _ → ⊤)) (ensures (λ _ _ s1 → s1 == s))
```

and, when taking state = ℕ, the double function below has a trivial precondition (which we often omit) and a postcondition stating that the final state is twice the initial state.

```
val double : unit → ST unit (requires (λ _ → ⊤)) (ensures (λ s0 _ s1 → s1 == 2 ∗ s0))
let double () = let x = get () in put (x + x)
```

2.2 MST: The Monotonic-State Monad

In a nutshell, we use an abstract variant of st parameterized by a preorder that restricts how the state may be updated—abstraction is key here, since it allows us to enforce this update condition. A preorder is simply a reflexive and transitive relation:

```
let preorder a = rel: (a → a → Type) {(∀ x . x `rel` x) ∧ (∀ x y z. x `rel` y ∧ y `rel` z ⟹ x `rel` z)}
```

where x `rel` y is F⋆ infix notation for rel x y. (Preorders are convenient, inasmuch as we usually do not wish to track the actual sequence of state updates.) Figure 1 gives the signature of the abstract monotonic-state monad MST, parameterized by an implicit relation rel of type preorder state. Analogously to ST, the MST computation type is also indexed by a result type a, by a precondition

```
effect MST (a:Type) (requires (pre:(state → Type))) (ensures (post:(state → a → state → Type)))
(* [get ()]: A standard action to retrieve the current state *)
val get : unit → MST state (requires (λ _ → ⊤)) (ensures (λ s0 x s1 → s0 == x ∧ s0 == s1))
(* [put s]: An action to evolve the state to `s`, when `s` is related to the current state by `rel` *)
val put : s:state → MST unit (requires (λ s0 → s0 `rel` s)) (ensures (λ s0 _ s1 → s1 == s))
(* [stable rel p]: `p` is stable if it is invariant w.r.t `rel` *)
let stable (#a:Type) (rel:preorder a) (p: (a → Type)) = ∀x y. p x ∧ x `rel` y ⟹ p y
(* [witnessed p]: `p` was true in some prior program state and will remain true *)
val witnessed : (state → Type) → Type
(* [witness p]: A logical action; if `p` is true now, and is stable w.r.t `rel`, then it remains true *)
val witness : p:(state → Type) → MST unit (requires (λ s0 → p s0 ∧ stable rel p))
                                          (ensures (λ s0 _ s1 → s0 == s1 ∧ witnessed p))
(* [recall p]: A logical action; if `p` was witnessed in a prior state, recall that it holds now *)
val recall : p:(state → Type) → MST unit (requires (λ _ → witnessed p))
                                         (ensures (λ s0 _ s1 → s0 == s1 ∧ p s1))
(* [witnessed_weaken p q]: `witnessed` is functorial *)
val witnessed_weaken : p:_ → q:_ → Lemma ((∀ s. p s ⟹ q s) ⟹ witnessed p ⟹ witnessed q)
```

Fig. 1. MST: The monotonic-state monad (for rel:preorder state)

on the initial state, and by a postcondition relating the initial state, the result, and the final state. The bind and return for MST are the same as for ST. The actions of MST are its main points of interest, although the get action (line 3) is still unsurprising—it simply returns the current state.

Enforcing Monotonicity by Restricting put. The put action requires that the new state be related to the old state by our preorder (line 5). This is the main condition to enable monotonic reasoning.

Making Use of Monotonicity with witness and recall. The two remaining actions in MST have no operational significance—they are erased after verification. Logically, they capture the intuition that any reachable state evolves according to rel. The first action, witness p (line 11), turns a *stable* predicate p valid in the current state (i.e., p s_0 holds) into a state-independent, logical capability witnessed p. Conversely, the second action, recall p, turns such a capability into a property that holds in the current state p s_1. In other words, a stable property, once witnessed to be valid, can be freely assumed to remain valid (via recall) irrespective of any intermediate state updates, since each of these updates (via the precondition of put) must respect the preorder.

A Tiny Example: Increasing Counters. Taking state = $\mathbb{N}$ and rel = $\leq$, we can use MST to model an increasing counter. The F* code below (adapted from §1) shows how to use monotonicity to prove the final assertion, regardless of the stateful complex_procedure.

```
let x = get () in witness (λ s → x ≤ s); complex_procedure (); recall (λ s → x ≤ s); assert (x ≤ get())
```

The key point here is that the proposition witnessed (λs → x ≤ s) obtained by the witness action is state independent and can thus be freely transported across large pieces of code, or even unknown ones like complex_procedure. In F* any state independent proposition can be freely transported using either the typing of bind or the rule for subtyping, which plays the same role as the rule of consequence in classical Hoare Logic. The following valid Hoare logic rule illustrates this intuition:

$$(\textsc{Transporting State-Independent } R)\ \frac{\{P\}\ c\ \{Q\}}{\{\lambda s.\ P\ s \wedge R\}\ c\ \{\lambda s.\ Q\ s \wedge R\}}$$

As such, MST provides a modular reasoning principle, which is key to scaling verification in the face of state updates—the success of approaches like separation logic [Reynolds 2002] and dynamic framing [Kassios 2006] speak to the importance of modular reasoning. Monotonicity provides another useful, modular principle, one that it is quite orthogonal to physical separation—in the example above, complex_procedure could very well mutate the state of its context, yet knowing that it only does so in a monotonic manner allows x ≤ get() to be preserved across it.

Discussion: Temporarily Escaping the Preorder. In some programs, a state update may temporarily break our intended monotonicity discipline. For example, consider a mutable 2D point whose coordinates can only be updated one at a time. If the given preorder were to require the point to follow some particular trajectory (e.g., x=y), it would prevent any update to x or y.

One way to accommodate such examples is to define a more sophisticated preorder to track states where the original preorder can be temporarily broken. For instance, if we want the state s to respect a base rel_s: preorder s, while temporarily tolerating violations of this preorder, we could instantiate MST with a richer state type, t = Ok: s → t | Tmp: s → s → t, where 'Tmp snapshot actual' represents a program state with 'actual' as its current value and 'snapshot' as its last value obtained by updates that followed rel_s. We lift rel_s to rel_t so that once in Tmp, the actual state can evolve regardless of the preorder rel_s (see all the Tmp cases below):

```
let rel_t t0 t1 = match t0, t1 with
  | Ok s0, Ok s1 | Ok s0, Tmp s1 _ | Tmp s0 _, Ok s1 | Tmp s0 _, Tmp s1 _ → s0 `rel_s` s1
```

To temporarily break and then restore the base preorder rel_s, we define the two actions below, checking that the current state is related to the last snapshot by rel_s before restoring monotonicity.

```
val break : unit → MST unit (requires (λ t0 → Ok? t0)) (ensures (λ t0 _ t1 → let Ok s = t0 in t1 == Tmp s s))
val restore : unit → MST unit (requires (λ t0 → match t0 with Ok _ → ⊥ | Tmp s0 s1 → s0 `rel_s` s1))
                              (ensures (λ t0 _ t1 → let Tmp _ s1 = t0 in t1 == Ok s1))
```

On the Importance of Abstraction. Suppose one were to treat an MST a (λ_ → ⊤) (λ_ _ _ → ⊤) computation as a pure state-passing function of type mst a = s_0:state → (x:a * s_1:state{s_0 `rel` s_1}). It might seem natural to work with this monadic representation of state, but it can quickly lead to unsoundness. To see why, notice that this representation allows the context to pick an initial state that is not necessarily the consequence of state updates adhering to the given preorder. For example, in the code snippet below, we first observe that the program state is strictly positive, and then define a closure f that relies on monotonicity to recall that its state, when called, is also positive. The precondition of f then requires the state-independent proposition witnessed (λs → s > 0) to be valid, which is provable because of the call to witness (λs → s > 0) before the let-binding. More importantly, however, this state-independent precondition of f does not put any restrictions on the actual state values with which one can call f (), a pure mst state-passing function. As a result, we can call f () with whichever state we please, e.g., 0, causing a division by zero error at runtime.

```
put(get() + 1); witness (λ s → s > 0); let f () = recall (λ s → s > 0); 1 / get() in f () 0 (* ⟵ BROKEN! *)
```

For the next several sections, we treat MST abstractly. We return to this issue in §6 and carefully introduce two coercions, reify and reflect, to turn MST computations into mst functions and back, while not compromising monotonicity. Using reify, we show how to prove relational properties of MST computations, e.g., noninterference; and using reflect, we show how to add new actions that respect the preorder (while potentially temporarily violating it in an unobservable way).

2.3 Heaps and References, Both Untyped and Typed

The global monotonic state provided by MST is a useful primitive, but for practical stateful programming we need richer forms of state. In this section we show how to instantiate the state and preorder of MST to model references to mutable, heap-allocated data. Our references come in two varieties. Untyped references (type uref) represent transient locations in the heap whose type may change as the program evolves, until they are explicitly deallocated. Typed references (type ref t) are always live (at least observably so, given garbage collection) and contain a t-typed value.

Prior works on abstractly encoding mutable heaps within dependent types, e.g., HTT [Nanevski et al. 2008] and CFML [Charguéraud 2011], only include untyped references; while primitive and general, they are also harder to use safely, since one must maintain explicit liveness invariants of the form $(r \mapsto_\tau \_)$ stating that an untyped reference contains a value of type τ in the current state. Swierstra's [2009] shape-indexed references are more sophisticated, but still require a proof to accompany each use of the reference to establish that the current heap contains it. Using our monotonic-state monad MST, we show how to directly account for both typed and untyped references, where typed references are just as easy to use as in more mainstream, ML-like languages, without the need for liveness invariants or explicit proof accompanying each use: a value r:ref t is a proof of its own well-typed, membership in the current heap.

We model heap (see code below) as a map from abstract *identifiers* to either Unused or Used cells, together with a counter (ctr) tracking the next Unused identifier. As in the CompCert memory model [Leroy and Blazy 2008], identifiers are internally represented by natural numbers (type $\mathbb{N}$) but these numbers are not connected to the real addresses used by an efficient, low-level implementation. New cell identifiers are freshly generated by bumping the counter and are never reused.

```
type tag = Typed : tag | Untyped : bool → tag
type cell = Unused | Used : a:Type → v:a → tag → cell
type heap = H : h:(ℕ → cell) → ctr:ℕ{∀ (n:ℕ{ctr ≤ n}). h n == Unused} → heap
```

A 'Used a v tag' cell is a triple, where a is a type and v is a value of type a. As such, heaps are heterogeneous maps (potentially) storing values of different types for each identifier. The tag is either Typed, marking a cell referred to by a typed reference; Untyped true, for a live, allocated untyped cell; or Untyped false, for a cell that was once live but has since been deallocated. (We distinguish Untyped false from Unused to simplify our model of freshness—a client of our library can treat a newly allocated reference as being distinct from all previously allocated ones.)

Using monotonicity, we now show how to define our two kinds of references. In particular, a *typed reference* ref t is an identifier id for which the heap has been witnessed to contain a Used, Typed cell of type t. An *untyped reference* uref, on the other hand, has a much weaker invariant: it was witnessed to contain a Used, Untyped cell that could have since been deallocated.

```
let has_a_t (id:ℕ) (t:Type) (H h _) = match h id with Used a _ Typed → a == t | _ → ⊥
abstract type ref t = id:ℕ{witnessed (has_a_t id t)}
let has (id:ℕ) (H h _) = match h id with Used _ _ (Untyped _) → ⊤ | _ → ⊥
abstract type uref = id:ℕ{witnessed (has id)}
```

To enforce these invariants on state-manipulating operations, we define a preorder rel on heap that constrains the heap evolution. It states that every Used identifier remains Used; every Typed reference has a stable type; and that no Untyped reference may be reused after deallocation.

```
let rel (H h0 _) (H h1 _) = ∀id. match h0 id, h1 id with
                                 | Used a _ Typed, Used b _ Typed → a == b
                                 | Used _ _ (Untyped live0), Used _ _ (Untyped live1) → not live0 ⟹ not live1
                                 | _ → ⊥
```

Instantiating MST with state=heap and the preorder above, we can implement the expected operations for allocating, reading, writing typed references. The alloc action below allocates a new, typed reference ref t by generating a fresh identifier id; extending the heap at id with a new typed cell; and witnessing the new state, ensuring that id will contain a t-typed cell for ever. Reading and writing a reference are similar: they both recall the reference exists in the heap at its expected type.

```
let alloc #t (v:t) : MST (ref t) (ensures (λ h0 id h1 → fresh id h0 h1 ∧ modifies {} h0 h1 ∧ h1.[id] == v)) =
    let H h id = get () in put (H (upd h id (Used t v Typed)) (id + 1)); witness (has_a_t id t); id
let (!) #t (r:ref t) : MST t (ensures (λ h0 v h1 → h0 == h1 ∧ has_ref r h1 ∧ h1.[r] == v)) =
    recall (has_ref r); let h = get () in h.[r]
let (:=) #t (r:ref t) (v:t) : MST unit (ensures (λ h0 _ h1 → modifies {r} h0 h1 ∧ has_ref r h1 ∧ h1.[r] == v)) =
    recall (has_ref r); let H h ctr = get () in put (H (upd h r (Used t v Typed), ctr))
```

These functions make use of a few straightforward auxiliary definitions for freshness of identifiers (fresh) and for the write-footprint of a computation (modifies). The one subtlety is in the definition of h.[r], a total function to select a reference r from a heap h. Unlike the stateful (!) operator, h.[r] has a precondition requiring that h actually contain r—even though the type of r indicates that it has been witnessed in some prior heap of the program, this does not suffice to recall that it is actually present in an arbitrary heap h. In other words, pure functions may not use recall. On the other hand, the stateful lookup (!) is free to recall the membership of r in the current heap in order to meet the precondition of h.[r].[2]

```
let has_ref #t (r:ref t) h = has_a_t r t h
let fresh #t (r:ref t) (H h0 _) (H h1 _) = h0 r == Unused ∧ has_ref r h1
let modifies (ids:set ℕ) (H h0 _) (H h1 _) = ∀id. id ∉ ids ∧ Used? (h0 id) ⟹ h0 id == h1 id
let `_.[_]` #t h (r:ref t{has_ref r h}) : t = let H h _ = h in match h r with Used _ v _ → v
```

The operations of untyped references are essentially simpler counterparts of alloc, (!) and (:=) with weaker types. The free operation is easily defined by replacing an Untyped cell with a version marking it as deallocated. The precondition of free prevents double-frees, and is necessary to show that the preorder is preserved as we mark the cell deallocated.

```
let live (r:uref) (H h _) = match h r with Used _ _ (Untyped live) → live | _ → false
let free (r:uref) : MST unit (requires (live r)) (ensures (λ h0 _ h1 → modifies {r} h0 h1)) =
    let H h ctr = get () in put (H (upd h r (Used unit () (Untyped false))) ctr)
```

2.4 Monotonic References

Typed references ref t use a fixed global preorder saying that the type of each ref-cell is invariant. However, we would like a more flexible form, allowing the programmer to associate a preorder of their choosing with each typed reference. In this section, we present a library for providing a type ‘mref a ra’, a typed reference to a value of type ‘a’ whose contents is constrained to evolve according to a preorder ‘ra’ on ‘a’. Using mrefs, one can for instance encode a form of typestate programming [Strom and Yemini 1986] by attaching to an mref a preorder that corresponds to the reachability relation of a state machine.

As above, our global, monotonic-state monad MST can be instantiated with a suitable heap type for the global state (defined below) and a preorder on the global state that is intuitively the pointwise composition of the preorders associated with each mref that the state contains. In this setting, the type ref t can be reconstructed as a derived form, i.e., ref t = mref t (λ_ _ → ⊤)

[2] In our revision to the libraries of F⋆, we use a more sophisticated representation of ref t that enables a variant of h.[r] without the has_ref h r precondition. This variant is convenient to use in specifications, since its well-typedness is easier to establish. We omit it for simplicity, but the curious reader may consult the FStar.Heap library for the full story.

An Interface for Monotonic References. When allocating a monotonic reference one picks both the initial value and the preorder constraining its evolution. An mref a ra can be dereferenced unconditionally, whereas assigning to an mref a ra requires maintaining the preorder, analogous to the precondition on put for the global state.

```
type mref : a:Type → ra:preorder a → Type
val (:=) : #a:Type → #ra:preorder a → r:mref a ra → v:a → MST unit
    (requires (λ h → h.[r] `ra` v))
    (ensures (λ h0 _ h1 → modifies {r} h0 h1 ∧ h1.[r] == v))
```

The local state analog of witness on the global state allows observing a predicate p on the global heap as long as the predicate is stable with respect to arbitrary heap updates that respect the preorder only on a given reference. Using recall to restore a previously witnessed property remains unchanged.

```
let stable #a #ra (r:mref a ra) (p:(heap → Type)) = ∀h0 h1. p h0 ∧ h0.[r] `ra` h1.[r] ⟹ p h1
val witness : #a:Type → #ra:preorder a → r:mref a ra → p:(heap → Type){stable r p} → MST unit
    (requires (λ h → p h))
    (ensures (λ h0 v h1 → h0==h1 ∧ witnessed p))
```

Implementing Monotonic References. To implement mref we choose the following, revised representation of heap and its global preorder. We enrich the tags from §2.3 to additionally record a preorder with every typed cell. Correspondingly, the global preorder on heaps is, as mentioned earlier, the pointwise composition of preorders on typed cells (the cell and heap types are unchanged).

```
type tag a = Typed : preorder a → tag a | Untyped : bool → tag a
type cell = Unused | Used : a:Type → a → tag a → cell
type heap = H : h:(ℕ → cell) → ctr:ℕ{∀ (n:ℕ{ctr ≤ n}). h n == Unused} → heap
let rel (H h0 _) (H h1 _) = ∀id. match h0 id, h1 id with
    | Used a0 v0 (Typed ra0), Used a1 v1 (Typed ra1) → a0 == a1 ∧ ra0 == ra1 ∧ v0 `ra0` v1
    | Used _ _ _ (Untyped live0), Used _ _ _ (Untyped live1) → not live0 ⟹ not live1
    | _ → ⊥
let mref a ra = id:ℕ{witnessed (λ (H h _) → match h id with Used b _ (Typed rb) → a == b ∧ ra == rb)}
```

Monotonic references are our main building block for defining more sophisticated abstractions. While we have shown them here in the context of a flat heap, our libraries provide monotonic references within *hyper-heaps* [Swamy et al. 2016] and *hyper-stacks* [Protzenko et al. 2017], more sophisticated, region-based memory models used in F⋆ to keep track of object-lifetimes and to encode a weak form of separation.

3 MONOTONIC STATE IN ACTION: A SECURE FILE-TRANSFER APPLICATION

Consider transferring a file f from a sender application S to a receiver application R. To ensure that the file is transferred securely, the applications rely on a protocol P, which is designed to ensure that R receives exactly the file that S sent (or detects a transmission error) while no one else learns anything about f. For instance, P could be based on TLS, and use cryptography and networking to achieve its goals, but its details are unimportant. Our example addresses the following concerns:

(1) *Low-level buffer manipulation*: For efficiency reasons, S and R prepare byte buffers shared with P. For instance, the receiver R allocates an uninitialized buffer and requests P to fill the buffer with the bytes it receives. Monotonicity ensures that once memory is initialized it remains so.
(2) *Modeling distributed state*: To state and prove the correctness of a distributed system, even one as simple as our 2-party file-transfer scenario, it is common to describe the state of the

system in terms of some global *ghost* (i.e., purely specificational) state. We use monotonicity to structure this ghost state as an append-only log of messages sent so far, ensuring that an observation of the state of the protocol remains consistent even while the state evolves.

(3) *Fragmentation and authenticity*: Protocol P dictates a maximum size of messages that can be sent at once. As such, the sender has to fragment the file f into several chunks and the receiver must reconstruct the file. We use monotonicity to show that R always reads a prefix of the stream that S has sent so far, i.e., R receives authentic file fragments from S in the correct order.

(4) *Secrecy*: Finally, we consider possible implementations of the protocol P itself and prove that under certain, standard cryptographic hypotheses, P leaks no information about the file f, other than some information about its length.

3.1 Safely Using Uninitialized and Frozen Memory

Reasoning about safety in the presence of uninitialized memory is a well-known problem, with many bespoke program analyses targeting it, e.g., Qi and Myers's [2009] masked types. The essence of the problem involves reasoning about monotonic state—memory is unreadable until it is initialized, and remains *readable* thereafter. A dual problem is deeming an object no longer *writable*. For instance, after validating its contents, one may want to freeze an object in a high-integrity state.

Using monotonic references, we designed and implemented a verified library for modeling the safe use of uninitialized arrays that may eventually be frozen, including support for a limited form of pointer arithmetic to refer to prefixes or suffixes of the array. We sketch a fragment of this library here, showing only the parts relevant to the treatment of uninitialized memory.

The main type provided by our library is an abstract type 'array a n' for a possibly uninitialized array indexed by its contents type a and length n. An array is implemented under the hood by using a monotonic reference containing a seq (option a) and constrained by the preorder remains_init.

```
abstract type array (a:Type) (n:ℕ) = mref (repr a n) remains_init
where repr a n = s:seq (option a){len s == n}
and   remains_init #a #n (s0:repr a n) (s1:repr a n) = ∀(i:ℕ{i < n}). Some? s0.(i) ⟹ Some? s1.(i)
```

Notice the interplay between refinements types and monotonic references. The refinement type s:seq (option a){len s == n} passed to mref constrains the stored sequence to be of the appropriate length in every state. Though concise and powerful, refinements of this form can only enforce invariants of each reachable program state *taken in isolation*. In order to constrain how the array contents can *evolve*, the preorder remains_init states that the sequence underlying an array can evolve from s_0 to s_1 only if every initialized index in s_0 remains initialized in s_1.

Given this representation of array, the rest of the code is mostly determined. For instance, its create function takes a length n but no initial value for the array contents.

```
abstract let create (a:Type) (n:ℕ) : ST (array a n) (ensures (λ h0 x h1 → fresh x h0 h1 ∧ modifies {} h0 h1)) =
  alloc (Seq.create n None) remains_init
```

The pure function as_seq h x enables reasoning about an array x in state h as a sequence of optional values. Below we use it to define which parts of an array are initialized, i.e., those indices at which the array contains Some value.

```
let index #a #n (x:array a n) = i:ℕ{i < n}
let initialized #a #n (x:array a n) (i:index x) (h:heap) = Some? (as_seq h x).(i)
```

The main use of monotonicity in our library is to observe that initialized is stable with respect to remains_init, the preorder associated with an array. As such, we can define a state-independent proposition (x `init_at` i) using the logical witnessed capabilities.

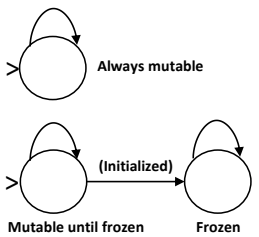
```
let init_at #a #n (x:array a n) (i:index x) = witnessed (initialized x i)
```

We can now prove that writing to an array at index i ensures that it becomes initialized at i, which is a necessary precondition to read from i. Notice the use of witness when writing, to record the fact that the index is initialized and will remain so; and the use of recall when reading, to recover that the array is initialized at i and so the underlying sequence contains Some v at i.

```
abstract let write (#a:Type) (#n:ℕ) (x:array a n) (i:index x) (v:a) : ST unit
 (ensures (λ h0 _ h1 → modifies {x} h0 h1 ∧ (as_seq h1 x).(i) == Some v ∧ x `init_at` i)) =
 x := Seq.upd !x i (Some v); witness (initialized x i)

abstract let read (#a:Type) (#n:ℕ) (x:array a n) (i:index x{x `init_at` i}) : ST a
 (ensures (λ h0 r h1 → modifies { } h0 h1 ∧ Some r == (as_seq h0 x).(i))) =
 recall (initialized x i); match !r.(i) with Some v → v
```

It is worth noting that in the absence of monotonicity, the init_at predicate defined above would need to be state-dependent, and thus carried through the subsequent stateful functions as a stateful invariant, causing unnecessary additional proof obligations and bloated specifications.



Freezing and Sub-Arrays. Our full array library supports freezing an array once it is fully initialized. We add a stateful predicate is_mutable x h indicating that the array x is mutable in state h. By design, is_mutable is not a stable predicate–freezing an array explicitly revokes mutability. While is_mutable is a precondition of write, freezing an initialized array provides a stable predicate frozen_with x s, where s:seq a represents the stable snapshot of the array contents–the clients can later recall that x's contents still correspond to s. The library also supports *always mutable* arrays in the same framework, for which is_mutable is indeed a stable predicate. Consequently, the write operation on always mutable arrays has no stateful precondition, since its argument's mutability can just be recalled. Internally, these temporal properties of an array are managed by a preorder capturing the state-transition system shown alongside. Aside from these core operations, our library provides functions to create aliases to a prefix or a suffix of an array, while propagating information about the fragments of the array that are initialized or frozen to the aliases.

3.2 Modeling the Protocol's Distributed State

Beyond simple safety, to prove our file-transfer application correct and secure, we first need to model the protocol P. Conceptually, we model each instance of the protocol using a *ghost* state shared between the protocol participants (in this case, just S and R). The shared state contains a log of message fragments already sent by S; the main invariant of the protocol is that successful receive operations return fragments from a prefix of the stream of fragments sent so far. This style of modeling distributed systems has a long tradition [Chandy and Lamport 1985], and several recent program logics and verification systems have incorporated special support for such shared ghost state [Bhargavan et al. 2010; Sergey et al. 2018; Swamy et al. 2013a]—we just make use of our monotonic references library for this.

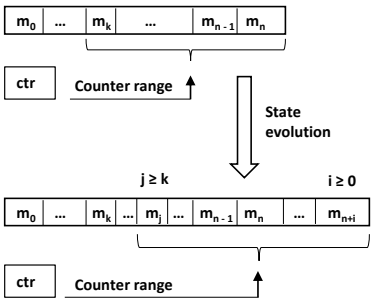
Connection State. The state of a protocol instance is the abstract type connection. Its interface (shown below) provides a function log c h, representing the messages sent so far on c; and a sequence counter ctr c h, representing the current position in the log. Whereas the sender and

receiver each maintain their own counters, the log is shared, specification-only state between the two participants.[3]

```
type connection
type message = s:seq byte{len s ≤ fragment_size}

val log: connection → heap → seq message
val ctr: connection → heap → ℕ
val is_receiver: connection → bool

let receiver = c:connection{is_receiver c}
let sender = c:connection{not (is_receiver c)}
```



Monotonic Properties of the Protocol. Monotonicity comes into play with the snap operation. A snapshot of the distributed protocol state remains stable as the state evolves. Obtaining such stable snapshots is a basic component of designing and verifying distributed protocols, and our libraries make this easy to express. In particular, the snapshotted log remains a prefix of the log and the counter never decreases. The clients can also recall that in any state, the counter value is bounded by the length of the log, i.e., the valid transitions of the counter are dependent on the log, as depicted in the figure above.

```
let snapshot c h0 h = log c h0 `is_a_prefix_of` log c h ∧ ctr c h0 ≤ ctr c h ∧ ctr c h0 ≤ len (log c h)
val snap: c:connection → MST unit (ensures (λ h0 _ h1 → h0 == h1 ∧ witnessed (snapshot c h0)))
val recall_counter: c:connection → MST unit (ensures (λ h0 _ h1 → h0 == h1 ∧ ctr c h0 ≤ len (log c h0)))
```

Receiving a Message. Receiving a message using receive buf c requires buf to contain enough space for a message, and that buf and c use disjoint state. It ensures that only the connection and the buffer are modified; that at most fragment_size bytes are received into buf; and the bytes correspond to the message recorded in the log at the current counter position, which is then incremented.

```
val receive: #n:ℕ{fragment_size ≤ n} → buf:array byte n → c:receiver{disjoint c buf} → MST (option ℕ)
  (ensures (λ h0 ropt h1 → match ropt with None → h0 == h1
                         | Some r → modifies {c, buf} h0 h1 ∧ r ≤ fragment_size ∧ modifies_array buf 0 r ∧
                                    all_init (prefix buf r) ∧ ctr c h1 == ctr c h0 + 1 ∧ log c h1 == log c h0 ∧
                                    sub_sequence (as_Seq buf h1) 0 r == (log c h0).(ctr c h0)))
```

Sending a Message. Sending a message using send buf c, requires buf to be an array of initialized bytes.[4] The postcondition ensures that only the connection state is modified, at most fragment_size bytes are sent, the ctr is incremented by one, and the log is extended with the sent message.

```
val send : #n:ℕ → buf:array byte n{all_init buf} → c:sender{disjoint c buf} → MST ℕ
  (ensures (λ h0 sent h1 → modifies {c} h0 h1 ∧ sent ≤ min n fragment_size ∧ ctr c h1 == ctr c h0 + 1 ∧
                           log c h1 == snoc (log c h0) (sub_sequence (as_seq buf h0) 0 sent)))
```

[3]The log is only needed for specification and modeling purposes. So, in practice, we maintain the log in computationally irrelevant state, which F⋆ supports. However, we elide this level of detail from our example here, since it is orthogonal.
[4]Of course, send and receive actually send or receive messages on the network, so they have more than just MST effect. However, for simplicity here, we model IO in terms of state, as detailed in §3.3.

Look Ma, No Stateful Invariants! To reiterate the importance of monotonicity, note that a salient feature of our protocol interface is its lack of stateful preconditions and invariants. This means, for instance, that we may create and interleave the use of several connections without needing to worry about interference among the instances. In contrast, were our interface to use stateful invariants (e.g., that the counter is always bounded by the length of the log, among other properties), one would need to prove that the invariant is maintained through, or framed across, all state updates. Of course, one should not expect to always eliminate stateful invariants; however, monotonic state, when applicable, can greatly simplify the overhead of maintaining stateful invariants.

3.3 Implementing the Protocol Securely

There are many possible ways to securely implement our protocol interface. We chose perhaps the simplest option, assuming the sender and the receiver share a private source of randomness, and use one-time pads for perfectly secure encryption with a message authentication code (MAC) authenticating the cipher and a sequence number. However, more complex and broadly deployable alternatives have been proven secure using monotonic state in F⋆, including the main "authenticated encryption with additional data" (AEAD) constructions used in TLS 1.3 [Bhargavan et al. 2017b].

Connection. A connection comes in two varieties: a pair (S rand entries) for the sender, or a triple (R rand entries ctr) for the receiver. Both sides share the same source of randomness (a stream of random fragments) and a log of entries; the receiver additionally has a monotonic counter to keep track of where in the stream it is currently reading from. Each entry in the log is itself a 4-tuple of: an index i into the randomness; the plain text m; the cipher text c, with an invariant claiming it to be computed from m via the one-time pad; and the message authentication code, mac. We supplement the log with an invariant that the ith entry in the log has index i—this will be needed for proving the receiver correct.

```
type network_message = s:seq byte{len s == fragment_size}
type randomness = ℕ→ network_message
type entry (rand:randomness) =
  | Entry : i:ℕ → m:message → c:network_message{pad m ⊕ rand i == c} → mac:tag i c → entry
type entries rand = s:seq (entry rand){∀ (i:ℕ{i < len s}). let Entry j _ _ _ = s.(i) in j == i}
type connection =
  | S : rand:randomness → entries:mref (entries rand) is_a_prefix_of → connection
  | R : rand:randomness → entries:mref (entries rand) is_a_prefix_of
      → ctr:mref (n:ℕ{witnessed (λ h → n ≤ len h.[entries])}) increasing → connection
```

Logs, Counters, and Snapshots. The types above show several forms of interplay between monotonicity, stable predicates, and refinement types. The preorders `is_a_prefix_of` and `increasing` restrict how the log and the counter evolve. Within an Entry, the refinement on the cipher enforces an invariant on the data stored in the log. Perhaps most interestingly, the type of the counter mixes a refinement and a stable predicate: it states that the monotonic reference holding the counter contains n:ℕ that is guaranteed to be bounded by the number of entries in the current state. This combination encodes a form of monotonic dependence among multiple mutable locations, allowing the preorder of the counter to evolve as the log evolves, i.e., a log update allows the counter to be advanced further along its increasing preorder.

With our type definitions in place, we can implement the signature from the previous subsection.

```
let entries_of (S _ es | R _ es _) = es
let log c h = Seq.map (λ (Entry _ msg _ _) → msg) h.[entries_of c] (* the plain texts sent so far *)
let ctr c h = match c with S _ es → len h.[es] | R _ _ ctr → h.[ctr] (* write at end; read from a prefix *)
```

```
let recall_counter c = match c with S _ _ → () | R _ es ctr → let n = !ctr in recall (λ h → n ≤ len h.[es])
let snap c = let h0 = ST.get () in recall_counter c; witness (snapshot c h0)
```

Encrypting and Sending a Message Fragment. To implement send buf c, we take a message of size at most 'min n fragment_size' from buf, pad it if needed, encrypt it using the current key from rand, and compute the MAC using the cipher and the current counter. We then add a new entry to the log, preparing to send the cipher and the MAC, and then call network_send c cipher mac, which expects the cipher and mac to be the same as in the last log entry.

Receiving, Authenticating and Decrypting a Message. The receiver's counter indicates the sequence number *i* of the next expected fragment. To receive it, we parse the bytes received from the network into a pair of a cipher and a MAC. We then authenticate that this is indeed the *i*th message sent using mac_verify, which, based on a cryptographic model of the MAC, guarantees that the log contains an appropriate entry for the cipher, sequence number, and MAC. We then decipher the message using the one-time pad, and the invariant on the entries (together with properties about pad, unpad, and ⊕) guarantees that the received message msg is indeed recorded in the log of entries. Before returning the number of bytes received, we must increment the counter. Manipulating the counter's combination of refinement and stable predicate requires a bit of care: we have to first witness that the new value of the counter will remain bounded by length of the log.

```
let receive #n buf c = let R rand entries ctr = c in let i = !ctr in
  match parse (network_receive c) with
  | None → None
  | Some (cipher, mac) →
    if mac_verify cipher mac i entries then (* guarantees that (Entry i _ cipher mac _) is in entries *)
      let msg = unpad (cipher ⊕ rand i) in Array.fill buf msg;
      witness (λ h → i + 1 ≤ len h.[entries]); ctr := i + 1; Some (len msg)
    else None
```

3.4 Correctness, Authenticity, and Secrecy of File Transfer

Using this protocol, we program and verify the top-level applications for sending and receiving entire files. Our goal is to show that the sender application *S* successfully uses the protocol to fragment and send a file and that the receiver reconstructs exactly that file, i.e., file transfer is correct and authentic. Additionally, we prove that file transfer is confidential, i.e., under a suitable cryptographic model, a network adversary gains no information about the transferred file.

We first specify what it means to send a file correctly in terms of the abstractions provided by the underlying protocol. We use 'sent_bytes f c from to' to indicate that the protocol's log contains exactly the contents of the file f starting at position from until to. The stable property sent f c then expresses that the file f was sent on connection c at some point in the past.

```
let sent_bytes (file:seq byte) (c:connection) (from:ℕ) (to:ℕ{from ≤ to}) (h:heap) =
  let log = log c h in to ≤ len log ∧ file == flatten (sub_sequence log from to)
let sent (file:seq byte) (c:connection) = ∃ from to. witnessed (sent_bytes file from to)
```

The sender's top-level function calls into the protocol repeatedly sending message-sized chunks of the file, until no more bytes remain. Its specification ensures the file was indeed sent.

```
let send_file (#n:ℕ) (file:array byte n{all_init file}) (c:sender{disjoint c file}) : MST unit
  (ensures (λ h0 _ h1 → modifies {c} h0 h1 ∧ sent (as_seq file h0) c))
  = let rec aux (from:ℕ) (pos:ℕ{pos ≤ n}) : MST unit
      (requires (λ h0 → from ≤ ctr c h0 ∧ sent_bytes (sub_seq (as_seq file h0) 0 pos) c from (ctr c h0) h0))
```

```
  (ensures (λ h0 _ h1 → modifies {c} h0 h1 ∧ from ≤ ctr c h1 ∧
                        sent_bytes (as_seq file h0) c from (ctr c h1) h1))
  = if pos ≠n then let sub_file = suffix file pos in let sent = send sub_file c in aux from (pos + sent) in
  let h0 = ST.get () in aux (ctr c h0) 0;
  let h1 = ST.get () in witness (sent_bytes (as_seq file h0) c (ctr c h0) (ctr c h1))
```

The receiver's top-level application is dual and similar to the sender. In the receiver's function 'receive_file file c', file starts off as a potentially uninitialized buffer, but the postcondition of receive_file guarantees that on a successful run, the file is partially filled with messages from a file that was previously sent on the same connection, i.e., file transfer is authentic.

```
let received (#n:ℕ) (file:array byte n) (c:receiver) (h:heap) = file `initialized_in` h ∧ sent (as_seq file h) c
val receive_file (#n:ℕ) (file:array byte n) (c:receiver{disjoint c file}) : MST (option ℕ)
  (ensures (λ h0 ropt h1 → modifies {file, c} h0 h1 ∧ (match ropt with None → ⊤
                                                     | Some r → r ≤ n ∧ received (prefix file r) c h1)))
```

Confidentiality. To prove that the file transfer is confidential, we relate the ciphers and tags sent on the network by two runs of the sender application and prove that for arbitrary files that contain the same number of messages, the network traffic is (probabilistically) indistinguishable. As such, our file-transfer application is only partially length hiding—the adversary learns lower and upper bounds on the size of the file based on the number of messages it contains. Our proof relies on a form of probabilistic coupling [Lindvall 2002], relating the randomness used in two runs of the sender by a bijection chosen to mask the differences between the two files. The proof technique used is independent of monotonicity and, as such, is orthogonal to this paper.

4 ARIADNE: STATE CONTINUITY VS. HARDWARE CRASHES

We present a second case study of monotonic state at work, this time verifying a protocol whose very purpose is to ensure a form of monotonicity. We may improve the resilience of systems confronted by unexpected failures (e.g., power loss) by having them persist their state periodically, and by restoring their state upon recovery. In this context, *state continuity* ensures that the state is restored from a recent backup, consistent with the observable effects of the system, rather than from a stale or fake backup that an attacker could attempt to replay or to forge.

For concreteness, we consider state continuity for Intel SGX enclaves. Such enclaves rely on a special CPU mode to protect some well-identified piece of code from the rest of the platform, notably from its host operating system. As long as the CPU is powered, the hardware automatically encrypts and authenticates every memory access, which ensures state confidentiality and integrity for the protected computation. For secure databases and many other applications, these guarantees must extend across platform crashes so as to ensure, for instance, that any commit that has been reported as complete (e.g., by witnessing its result) remains committed once the system resumes.

Several recent papers address this problem [Matetic et al. 2017; Parno et al. 2011; Strackx and Piessens 2016]. In this section, we present a first mechanized proof of Strackx and Piessens's *Ariadne* protocol using F⋆, relying on our libraries for monotonic state—the proof essentially consists of supplementing a hardware counter with a ghost state machine to track the recovery process (naturally expressed as a preorder) and then typechecking the recovery code. This example also illustrates the fairly natural combination of monotonic state with other effects like exceptions.

The Ariadne Protocol. The protocol relies on a single, hardware-based, non-volatile monotonic counter. Incrementing the counter is a privileged but unreliable operation. (It is also costly, hence the need to minimize increments.) If the operation returns, then the counter is incremented; if it crashes, the counter may or may not have been incremented. We model this behavior using

MSTExn, a combination of the MST monad and exceptions. The result of an MSTExn computation is either a normal result V v or an exception E e. In practice, our protocol code never throws or catches exceptions—these operations are available to the context only to model malicious hosts.

```
val incr: c:counter → MSTExn unit
 (ensures λh0 r h1 → if r=V() then ctr c h1 == ctr c h0 + 1 else (ctr c h1 == ctr c h0 ∨ ctr c h1 == ctr c h0 + 1))
```

The protocol also relies on a persistent but untrusted backup, modelled as a host function save. Before saving the backup, the current state is encrypted and authenticated together with a sequence number corresponding to the anticipated next value of the counter. The implementation of this authenticated encryption construction (using the auth_encrypt function below) is similar to what is presented in the §3.3; it makes use of a hardware-based key available only to the protected code. Conversely, the host presents an encrypted backup for recovery. Recovery from the last backup always succeeds, although it may be delayed by further crashes. Recovery from any other backup may fail, but must not break state-continuity; we focus on verifying this safety property of Ariadne.

Below we give the code for creating an Ariadne-protected reference cell and the two main operations for using it: store and recover. Their pre- and postconditions are given later, but at a high level, store requires a good state and recover does not. (There is no separate load from a good state, since the enclave can keep the state in volatile memory once it completes recovery.) For simplicity, we model hardware protection using a private datatype constructor: 'Protect c k' has type protected (defined shortly) and packages the enclave capabilities to use the monotonic counter c and backup key k. The recover function also takes as argument a (purportedly) last_saved backup, which is first authenticated and decrypted: if the decryption yields (m:ℕ,w:state) and m matches the current counter !c, then recovery continues with state w; otherwise it returns None.

```
let create (w:state) = let c = ref 0 in let k = keygen c in save (auth_encrypt k 0 w); Protect c k
let store (Protect c k) (w:state) = save (auth_encrypt k (!c+1) w); (*1*) incr c
let recover (Protect c k) last_saved =
  match auth_decrypt k !c last_saved with | None → None | Some (w:state) →
  save (auth_encrypt k (!c+1) w); (*2*) incr c; (*3*) save (auth_encrypt k (!c+1) w); (*4*) incr c; Some w
```

In this code, any particular call to save or incr may fail. To update the state, 'store' first encrypts and saves a backup of its new state (associated with the anticipated next value of the counter) and then increments the counter. This ordering of save and incr is necessary to enable recovery if a crash occurs at point *(*1*)* between these two operations, but may provide the host with several valid backups for recovery. For instance, a malicious host may obtain encryptions of both v and w at both !c and !c+1 by causing failure at *(*1*)*; recovering from the older backup; causing failure at *(*4*)*; then recovering and causing failure at *(*2*)*. This is fine as long as the recovery process eventually commits to either v or w before returning it. To this end, recovery actually performs *two* successive counter increments to clear up any ambiguity, and to ensure there is a unique backup at the current counter and no backup at the next counter. (On the other hand, completing recovery after its first increment would break continuity: with backups *at the current counter* for both v and w, the host has "forked" the enclave and can indefinitely get updated backups for both computations.) To capture this argument on the intermediate steps of the protocol, we supplement the real state of the counter (modeled here as an integer) with one out of 4 ghost cases, listed below.

```
type case =
  | Ok: saved:state → case (* clean state at the end of store or recover *)
  | Recover: read:state → other:state → case (* at step (3) above *)
  | Writing: written:state → old:state → case (* at steps (1) and (4) above *)
  | Crash: read:state → other:state → case (* worst case at step (2) of recovery, outlined above *)
```

Next, we give the specification of the protocol code, with 'ghost' selecting the current ghost case of a counter, and g `in_between` (v,w) stating that the ghost case g holds at most v and w.

```
val create: w:state → MSTExn protected (ensures λh0 r h1 → V? r ⟹ ghost h1 (V?.v r) == Ok v)

val store: p:protected → w:state → MSTExn unit (requires λh0 → Ok? (ghost h0 p))
  (ensures λh0 r h1 → match r with
                      | V () → ghost h1 p == Ok w
                      | E _ → ghost h1 p `in_between` (Ok?.saved (ghost h0 p),w))

val recover: p:protected → backup (Protect?.c p) → MSTExn (option state)
 (ensures λh0 r h1 → ∃v w. ghost h0 p `in_between` (v,w) ∧ match r with
                                                          | V (Some u) → ghost h1 p == Ok w ∧ u == w
                                                          | _          → ghost h1 p `in_between` (v,w))
```

To verify this specification, we introduce a 'saved' predicate that controls, in any given counter state (n,g), which backups (seqn,u) may have been encrypted and saved, and thus may be presented by the host for recovery. We use this predicate both to define our preorder on counter states, and to refine the type ℕ⋆state of general authenticated-encrypted values so as to define the type backup c of authenticated, encrypted, and saved backups associated with a given counter state c.

```
let saved (n:ℕ,g:case) (seqn:ℕ,u:state) = (* overapproximating what may have been encrypted and saved *)
  (seqn < n) ∨ (* an old state; authentication of `seqn` will fail, so nothing to say about it *)
  (seqn == n   ∧ (match g with | Ok v → u == v
                               | Recover w v | Writing w v | Crash w v → u == w ∨ u == v)) ∨
  (seqn == n+1 ∧ (match g with | Ok _ | Recover _ _ → ⊥
                               | Writing v _ → u == v
                               | Crash v w → u == v ∨ u == w))

let preorder (n0,g0) (n1,g1) = ∀s. saved (n0,g0) s ⟹ saved (n1,g1) s
let saved_backup c s h = h `contains` c ∧ saved (h.[c]) s
type backup c = s:(ℕ⋆state){witnessed (saved_backup c s)}

(* append-only ghost logs used to model the history of saving backups; we attach one to every backup key `k` *)
type log c = mref (list (backup c)) (λ l0 l1 → l0 `prefix_of` l1)
type protected = Protect: c:mref (ℕ⋆case) preorder → k:key c → protected
```

Our method for verifying the protocol is to use monotonicity to capture the history of saving backups, by augmenting the save function with witness (saved_backup c s) for the given authenticated-encrypted backup s, and the recover function with recall (saved_backup c last_saved) just after authenticated decryption of the (purportedly) last_saved backup the host provides for recovery. This allows us to recover at which stage of the protocol last_saved might have been created. We also instrument key points in the code with erasable modeling functions that operate on the ghost counter state. Typechecking enforces that all updates respect the preorder that ties the ghost state to the actual counter value. For instance, incr has two ghost transitions: (n, Crash w v) ⇝ (n+1, Recover w v) for typing call *(⋆2⋆)* above, and (n, Writing w v)⇝ (n+1, Ok w) for typing calls *(⋆1⋆)* and *(⋆4⋆)*. The other transitions update just the ghost case of the counter state. For instance, the only transition that introduces a new w:state is (n,Ok v) ⇝ (n,Writing w v), used in the store function before the call to encrypt the new backup. This yields a concise F⋆ proof of the safety of Ariadne's state continuity, possibly simpler than the original paper proof of Strackx and Piessens.

5 META-THEORY OF THE ABSTRACT MONOTONIC-STATE MONAD

While the applications of monotonic state are many, diverse, and sometimes quite complex, in the previous sections we showed how they can all be reduced to our general monotonic-state monad from §2.2 (MST in Figure 1 on page 6). In this section and the next, we put reasoning with this monotonic-state monad on solid foundations by presenting its meta-theory in two stages.

We first present a calculus called λ_{MST} to validate the soundness of witness and recall in a setting in which MST is treated abstractly. In fact, λ_{MST} is a fragment of another calculus, λ_{MST_b}, which we study in the next section (§6), and that extends λ_{MST} with support for safe, controlled monadic reflection and reification for revealing the representation of MST. Both calculi are designed to only capture the essence of reasoning with the monotonic-state monad—thus they do not include other advanced features of dependently typed languages, e.g., full dependency, refinement types, inductives, and universes. The complete definitions and proofs are available in the supplementary materials at https://fstar-lang.org/papers/monotonicity

5.1 λ_{MST}: A Dependently Typed Lambda Calculus with Support for witness and recall

The syntax of λ_{MST} is inspired by Levy's fine-grain call-by-value language [Levy 2004] in that it makes a clear distinction between values and computations. This set-up enables us to focus on stateful computations and validating their correctness, in particular for witness and recall actions.

First, *value types* t and *computation types* C are given by the following grammar:

$$t \quad ::= \quad \mathsf{state} \mid \mathsf{unit} \mid t_1 \times t_2 \mid t_1 + t_2 \mid x{:}t \to C \qquad C \quad ::= \quad \mathsf{MST}\ t\ (s.\ \varphi_{\text{pre}})\ (s\ x\ s'.\ \varphi_{\text{post}})$$

Here, state is a fixed abstract type of states, and $x{:}t \to C$ is a value-dependent function type, where the computation type C depends on values of type t. Similarly to Hoare Type Theory and F* surface syntax, λ_{MST} computation types $\mathsf{MST}\ t\ (s.\ \varphi_{\text{pre}})\ (s\ x\ s'.\ \varphi_{\text{post}})$ are indexed by pre- and postconditions: the former are given over initial state s, and the latter over initial state s, return value x, and final state s'. As a convention, we use variables s, s', s'', … to stand for states.

Next, *value terms* v and *computation terms* e are given by the following grammar:

$$\begin{array}{lll} v, \sigma & ::= & x \mid c \in S \mid () \mid (v_1, v_2) \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \lambda x{:}t.\ e \\ e & ::= & \mathsf{return}\ v \mid \mathsf{bind}\ x \leftarrow e_1\ \mathsf{in}\ e_2 \mid v_1\ v_2 \\ & \mid & \mathsf{pmatch}\ v\ \mathsf{as}_x\ (x_1, x_2)\ \mathsf{in}\ e \mid \mathsf{case}\ v\ \mathsf{of}_x\ (\mathsf{inl}(x_1)\ \mathsf{in}\ e_1; \mathsf{inl}(x_2)\ \mathsf{in}\ e_2) \\ & \mid & \mathsf{get} \mid \mathsf{put}\ \sigma \mid \mathsf{witness}\ s.\varphi \mid \mathsf{recall}\ s.\varphi \end{array}$$

Value terms are mostly standard, with S denoting a fixed non-empty set of state-valued constant symbols. As a convention, we use σ, σ', … to stand for value terms of type state.

Computation terms include returning values ($\mathsf{return}\ v$), sequential composition ($\mathsf{bind}\ x \leftarrow e_1\ \mathsf{in}\ e_2$), function applications ($v_1\ v_2$), and pattern matching for products (pmatch) and sums (case). In addition, computations include get and put actions for accessing the state and, finally, witness and recall actions parameterized by a first-order logic predicate $s.\varphi$ (where the variable s is bound in φ).

Formulas (φ) used for the pre- and postconditions, as well as in witness and recall, are drawn from a classical first-order logic extended with (1) a fixed preorder on states (rel), (2) typed equality on value terms (==), and (3) logical capabilities stating that a predicate was true in some prior program state and can be recalled in any reachable state (witnessed $s.\varphi$). The notation $s.\varphi$ defines a *predicate* on states by binding the variable s of type state in the formula φ.

$$\begin{array}{lll} p & ::= & \mathsf{rel} \mid == \\ \varphi & ::= & p\ \vec{v} \mid \top \mid \varphi_1 \wedge \varphi_2 \mid \bot \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \forall x{:}t.\ \varphi \mid \exists x{:}t.\ \varphi \mid \mathsf{witnessed}\ s.\varphi \end{array}$$

We take formulas to be in first-order logic (as opposed to Type-valued functions in F*) in order to keep the meta-theory simple and focused on stateful computations. Furthermore, this approach

$$\text{(T-Return)}\ \frac{\Gamma \vdash v : t}{\Gamma \vdash \mathsf{return}\ v : \mathsf{MST}\ t\ (s.\top)\ (s\ x\ s'.\ s == s' \wedge x == v)} \qquad \text{(T-App)}\ \frac{\Gamma \vdash v_1 : x{:}t \to C \qquad \Gamma \vdash v_2 : t}{\Gamma \vdash v_1\ v_2 : C[v_2/x]}$$

$$\text{(T-Bind)}\ \frac{\Gamma \vdash e_1 : \mathsf{MST}\ t_1\ (s.\varphi_{\mathrm{pre}})\ (s\ x_1\ s'.\ \varphi_{\mathrm{post}}) \quad \Gamma, x_1{:}t_1 \vdash e_2 : \mathsf{MST}\ t_2\ (s'.\ \exists s{:}\mathsf{state}.\ \varphi_{\mathrm{post}})\ (s'\ x_2\ s''.\ \varphi'_{\mathrm{post}})}{\Gamma \vdash \mathsf{bind}\ x_1 \leftarrow e_1\ \mathsf{in}\ e_2 : \mathsf{MST}\ t_2\ (s.\varphi_{\mathrm{pre}})\ (s\ x_2\ s''.\ \exists x{:}t_1.\ \exists s'{:}\mathsf{state}.\ \varphi_{\mathrm{post}} \wedge \varphi'_{\mathrm{post}})}$$

$$\text{(T-Get)}\ \frac{\vdash \Gamma\ \mathsf{wf}}{\Gamma \vdash \mathsf{get} : \mathsf{MST}\ \mathsf{state}\ (s.\top)\ (s\ x\ s'.\ s == x == s')} \qquad \text{(T-Put)}\ \frac{\Gamma \vdash \sigma : \mathsf{state}}{\Gamma \vdash \mathsf{put}\ \sigma : \mathsf{MST}\ \mathsf{unit}\ (s.\ \mathsf{rel}\ s\ \sigma)\ (s\ x\ s'.\ s' == \sigma)}$$

$$\text{(T-Witness)}\ \frac{\Gamma, s{:}\mathsf{state} \vdash \varphi\ \mathsf{wf} \qquad \mathsf{stable}\ s.\varphi =_{\mathrm{def}} \forall s'.\ \forall s''.\ \mathsf{rel}\ s'\ s'' \wedge \varphi[s'/s] \Longrightarrow \varphi[s''/s]}{\Gamma \vdash \mathsf{witness}\ s.\varphi : \mathsf{MST}\ \mathsf{unit}\ (s'.\ \mathsf{stable}\ s.\varphi \wedge \varphi[s'/s])\ (s'\ x\ s''.\ s' == s'' \wedge \mathsf{witnessed}\ s.\varphi)}$$

$$\text{(T-Recall)}\ \frac{\Gamma, s{:}\mathsf{state} \vdash \varphi\ \mathsf{wf}}{\Gamma \vdash \mathsf{recall}\ s.\varphi : \mathsf{MST}\ \mathsf{unit}\ (s'.\ \mathsf{witnessed}\ s.\varphi)\ (s'\ x\ s''.\ s' == s'' \wedge \varphi[s''/s])}$$

$$\text{(T-SubComp)}\ \frac{\Gamma \vdash e : C \quad \Gamma \vdash C <: C'}{\Gamma \vdash e : C'} \qquad \text{(Sub-MST)}\ \frac{\Gamma \vdash t <: t' \qquad \Gamma, s{:}\mathsf{state} \mid \varphi'_{\mathrm{pre}} \vdash \varphi_{\mathrm{pre}} \qquad \Gamma, s{:}\mathsf{state}, x{:}t, s'{:}\mathsf{state} \mid \varphi'_{\mathrm{pre}}, \mathsf{rel}\ s\ s', \varphi_{\mathrm{post}} \vdash \varphi'_{\mathrm{post}}}{\Gamma \vdash \mathsf{MST}\ t\ (s.\ \varphi_{\mathrm{pre}})\ (s\ x\ s'.\ \varphi_{\mathrm{post}}) <: \mathsf{MST}\ t'\ (s.\ \varphi'_{\mathrm{pre}})\ (s\ x\ s'.\ \varphi'_{\mathrm{post}})}$$

Fig. 2. Selected typing and subtyping rules for λ_{MST}

enables us to easily establish proof-theoretic properties of the logical witnessed-capabilities via a corresponding sequent calculus presentation (see §5.5). We use a classical logic to be faithful to F*'s SMT-logic of pre- and postconditions shown in our examples.

5.2 Static Semantics of λ_{MST}

We define the type system of λ_{MST} using judgments of well-formed contexts ($\vdash \Gamma$ wf), well-formed logical formulas ($\Gamma \vdash \varphi$ wf), well-formed value types ($\Gamma \vdash t$ wf), well-typed value terms ($\Gamma \vdash v : t$), well-formed computation types ($\Gamma \vdash C$ wf), and well-typed computation terms ($\Gamma \vdash e : C$). The rules defining the first five judgments are completely standard—we thus omit most of them and only give the formation rule for computation types:

$$\text{(Ty-MST)}\ \frac{\Gamma \vdash t\ \mathsf{wf} \quad \Gamma, s{:}\mathsf{state} \vdash \varphi_{\mathrm{pre}}\ \mathsf{wf} \quad \Gamma, s{:}\mathsf{state}, x{:}t, s'{:}\mathsf{state} \vdash \varphi_{\mathrm{post}}\ \mathsf{wf}}{\Gamma \vdash \mathsf{MST}\ t\ (s.\ \varphi_{\mathrm{pre}})\ (s\ x\ s'.\ \varphi_{\mathrm{post}})\ \mathsf{wf}}$$

The typing rules for computation terms are more interesting—we present a selection of them in Figure 2. The rules for the put, witness, and recall actions correspond directly to the typed interface from Figure 1 on page 6. Similarly to F*, λ_{MST} supports subtyping for value types ($\Gamma \vdash t <: t'$) and computation types ($\Gamma \vdash C <: C'$). Subtyping computation types (rule SubMST from Figure 2) is covariant in the postconditions, and contravariant in the preconditions. Moreover, logical entailment between the postconditions is proved assuming that the preconditions hold, and that the initial and final states are related by rel (recall that the state is only allowed to evolve according to rel).

We define logical entailment for formulas φ using a natural deduction system ($\Gamma \mid \Phi \vdash \varphi$, where Φ is a finite set of formulas). Most rules are standard for classical first-order logic, so Figure 3 only lists the rules that are specific to our setting. These include weakening for witnessed, reflexivity and transport for equality, reflexivity and transitivity for rel, and rules about the equality of pair and sum values. In more expressive logics, the latter rules are commonly derivable.

$$\frac{\Gamma, s{:}\mathsf{state} \mid \Phi \vdash \varphi \Longrightarrow \varphi' \quad s \notin FV(\Phi)}{\Gamma \mid \Phi \vdash (\mathsf{witnessed}\ s.\varphi) \Longrightarrow (\mathsf{witnessed}\ s.\varphi')}\ (\textsc{Witnessed-Weaken}) \qquad \frac{\Gamma \vdash v : t}{\Gamma \mid \Phi \vdash v == v}\ (\textsc{Eq-Refl}) \qquad \frac{\Gamma \mid \Phi \vdash v == v' \quad \Gamma \mid \Phi \vdash \varphi[v/x]}{\Gamma \mid \Phi \vdash \varphi[v'/x]}\ (\textsc{Eq-Transport})$$

$$\frac{\Gamma \vdash \sigma : \mathsf{state}}{\Gamma \mid \Phi \vdash \mathsf{rel}\ \sigma\ \sigma}\ (\textsc{Rel-Refl}) \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{rel}\ \sigma_1\ \sigma_2 \quad \Gamma \mid \Phi \vdash \mathsf{rel}\ \sigma_2\ \sigma_3}{\Gamma \mid \Phi \vdash \mathsf{rel}\ \sigma_1\ \sigma_3}\ (\textsc{Rel-Trans}) \qquad \frac{\Gamma \mid \Phi \vdash \mathsf{inl}\ v_1 == \mathsf{inr}\ v_2}{\Gamma \mid \Phi \vdash \bot}\ (\textsc{Sum-Disjoint}) \qquad \frac{\Gamma \mid \Phi \vdash (v_1, v_2) == (v_1', v_2')}{\Gamma \mid \Phi \vdash v_i == v_i'}\ (\textsc{Pair-Injective})$$

Fig. 3. Natural deduction (selected rules)

5.3 Instrumented Operational Semantics of λ_{MST}

We equip λ_{MST} with a small-step operational semantics that we instrument with a log of witnessed stable properties, which we use as additional logical assumptions in the correctness theorem we prove in §5.4, so as to accommodate the witness and recall actions, and their typing. Formally, we define the reduction relation $\rightsquigarrow$ on *configurations* (e, σ, W), where e is the computation term being reduced, σ is the current state value, and W is a finite set logging the stable predicates $s.\varphi$ witnessed so far. Most reduction rules are standard, so we only list the rules for actions below:

$$\begin{array}{rrcl}
(\textsc{Get}) & (\mathsf{get}, \sigma, W) & \rightsquigarrow & (\mathsf{return}\ \sigma, \sigma, W) \\
(\textsc{Put}) & (\mathsf{put}\ \sigma', \sigma, W) & \rightsquigarrow & (\mathsf{return}\ (), \sigma', W) \\
(\textsc{Witness}) & (\mathsf{witness}\ s.\varphi, \sigma, W) & \rightsquigarrow & (\mathsf{return}\ (), \sigma, W \cup s.\varphi) \\
(\textsc{Recall}) & (\mathsf{recall}\ s.\varphi, \sigma, W) & \rightsquigarrow & (\mathsf{return}\ (), \sigma, W)
\end{array}$$

The witness action adds the witnessed stable predicate to the log W, while get, put, and recall do not use the log at all. The get action returns the current state; the put action overwrites it.

5.4 Correctness for λ_{MST}

We now prove the correctness of our instrumented operational semantics in a Hoare-style program logic sense. In particular, we show that if $\vdash e : \mathsf{MST}\ t\ (s.\ \varphi_{\mathrm{pre}})\ (s\ x\ s'.\ \varphi_{\mathrm{post}})$, e reduces to return v, and φ_{pre} holds of the initial state, then φ_{post} holds of the initial state, the returned value v, and the final state. We also establish that the initial state is related to the final state, and that the logs can ever only increase. In order to better structure our proofs, we split them into progress and preservation theorems, which when combined give the above-mentioned result in Theorem 5.4.

A key ingredient of the following correctness results is the use of the instrumentation (log W of witnessed stable predicates) to provide additional logical assumptions corresponding to the logical witnessed capabilities resulting from earlier witness actions. In detail, these additional logical assumptions take the form $\mathsf{witnessed}\ W =_{\mathrm{def}} \mathsf{witnessed}\ s'.(\bigwedge_{s.\varphi \in W} \varphi[s'/s])$. In the results below, we also use well-formed state-log pairs $\Gamma \vdash (\sigma, W)\ \mathsf{wf}$, defined to hold iff $\Gamma \vdash \sigma : \mathsf{state}$ and

$$\Gamma \mid \mathsf{witnessed}\ W \vdash \varphi[\sigma/s] \qquad \Gamma \mid \mathsf{witnessed}\ W \vdash \mathsf{stable}\ s.\varphi \qquad (\mathsf{for\ all}\ s.\varphi \in W)$$

Theorem 5.1 (Progress). *If* $\vdash e : \mathsf{MST}\ t\ (s.\ \varphi_{pre})\ (s\ x\ s'.\ \varphi_{post})$ *then either* $\exists v.\ e = \mathsf{return}\ v$ *or* $\forall \sigma\ W.\ \exists e'\ \sigma'\ W'.\ (e, \sigma, W) \rightsquigarrow (e', \sigma', W')$.

Preservation crucially uses the well-formedness of (σ, W) to record that all previously witnessed stable predicates are in fact true of the current state, in combination with the witnessed W assumptions which ensure that, once obtained, logical capabilities remain usable in the future.

Theorem 5.2 (Preservation). *If* $\vdash e : \mathsf{MST}\ t\ (s.\ \varphi_{pre})\ (s\ x\ s'.\ \varphi_{post})$ *and* $(e, \sigma, W) \rightsquigarrow (e', \sigma', W')$ *such that* $\vdash (\sigma, W)\ \mathsf{wf}$ *and* $\mathsf{witnessed}\ W \vdash \varphi_{pre}[\sigma/s]$, *then*

(1) $W \subseteq W'$ *and* witnessed $W' \vdash$ rel $\sigma\ \sigma'$ *and* $\vdash (\sigma', W')$ wf *and*
(2) $\exists \varphi'_{pre}\ t'\ \varphi'_{post}.\ \vdash e' : \mathsf{MST}\ t'\ (s.\ \varphi'_{pre})\ (s\ x\ s'.\ \varphi'_{post})$ *and* $\vdash t' <: t$ *and* witnessed $W' \vdash \varphi'_{pre}[\sigma'/s]$ *and* $x{:}t', s'{:}$state $\mid$ witnessed W', rel $\sigma'\ s', \varphi'_{post}[\sigma'/s] \vdash \varphi_{post}[\sigma/s]$.

Proof. By induction on the sum of the height of the derivation of $(e, \sigma, W) \rightsquigarrow (e', \sigma', W')$ and the size of the computation term e, and by inverting the judgment $\vdash e : \mathsf{MST}\ t\ (s.\ \varphi_{\mathrm{pre}})\ (s\ x\ s'.\ \varphi_{\mathrm{post}})$ for each concrete e. Below we comment briefly on the more interesting cases of this proof.

Put: In this case, $e = \mathsf{put}\ \sigma''$, $\sigma' = \sigma''$, and $W' = W$, and we prove witnessed $W' \vdash$ rel $\sigma\ \sigma'$ by combining the assumption witnessed $W \vdash \varphi_{\mathrm{pre}}[\sigma/s]$ with s:state $\mid \varphi_{\mathrm{pre}} \vdash$ rel $s\ \sigma''$ that we get by inverting the typing of put σ''. We then derive $\vdash (\sigma', W')$ wf from $\vdash (\sigma, W)$ wf by using the stability of the witnessed predicates, proving witnessed $W \vdash \varphi[\sigma/s] \Longrightarrow \varphi[\sigma''/s]$ for all $s.\varphi \in W$.

Witness: In this case, $e = \mathsf{witness}\ s.\varphi$, $\sigma' = \sigma$, $W' = W \cup s.\varphi$, and $\varphi'_{\mathrm{pre}} = \mathsf{witnessed}\ s.\varphi$, and we prove witnessed $W' \vdash \varphi'_{\mathrm{pre}}[\sigma'/s]$ by using the Witnessed-Weaken rule from Figure 3.

Recall: In this case, $e = \mathsf{recall}\ s.\varphi$, $\sigma' = \sigma$, $W' = W$, and $\varphi'_{\mathrm{pre}} = \varphi$, and we are required to prove witnessed $W' \vdash \varphi'_{\mathrm{pre}}[\sigma'/s]$. We do so by combining the assumption $\vdash (\sigma, W)$ wf with the proof of s:state $\mid \bigwedge_{s'.\varphi' \in W} \varphi'[s/s'] \vdash \varphi$, which follows from the assumed precondition witnessed $W \vdash$ witnessed $s.\varphi$; this is a proof-theoretic property of the logic, see §5.5 for details. □

Finally, we combine progress and preservation results to prove partial correctness for λ_{MST}.

Proposition 5.3 (Correctness of return). *If* $\vdash \mathsf{return}\ v : \mathsf{MST}\ t\ (s.\ \varphi_{pre})\ (s\ x\ s'.\ \varphi_{post})$ *and* $\vdash \sigma :$ state, *then* $\vdash v : t$ *and* $\vdash \varphi_{pre}[\sigma/s] \Longrightarrow \varphi_{post}[\sigma/s, v/x, \sigma/s']$.

Theorem 5.4 (Partial correctness). *If* $\vdash e : \mathsf{MST}\ t\ (s.\ \varphi_{pre})\ (s\ x\ s'.\ \varphi_{post})$ *and we have a reduction sequence* $(e, \sigma, W) \rightsquigarrow^* (\mathsf{return}\ v, \sigma', W')$ *such that* $\vdash (\sigma, W)$ wf *and* witnessed $W \vdash \varphi_{pre}[\sigma/s]$, *then* $W \subseteq W'$ *and* witnessed $W' \vdash$ rel $\sigma\ \sigma'$, $\vdash v : t$ *and* witnessed $W' \vdash \varphi_{post}[\sigma/s, v/x, \sigma'/s']$.

We strengthen this to total correctness by also showing that λ_{MST} is strongly normalizing.

Theorem 5.5. *If* $\Gamma \vdash e : C$ *and* $\vdash (\sigma, W)$ wf, *then* (e, σ, W) *is strongly normalizing in* λ_{MST}.

Proof. The proof is based on defining a typing- and reduction structure preserving translation |-| of λ_{MST} types, terms, and configurations to a corresponding strongly normalizing simply typed calculus (by erasing type dependency, logical formulas, and logs, e.g., $|x{:}t \to C| =_{\mathrm{def}} |t| \to |C|$, $|\mathsf{witness}\ s.\varphi| =_{\mathrm{def}} \mathsf{witness}$[5], and $|(e, \sigma, W)| =_{\mathrm{def}} (|e|, |\sigma|)$). We establish the strong normalization of this simply typed calculus using the standard $\top\top$-lifting approach [Lindley and Stark 2005]. □

5.5 Proof-theoretic Properties of the Logic of Pre- and Postconditions

We conclude our investigation into the meta-theory of λ_{MST} by recalling that in the proof of Theorem 5.2, it was crucial (in the Recall case) to construct a derivation of s:state $\mid \varphi \vdash \varphi'$ from a derivation of witnessed $s.\varphi \vdash$ witnessed $s.\varphi'$. Intuitively, this proof-theoretic property of the logic means that φ' must be a logical consequence of φ because witnessed $s.\varphi \vdash$ witnessed $s.\varphi'$ could only have been proved using the natural deduction hypothesis and Witnessed-Weaken rules.

It is well known that establishing such properties directly in natural deduction is difficult due to the introduction rule for implication. We follow standard practice and turn to sequent calculus, which we define using judgment $\Gamma \mid \Phi \vdash \Phi'$, where Φ and Φ' are finite sets of formulas. Most rules are standard for classical sequent calculus, so Figure 4 only lists the more relevant ones. Importantly, the rules for $\Gamma \mid \Phi \vdash \Phi'$ do not include cut—we instead prove that it is admissible.

[5]To simplify the proof, the simply typed calculus includes computationally irrelevant computation terms witness and recall.

$$\text{(Eq-Refl-SC)}\ \frac{\Gamma \mid \Phi, v == v \vdash \Phi' \quad \Gamma \vdash v : t}{\Gamma \mid \Phi \vdash \Phi'} \qquad \text{(Eq-Transport-SC)}\ \frac{\Gamma \mid \Phi, (p\ \vec{v})[v''/x] \vdash \Phi' \quad \Gamma \vdash v' : t \quad \Gamma \vdash v'' : t}{\Gamma \mid \Phi, v == v', (p\ \vec{v})[v'/x] \vdash \Phi'}$$

$$\text{(Witnessed-Weaken-SC)}\ \frac{\Gamma, s{:}\mathsf{state} \mid \Phi, \varphi \vdash \varphi', \Phi' \quad s \notin FV(\Phi) \quad s \notin FV(\Phi')}{\Gamma \mid \Phi, \mathsf{witnessed}\ s.\varphi \vdash \mathsf{witnessed}\ s.\varphi', \Phi'}$$

Fig. 4. Sequent calculus (selected rules)

Theorem 5.6 (Admissibility of cut). *The cut rule is admissible in this sequent calculus.*

In order to accommodate the rules concerning == and rel from Figure 3, while at the same time ensuring that cut remains admissible, we follow Negri and von Plato [1998] in defining the corresponding rules in the sequent calculus such that they only modify the left-hand side of the entailment judgment—see the equality rules in Figure 4. Also following Negri and von Plato, we restrict the Eq-Transport-SC rule to atomic predicates (the general rule is admissible).

Next, we relate this sequent calculus to our natural deduction system in the standard way, e.g., if $\Gamma \mid \Phi \vdash \Phi'$ in the sequent calculus, then $\Gamma \mid \Phi \vdash \bigvee_{\varphi \in \Phi'} \varphi$ in natural deduction, and vice versa. Based on the equivalence of the systems, and the syntax directed nature of the sequent calculus rules, we finally prove the property of the logical witnessed capabilities we used in the proof of Theorem 5.2.

Theorem 5.7. *If we have a derivation of* $\Gamma \mid \Phi, \mathsf{witnessed}\ s.\varphi \vdash \mathsf{witnessed}\ s.\varphi'$ *in the sequent calculus such that* $s \notin FV(\Phi)$ *and* Φ *contains only formulas* $p\ \vec{v}$*, then we have* $\Gamma, s{:}\mathsf{state} \mid \Phi, \varphi \vdash \varphi'$.

Another standard consequence of cut admissibility is logical *consistency*: the addition of the rules concerning witnessed, ==, and rel do not make our natural deduction system inconsistent.

6 REVEALING THE REPRESENTATION OF THE MONOTONIC-STATE MONAD

So far, the key ingredient for ensuring that the witness and recall actions are sound has been the abstract treatment of MST. In particular, as illustrated in §2.2, carelessly breaking the abstraction and revealing MST computations as pure state-passing functions quickly leads to unsoundness in the presence of witness and recall. In this section, we present our full formal dependently typed lambda calculus λ_{MST_b} (as a delta relative to λ_{MST} that we studied in §5), which also supports revealing the representation of computations in a controlled and provably safe way.

6.1 Reify and Reflect, by Example

Our starting point is to introduce two coercions for soundly exposing the representation of the monotonic-state monad: reify for revealing the representation of an MST computation as a state-passing function; and reflect that turns a state-passing function into an MST computation [Ahman et al. 2017; Filinski 1994]. Then, we carefully arrange the λ_{MST_b} calculus so that the typing of computations that rely on the representation of MST effectively treat the witness and recall actions as no-ops. By following this approach, we prevent the inconsistencies sketched in §2.2, yet allow several interesting applications based on the pure state-passing representation of MST.

Reification is Useful. One useful application of reify is to prove relational properties of stateful programs [Grimm et al. 2017]. Consider a variant of the monotonic counter from §2.2 that we can increment by an arbitrary positive amount. We would like to be able to show that a simple computation branching on a secret boolean h and then either incrementing the counter by 2 or incrementing it twice by 1 (see incr2 below) does not leak information about h into the public counter, a noninterference property proven by the assertion on reified terms on the last line:

```
let incr (n:ℕ) : MST unit = put (get() + n)
let incr2 (h:bool) : MST unit = if h then incr 2 else (incr 1; incr 1)
assert (∀ h0 h1 c. reify (incr2 h0) c = reify (incr2 h1) c) (* noninterference *)
```

In particular, observe that by turning an abstract MST computation into a pure state-passing function, reify serves two purposes in this example: on the one hand, it enables one to apply the MST computation incr2 to an initial state argument c; on the other hand, it allows this MST computation to be used in the assertion where only pure (effect-free) programs are allowed in F*.

Reflection is Useful. The dual of reification, monadic reflection, enables users to extend effect interfaces with new actions defined as pure state-passing functions. For instance, suppose we want to add a new monadic action for squaring the value of a monotonic counter. The counter interface might not provide all the operations for conveniently implementing this (e.g., it might only provide incr and is_above n), but if the interface exposes reflect, then we can easily implement the squaring action from scratch. e.g., let square () : MST unit = reflect (λs $\rightarrow$ ((), s $*$ s)).

We expect this kind of reasoning to be sound, certainly when, as in the examples above, witness and recall are not used at all. However, even slightly more complex examples implicitly rely on witness/recall, though this may not always be immediate, e.g., programs using typed references implicitly use recall at dereferencing. Yet, allowing unrestricted combinations of reify, reflect, witness, and recall immediately enables counterexamples (§2.2). Our type system of λ_{MST_b} ensures that programs relying on the representation of MST gain no benefit from using witness and recall.

6.2 λ_{MST_b}: Syntax and Static Semantics

The main idea of λ_{MST_b} is to simultaneously provide two version of the MST computation type, one which is abstract, and another whose representation is exposed. As such, we propose an *indexed computation type*, MST_b, where the boolean b signifies whether or not the computation is "representation aware" (b = true) or abstract (b = false), i.e., the abstract MST computation type of §5 is really just $\text{MST}_{\text{false}}$. Moreover, reusable MST_b computations can be parameterized over an arbitrary boolean b, and so will be usable with both values of b, similar to an intersection type.

The value types t of λ_{MST_b} are identical to λ_{MST}; and we encode the type of booleans simply as bool $=_{\text{def}}$ unit + unit. As a convention, we use b to range over value terms of type bool. More interestingly, λ_{MST_b}'s computation types C are given by the following extended grammar:

$$C \quad ::= \quad \text{Pure } t\ (\varphi_{\text{pre}})\ (x.\ \varphi_{\text{post}}) \ \mid\ \text{MST}_b\ t\ (s.\ \varphi_{\text{pre}})\ (s\ x\ s'.\ \varphi_{\text{post}})$$

To give strong types to the state-passing functions involved in reify and reflect, we follow F* and introduce Pure, the type of pure, effect-free computations [Ahman et al. 2017], enabling us to reason about pure programs in terms of pre- and postconditions using the same logic as in §5.

The value and computation terms of λ_{MST_b} are also a minor extension of λ_{MST}'s terms. In particular, λ_{MST_b} also includes a value term for monadic reification (reify e), and a computation term for monadic reflection (reflect v). We also provide a coercion from representation-aware to abstract computations (coerce e, also a computation term), to be able to reuse a reflected state-passing function in a context expecting an abstract computation.

As the monotonic-state monad is now indexed by a boolean b, the get, put, witness, and recall actions, and the return of MST_b, all take an additional boolean-valued argument. Nevertheless, computation terms in λ_{MST_b} include return v to return values in the Pure computation type.

$$\begin{array}{lcl} v & ::= & \ldots \mid \text{reify } e \\ e & ::= & \ldots \mid \text{return}_b\ v \mid \text{get}_b \mid \text{put}_b\ \sigma \mid \text{witness}_b\ s.\varphi \mid \text{recall}_b\ s.\varphi \mid \text{reflect } v \mid \text{coerce } e \end{array}$$

$$\text{(T-Witness)}\quad \frac{\Gamma \vdash b : \mathsf{bool} \qquad \Gamma, s{:}\mathsf{state} \vdash \varphi\ \mathsf{wf} \qquad \mathsf{stable}\ s.\varphi =_{\mathrm{def}} \forall s'.\ \forall s''.\ \mathsf{rel}\ s'\ s'' \wedge \varphi[s'/s] \Longrightarrow \varphi[s''/s]}{\begin{array}{l}\Gamma \vdash \mathsf{witness}_b\ s.\varphi : \mathsf{MST}_b\ \mathsf{unit}\ (s'.\ (b == \mathsf{false}) \Longrightarrow (\mathsf{stable}\ s.\varphi \wedge \varphi[s'/s])) \\ \qquad\qquad (s'\ x\ s''.\ s' == s'' \wedge ((b == \mathsf{false}) \Longrightarrow \mathsf{witnessed}\ s.\varphi))\end{array}}$$

$$\text{(T-Recall)}\quad \frac{\Gamma \vdash b : \mathsf{bool} \qquad \Gamma, s{:}\mathsf{state} \vdash \varphi\ \mathsf{wf}}{\begin{array}{l}\Gamma \vdash \mathsf{recall}_b\ s.\varphi : \mathsf{MST}_b\ \mathsf{unit}\ (s'.\ (b == \mathsf{false}) \Longrightarrow \mathsf{witnessed}\ s.\varphi) \\ \qquad\qquad (s'\ x\ s''.\ s' == s'' \wedge ((b == \mathsf{false}) \Longrightarrow \varphi[s''/s]))\end{array}}$$

$$\text{(T-Coerce)}\quad \frac{\Gamma \vdash e : \mathsf{MST}_{\mathsf{true}}\ t\ (s.\ \varphi_{\mathrm{pre}})\ (s\ x\ s'.\ \varphi_{\mathrm{post}})}{\Gamma \vdash \mathsf{coerce}\ e : \mathsf{MST}_{\mathsf{false}}\ t\ (s.\ \varphi_{\mathrm{pre}})\ (s\ x\ s'.\ \varphi_{\mathrm{post}})}$$

$$\text{(T-Reify)}\quad \frac{\Gamma \vdash e : \mathsf{MST}_{\mathsf{true}}\ t\ (s.\ \varphi_{\mathrm{pre}})\ (s\ x\ s'.\ \varphi_{\mathrm{post}})}{\Gamma \vdash \mathsf{reify}\ e : s{:}\mathsf{state} \to \mathsf{Pure}\ (t \times \mathsf{state})\ (\varphi_{\mathrm{pre}})\ (y.\ \exists x.\ \exists s'.\ y == (x, s') \wedge \mathsf{rel}\ s\ s' \wedge \varphi_{\mathrm{post}})}$$

$$\text{(T-Reflect)}\quad \frac{\Gamma \vdash v : s{:}\mathsf{state} \to \mathsf{Pure}\ (t \times \mathsf{state})\ (\varphi_{\mathrm{pre}})\ (y.\ \exists x.\ \exists s'.\ y == (x, s') \wedge \mathsf{rel}\ s\ s' \wedge \varphi_{\mathrm{post}})}{\Gamma \vdash \mathsf{reflect}\ v : \mathsf{MST}_{\mathsf{true}}\ t\ (s.\ \varphi_{\mathrm{pre}})\ (s\ x\ s'.\ \varphi_{\mathrm{post}})}$$

Fig. 5. Selected typing rules for λ_{MST_b}

Figure 5 presents the main differences to the typing rules of λ_{MST_b} relative to λ_{MST}, where parts typeset in gray apply only to abstract $\mathsf{MST}_{\mathsf{false}}$ computations. The T-Witness and T-Recall rules now guard their pre- and postconditions with the boolean b. As noted previously, for representation-aware computations ($b = \mathsf{true}$), witness and recall are just no-ops, as seen by focusing only on the non-gray parts of the rules. Since representation-aware computations cannot make meaningful use of monotonicity, the T-Coerce rule allows them to be promoted to abstract MST computations—a coercion in the other direction would immediately break soundness. Finally, the postcondition of the state-passing function type used in the T-Reify and T-Reflect rules crucially captures the monotonic evolution of state. For reify, this is an important invariant that MST computations provide, which we can *rely* on when their representation is revealed. Dually, for reflect, representation-aware code must *guarantee* to preserve an abstract computation's monotonic view of the state.

6.3 Instrumented Operational Semantics of λ_{MST_b}

We equip λ_{MST_b} with an instrumented operational semantics that is a minor extension of λ_{MST}'s semantics from §5.3. The reduction relation $\rightsquigarrow$ again works on *configurations* (e, σ, W). We give the new reduction rules in Figure 6. Importantly, λ_{MST_b} has two rules for the witness action: Witness-False is just the Witness reduction rule of §5.3, whereas Witness-True states that witness is a no-op in representation-aware code. The Reify-Return and Reify-Context rules make it precise that reify e is a pure state-passing function that behaves exactly like the given stateful computation e, when applied to a state σ. In particular, the Reify-Return rule says that the reification of $\mathsf{return}_{\mathsf{true}}\ v$ is essentially the same as the pure state-passing function '$\lambda s{:}\mathsf{state}.\ \mathsf{return}\ (v, s)$'; and the Reify-Context rule takes care of get, put, etc., by evaluating them as stateful computations.

The Reflect-Return and Reflect-Context rules describe that reflect v behaves exactly like the given pure state-passing function v. In particular, in Reflect-Context we evaluate the given function $\lambda s{:}\mathsf{state}.\ e$ by first substituting the initial state σ into its body, and by then reducing the resulting pure computation term. The computation term e' (the reduct) is no longer dependent on the lambda-bound variable, since the initial substitution of σ into e removes all occurrences of the original bound variable. Finally, when the state-passing function finishes reducing under

$$
\begin{array}{rrcl}
\textsc{(Witness-False)} & (\mathsf{witness}_{\mathsf{false}}\ s.\varphi, \sigma, W) & \rightsquigarrow & (\mathsf{return}_{\mathsf{false}}\ (), \sigma, W \cup s.\varphi) \\
\textsc{(Witness-True)} & (\mathsf{witness}_{\mathsf{true}}\ s.\varphi, \sigma, W) & \rightsquigarrow & (\mathsf{return}_{\mathsf{true}}\ (), \sigma, W) \\
\textsc{(Reify-Return)} & ((\mathsf{reify}\ (\mathsf{return}_{\mathsf{true}}\ v))\ \sigma, \sigma', W) & \rightsquigarrow & (\mathsf{return}\ (v, \sigma), \sigma', W) \\
\textsc{(Reflect-Reify)} & (\mathsf{reflect}\ (\mathsf{reify}\ e), \sigma, W) & \rightsquigarrow & (e, \sigma, W) \\
\textsc{(Reflect-Return)} & (\mathsf{reflect}\ (\lambda s{:}\mathsf{state}.\ \mathsf{return}\ (v, \sigma')), \sigma, W) & \rightsquigarrow & (\mathsf{return}_{\mathsf{true}}\ v[\sigma/s], \sigma'[\sigma/s], W) \\
\textsc{(Coerce-Return)} & (\mathsf{coerce}\ (\mathsf{return}_{\mathsf{true}}\ v), \sigma, W) & \rightsquigarrow & (\mathsf{return}_{\mathsf{false}}\ v, \sigma, W)
\end{array}
$$

$$
\textsc{(Reify-Context)}\ \frac{(e, \sigma, W) \rightsquigarrow (e', \sigma', W')}{((\mathsf{reify}\ e)\ \sigma, \sigma'', W) \rightsquigarrow ((\mathsf{reify}\ e')\ \sigma', \sigma'', W')}
$$

$$
\textsc{(Reflect-Context)}\ \frac{(e[\sigma/s], \sigma, W) \rightsquigarrow (e', \sigma', W')}{(\mathsf{reflect}\ (\lambda s{:}\mathsf{state}.\ e), \sigma, W) \rightsquigarrow (\mathsf{reflect}\ (\lambda \_{:}\mathsf{state}.\ e'), \sigma', W')}
$$

Fig. 6. Selected reduction rules for λ_{MST_b}

reflect, the Reflect-Return rule replaces the current program state with the state computed by the pure state-passing function. Together, Reflect-Context and Reflect-Return ensure that a reflected computation's effect on the state σ occur atomically, i.e., the intermediate states of pure, state-passing function are not observable and need not respect the preorder, although, by T-Reflect, taken as a single state transition, a reflected function is preorder compliant.

6.4 Correctness for λ_{MST_b}

We establish a partial correctness result for λ_{MST_b}, similar to the one proved for λ_{MST} in §5.4. We also structure our correctness proof using progress and preservation theorems; as they are mostly analogous to §5.4, we omit them here and only note the main differences: (1) in order to use the well-typedness of the substitution $e[\sigma/s]$ for the Reflect-Context rule, we consider only well-typed initial states in the progress theorem; (2) Pure reduction steps do not modify the state and the log of witnessed stable predicates; and (3) $\mathsf{MST}_{\mathsf{true}}$ reduction steps to not change the log.

Theorem 6.1 (Partial correctness, for Pure). *If* $\vdash e : \mathsf{Pure}\ t\ (\varphi_{pre})\ (x.\ \varphi_{post})$ *and we have a reduction sequence* $(e, \sigma, W) \rightsquigarrow^* (\mathsf{return}\ v, \sigma', W')$ *such that* $\vdash (\sigma, W)\ \mathsf{wf}$ *and* $\mathsf{witnessed}\ W \vdash \varphi_{pre}$, *then* $W = W'$ *and* $\sigma = \sigma'$ *and* $\vdash v : t$ *and* $\mathsf{witnessed}\ W' \vdash \varphi_{post}[\sigma/s, v/x, \sigma'/s']$.

Theorem 6.2 (Partial correctness, for MST_b). *If* $\vdash e : \mathsf{MST}_b\ t\ (s.\ \varphi_{pre})\ (s\ x\ s'.\ \varphi_{post})$ *and* $(e, \sigma, W) \rightsquigarrow^* (\mathsf{return}\ v, \sigma', W')$ *such that* $\vdash (\sigma, W)\ \mathsf{wf}$ *and* $\mathsf{witnessed}\ W \vdash \varphi_{pre}[\sigma/s]$, *then* $W \subseteq W'$ *(*$W = W'$ *if* $b = \mathsf{true}$*) and* $\mathsf{witnessed}\ W' \vdash \mathsf{rel}\ \sigma\ \sigma'$ *and* $\vdash v : t$ *and* $\mathsf{witnessed}\ W' \vdash \varphi_{post}[\sigma/s, v/x, \sigma'/s']$.

6.5 Discussion

We conclude by observing that, while λ_{MST_b} enables us to mix monotonic-state computations and reification-reflection in a sound way, the situation is not entirely satisfactory. Any MST_b computation that is parametric in b essentially has to be equipped with two specifications, one making use of on monotonicity and employing the state-independent witnessed $s.\varphi$ propositions, and the other using the corresponding predicates $s.\varphi$ as stateful invariants, so as to compensate for witness and recall being treated as no-ops when $b == \mathsf{true}$. Instead, we expect it may be possible to avoid such duplication, with abstract monotonic-state computations given a single specification using state-independent witnessed $s.\varphi$ propositions, as in §5, from which the corresponding specification of the underlying representation (using predicates $s.\varphi$ as stateful invariants) would then be derived automatically. We leave such a solution for future work and further discuss it in relation to hybrid modal logic in §8.

7 RELATED WORK

We have covered several strands of related work throughout the paper. As discussed earlier, the closest related work to ours is Swamy et al.'s [2013b], who propose and use in F* (but do not formalize) a simple form of the witness/recall style that we generalize, formalize, implement, and illustrate in this paper. However, reasoning about monotonic state has a longer tradition—we discuss some closely related, recent efforts below.

The Essence of Monotonic State. Pilkiewicz and Pottier [2011] devise a type system based on linear and non-linear capabilities to model monotonic state. Their *fates* are linear capabilities constrained to evolve by a preorder that are associated with, and describe invariants of, fragments of the state. Due to their linearity, programs can gain ownership of a fate, update the associated state, restoring monotonicity without allowing intermediate, potentially non-monotonic state transitions to be observable. Analogous to our stable predicates, the non-linear capabilities in their system (called *predictions*) are properties that are stable with respect to the evolution of a fate. In contrast, our MST monad is based on a simpler, more abstract, global state, deriving notions like typed references which Pilkiewicz and Pottier take as primitive. On the other hand, our technique lacks any specific support for linearity or ownership; it would be interesting to investigate this in the future.

Rely-Guarantee References. Gordon et al. [2013] devise a monadic DSL in Coq with an *axiomatic* heap and typed references, where, like our monotonic references, a reference is associated with a *guarantee* relation constraining how its values may evolve. Whereas in our 'mref a rel' the relation is a preorder on a, Gordon et al. allow their guarantee relation to also mention the entire heap. This appears to be a middle ground between global preorders on the entire state and the per-mref preorders. Gordon et al. also decorate references with a *rely* relation, analogous to our stable predicates, that are stable with respect to guarantee. They also include aliasing control whereby multiple references to the same data can provide different, but compatible, rely-guarantee relations.

PCM-Based Separation Logics. Starting with the Fictional Separation Logic of Jensen and Birkedal [2012], many modern (concurrent) separation logics (e.g., Krebbers et al.'s [2017] Iris) use partial commutative monoids (PCMs) to allow users to define their own problem-specific abstractions of the physical state. These PCM-based separation logics are very general, and can be used to encode many styles of spatial and temporal reasoning about programs, including reasoning about monotonic state. For example, Jensen and Birkedal show how to use their logic to reason about a monotonic counter c using a (freely duplicable) predicate MC(c,i) from which one can conclude that the value of the counter c is at least i, analogously to how we use the logical capability witnessed (λc $\rightarrow$ i $\leq$ c) to reason about monotonic counters in §2.2.

This PCM-based model of monotonic counters is however somewhat imprecise, in that it only supports reasoning about monotonic counters using stable predicates, rather than freely mixing stable predicates with precise reasoning about the concrete value of the counter. In particular, after incrementing the counter c, one can prove MC(c,i + 1) but not that the new value of the counter is exactly 1 greater than its previous value. In order to reason using both precise values and stable predicates, following discussions with an author of Iris, it seems necessary to move to a richer PCM that encodes a finer model of permissions, supporting both duplicable, stable observations as well as exclusively owned, precise assertions about the same resource, such as a monotonic counter. As such, even simple uses of monotonic reasoning in PCM-based logics seem to require some sophisticated encodings, even if staying in their sequential fragment. In contrast, we value the simplicity of our classical Hoare logic based verification model, recognizing that while PCMs are more general, our model is easy to use, benefits from SMT automation, and scales well.

State-Machine Based Separation Logics. Some concurrent separation logics, such as CAP [Dinsdale-Young et al. 2010] and its successors, use state machines to control how the program state is allowed to evolve. For instance, Sergey et al. [2018] recently devised a variant of Hoare Type Theory for distributed protocols, that includes primitive support for distributed ghost state governed by state machines. This is analogous to our use of preorders in the monotonic-state monad: in §3.2 we show how distributed state and state machines can be modeled using MST. However, all separation logics that employ state machines of which we are aware have explicit side-conditions on their reasoning rules requiring that all employed predicates are stable with respect to taking transitions in the state machine, ruling out mixing monotonic reasoning about stable predicates with precise reasoning about non-stable ones when concurrent state updates are not a concern.

Reasoning About Object-Oriented Programs. Monotonicity is also of importance in the verification of object-oriented programs, where reflexive-transitive two-state invariants [Cohen et al. 2010; Leino and Schulte 2007] and update guards [Barnett and Naumann 2004; Polikarpova et al. 2014] are used to constrain how one is allowed to update an object, enabling one to establish that any consistent object depending on a consistent object O remains consistent after updates to O. Also, similarly to our use of witness and recall to simplify the verification of stateful sequential programs, these works simplify the verification of object-oriented programs further by allowing an object O to be explicitly marked as *packed*, asserting that O is consistent and indicating that it remains so irrespective of updates to unrelated objects; and as *unpacked*, indicating that O is susceptible to consistency-breaking updates. As such, the use of packing and unpacking in these works is also similar to our use of snapshots to temporarily escape a given preorder in §2.2.

Other Related Works. For example, Bengtson et al.'s [2011] RCF calculus takes as primitive a monotonic log of formulas to represent the distributed state of a protocol; and Chajed et al.'s [2017] Crash Hoare Logic is designed for reasoning about program correctness in the presence of failure and recovery. We expect the latter could be used to reason about the Ariadne protocol (§4), although, lacking monotonic state, we expect the proof would have to be conducted using stateful invariants.

8 CONCLUSION AND FUTURE WORK

In this paper, we have provided a practical way to ease the verification of programs whose state evolves monotonically. In summary, the main distinctive contributions of our work are:

- A new, simple, core design for reasoning about stable predicates on monotonic global state, namely, the *monotonic-state monad* MST, together with its witness and recall actions.
- A demonstration that despite its simplicity, MST is general enough to account for diverse forms of monotonic state, capturing both *language-level* (e.g., a memory model with typed references) and *application-level* (e.g., ghost distributed state) monotonicity.
- An application of MST to two substantial case studies: we provide machine-checked proofs of a secure file-transfer application, and the recent Ariadne state-continuity protocol.
- A formalization of the monotonic-state monad in the context of a higher-order, dependently typed calculus, showing how to support both *intrinsic* proofs when treating MST abstractly, and *extrinsic* proofs when carefully revealing the underlying representation of MST.

These contributions are focused on controlling the temporal evolution of state in sequential programs, leaving spatial properties to be handled by existing techniques. As discussed in §7, some strands of related work profitably combine reasoning about the temporal evolution of state with concurrency and various forms of aliasing control, an area that we hope to investigate in the future. We are also currently working on extending F* with indexed effects, so as to make programming and reasoning about MST computations more convenient, allowing MST to be indexed in F* by state

types, preorders, and booleans. In addition, while our monotonic references support programming with multiple *local* preorders, we plan to investigate allowing different parts of the programs to also modularly make use of different *global* preorders, e.g., by using a graded-monads-style ordered monoidal structure [Katsumata 2014] on the preorder index of MST. Finally, we are also working on making formal the connection between the intuitive holds-everywhere-in-the-future reading of our witnessed logical capability and the standard $\Box$ necessity modality of modal logics. In particular, we conjecture that if one were to use a hybrid modal logic in which one can bind the "current" modal world in the assertions, such as Reed's [2009] Hybrid LF, witnessed $s.\varphi$ could be expressed as the combination of the $\Box$ modality and this hybrid binding operation, i.e., as $\Box(\downarrow s.\varphi)$. We hope that such a modal treatment of witnessed will also address the duplication of specifications issue discussed in §6.5.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers and Nadia Polikarpova for suggesting many ways in which to improve our work. We also thank David Baelde, Derek Dreyer, Gordon Plotkin, François Pottier, and the Project Everest team for many useful discussions. Danel Ahman's work was done, in part, during an internship at Microsoft Research. The work of Danel Ahman, Cătălin Hriţcu, and Kenji Maillard is supported, in part, by the European Research Council under ERC Starting Grant SECOMP (715753).

REFERENCES

D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017.

M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. *MPC*. 2004.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Trans. Prog. Lang. Syst. (TOPLAS)*, 33(2):8, 2011.

K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. *POPL*, 2010.

K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hriţcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pang, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Béguelin, and J.-K. Zinzindohoué. Everest: Towards a verified, drop-in replacement of HTTPS. *SNAPL*, 2017a.

K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. *IEEE Security & Privacy*, 2017b.

T. Chajed, H. Chen, A. Chlipala, M. F. Kaashoek, N. Zeldovich, and D. Ziegler. Certifying a file system using crash hoare logic: correctness in the presence of crashes. *Commun. ACM*, 60(4):75–84, 2017.

K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

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

E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. *CAV*. 2010.

T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. *ECOOP*. 2010.

A. Filinski. Representing monads. *POPL*. 1994.

C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee references for refinement types over aliased mutable data. *PLDI*. 2013.

N. Grimm, K. Maillard, C. Fournet, C. Hriţcu, M. Maffei, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, and S. Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. arXiv:1703.00055, 2017.

S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. *POPL*. 2001.

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

I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. *FM*. 2006.

S. Katsumata. Parametric effect monads and semantics of effect systems. *POPL*. 2014.

R. Krebbers, R. Jung, A. Bizjak, J. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. *ESOP*. 2017.

K. R. M. Leino and W. Schulte. Using history invariants to verify observers. *ESOP*. 2007.

X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1):1–31, 2008.

P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.

S. Lindley and I. Stark. Reducibility and tt-lifting for computation types. *TLCA*. 2005.

T. Lindvall. *Lectures on the Coupling Method*. Dover Books on Mathematics Series. Dover Publications, Incorporated, 2002.

S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, and A. Gervai. Rote: Rollback protection for trusted execution. *USENIX Security*. 2017.

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

S. Negri and J. von Plato. Cut elimination in the presence of axioms. *Bulletin of Symbolic Logic*, 4(4):418–435, 1998.

B. Parno, J. R. Lorch, J. R. Douceur, J. W. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. *S&P*. 2011.

A. Pilkiewicz and F. Pottier. The essence of monotonic state. *TLDI*. 2011.

N. Polikarpova, J. Tschannen, C. A. Furia, and B. Meyer. Flexible invariants through semantic collaboration. *FM*. 2014.

J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *ICFP*, 2017.

X. Qi and A. C. Myers. Masked types for sound object initialization. *POPL*. 2009.

J. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, 2009.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *LICS*. 2002.

I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols. *POPL*, 2018.

R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. *USENIX Security*. 2016.

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *JFP*, 23(4):402–451, 2013a.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. *PLDI*, 2013b.

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

W. Swierstra. *A functional specification of effects*. PhD thesis, University of Nottingham, UK, 2009.

J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. *CCS*, 2017.