

Embracing monotonicity in



Danel Ahman @ INRIA Paris

joint work with

Cătălin Hrițcu and Kenji Maillard @ INRIA Paris

Cédric Fournet, Aseem Rastogi, and Nikhil Swamy @ MSR

(based on our POPL 2018 paper)

ICE-TCS Seminar

January 29, 2018

Outline

- F^*
- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F^*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

Outline

- F^*
- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F^*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

- **F*** is
 - a **functional programming language**
 - ML, OCaml, F#, Haskell, ...
 - extracted to OCaml or F#; subset compiled to efficient C code
 - an **interactive proof assistant**
 - Agda, Coq, Lean, Isabelle/HOL, ...
 - interactive modes for Emacs and Atom
 - a **semi-automated verifier** of imperative programs
 - Dafny, Why3, FramaC, ...
 - Z3-based SMT-automation; tactics and metaprogramming (WIP)
- **Application-driven development**
 - Project Everest [project-everest.github.io]
 - Microsoft Research (US, UK, India), INRIA (Paris), ...
 - miTLS, HACL*, Vale, ...

- **F*** is
 - a **functional programming language**
 - ML, OCaml, F#, Haskell, ...
 - extracted to OCaml or F#; subset compiled to efficient C code
 - an **interactive proof assistant**
 - Agda, Coq, Lean, Isabelle/HOL, ...
 - interactive modes for Emacs and Atom
 - a **semi-automated verifier** of imperative programs
 - Dafny, Why3, FramaC, ...
 - Z3-based SMT-automation; tactics and metaprogramming (WIP)
- **Application-driven** development
 - Project Everest [project-everest.github.io]
 - Microsoft Research (US, UK, India), INRIA (Paris), ...
 - miTLS, HACl*, Vale, ...

F* – a prog. lang./proof assistant/verifier

```
module Talk

// Inductive types

type vector 'a : nat -> Type =
  | Nil : vector 'a 0
  | Cons : #n:nat -> 'a -> vector 'a n -> vector 'a (n + 1)

// Dependently typed recursive functions

val append : #a:Type -> #n:nat -> #m:nat -> vector a n -> vector a m -> Tot (vector a (n + m))
let rec append #a #n #m xs ys =
  match xs with
  | Nil -> ys
  | Cons #n x xs -> Cons x (append xs ys)

// Refinement types (nat is defined as z:int{z >= 0})

val lkp : #a:Type -> #n:nat -> vector a n -> i:nat{0 < i /\ i <= n} -> a
let rec lkp #a #n xs i =
  match xs with
  | Cons x xs -> if i = n then x else lkp xs i

// First-class predicates (for which Type0 behaves like (classical) Prop)

type is_prefix_of (#a:Type) (#n:nat) (#m:nat{n <= m}) (xs:vector a n) (zs:vector a m) : Type0 =
  forall (i:nat) . (0 < i /\ i <= n) ==> lkp xs i == lkp zs i

type is_suffix_of (#a:Type) (#n:nat) (#m:nat{n <= m}) (ys:vector a n) (zs:vector a m) : Type0 =
  forall (i:nat) . (0 < i /\ i <= n) ==> lkp ys i == lkp zs (m - n + i)

// Extrinsic reasoning (using separate lemmas)

val lemma : #a:Type -> #n:nat -> #m:nat -> xs:vector a n -> ys:vector a m -> Lemma (requires (True))
  (ensures (xs `is_prefix_of` (append xs ys)))

let lemma #a #n #m xs ys =
  match xs with
  | Nil -> ()
  | Cons x xs -> admit () // need to call an appropriate sub-lemma here

// Intrinsic reasoning (making lemmas part of definitions)

val append' : #a:Type -> #n:nat -> #m:nat -> xs:vector a n -> ys:vector a m -> Pure (vector a (n + m)) (requires (True))
  (ensures (fun zs -> xs `is_prefix_of` zs
    /\ ys `is_suffix_of` zs))
```

F* – not just a pure programming language

- Tot, Lemma, Pure, ... are just some **effects** amongst many
 - Tot t
 - Lemma (requires $\text{pre}_{\text{Lemma}}$) (ensures $\text{post}_{\text{Lemma}}$)
 - Pure t (requires pre_{Pure}) (ensures $\text{post}_{\text{Pure}}$)
 - Div t (requires pre_{Div}) (ensures post_{Div})
 - Exc t (requires pre_{Exc}) (ensures post_{Exc})
 - ST t (requires pre_{ST}) (ensures post_{ST})
 - ...
- **Monad morphs.** $\text{Pure} \rightsquigarrow \{\text{Div}, \text{Exc}, \text{ST}\}; \text{Exc} \rightsquigarrow \text{STExc}; \dots$
- Systematically derived from **WP-calculi** (see POPL'17 paper)

Outline

- F^*
- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F^*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

Monotonicity in program verification

- Consider a program operating on **set-valued state**

`insert v; complex_procedure(); assert ($v \in \text{get}()$)`

- To prove the assertion (say, in a Floyd-Hoare style logic), we could prove that the code maintains a **stateful invariant**

$\{\lambda s. v \in s\}$ `complex_procedure()` $\{\lambda s. v \in s\}$

- likely that we have to **carry** $\lambda s. v \in s$ **through** the proof of `c_p`
- does not guarantee** that $\lambda s. v \in s$ holds at every point in `c_p`
- sensitive** to proving that `c_p` maintains $\lambda s. w \in s$ for some w
- However, if `c_p` **never removes**, then $\lambda s. v \in s$ is **stable**, and we would like the program logic to give us $v \in \text{get}()$ “for free”

Monotonicity in program verification

- Consider a program operating on **set-valued state**

insert v; complex_procedure(); **assert** ($v \in \text{get}()$)

- To prove the assertion (say, in a Floyd-Hoare style logic), we could prove that the code maintains a **stateful invariant**

$\{\lambda s. v \in s\}$ complex_procedure() $\{\lambda s. v \in s\}$

- likely that we have to carry $\lambda s. v \in s$ through the proof of c_p
- does not guarantee that $\lambda s. v \in s$ holds at every point in c_p
- sensitive to proving that c_p maintains $\lambda s. w \in s$ for some w
- However, if c_p never removes, then $\lambda s. v \in s$ is stable, and we would like the program logic to give us $v \in \text{get}()$ “for free”

Monotonicity in program verification

- Consider a program operating on **set-valued state**

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

- To prove the assertion (say, in a Floyd-Hoare style logic), we could prove that the code maintains a **stateful invariant**

$\{\lambda s. v \in s\}$ `complex_procedure()` $\{\lambda s. v \in s\}$

- likely that we have to **carry** $\lambda s. v \in s$ **through** the proof of `c_p`
- does not guarantee** that $\lambda s. v \in s$ holds at every point in `c_p`
- sensitive** to proving that `c_p` maintains $\lambda s. w \in s$ for some `w`
- However, if `c_p` **never removes**, then $\lambda s. v \in s$ is **stable**, and we would like the program logic to give us `v ∈ get()` “for free”

Monotonicity in program verification

- Consider a program operating on **set-valued state**

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

- To prove the assertion (say, in a Floyd-Hoare style logic), we could prove that the code maintains a **stateful invariant**

$\{\lambda s. v \in s\}$ `complex_procedure()` $\{\lambda s. v \in s\}$

- likely that we have to **carry** $\lambda s. v \in s$ **through** the proof of `c_p`
- does not guarantee** that $\lambda s. v \in s$ holds at every point in `c_p`
- sensitive** to proving that `c_p` maintains $\lambda s. w \in s$ for some `w`
- However, if `c_p` **never removes**, then $\lambda s. v \in s$ is **stable**, and we would like the program logic to give us `v ∈ get()` “for free”

Monotonicity in programming

- **Programming** also relies on **monotonicity**, even if you don't realise it!
- Consider ML-style typed references $r:\text{ref } a$
 - r is a **proof of existence** of an a -typed value in the heap
- Correctness relies on **monotonicity**!
 - 1) Allocation **stores** an a -typed value in the heap
 - 2) Writes **don't change type** and there is **no deallocation**
 - 3) So, given a ref. r , it is **guaranteed to point** to an a -typed value
- Baked into the memory models of most languages
- We derive them from **global state** + **general monotonicity**

Monotonicity in programming

- **Programming** also relies on **monotonicity**, even if you don't realise it!
- Consider ML-style typed **references** $r:\text{ref } a$
 - r is a **proof of existence** of an a -typed value in the heap
- Correctness relies on **monotonicity**!
 - 1) Allocation stores an a -typed value in the heap
 - 2) Writes **don't change type** and there is **no deallocation**
 - 3) So, given a ref. r , it is **guaranteed to point** to an a -typed value
- Baked into the memory models of most languages
- We derive them from **global state** + **general monotonicity**

Monotonicity in programming

- **Programming** also relies on **monotonicity**, even if you don't realise it!
- Consider ML-style typed **references** $r:\text{ref } a$
 - r is a **proof of existence** of an a -typed value in the heap
- Correctness relies on **monotonicity**!
 - 1) Allocation **stores** an a -typed value in the heap
 - 2) Writes **don't change type** and there is **no deallocation**
 - 3) So, given a ref. r , it is **guaranteed to point** to an a -typed value
- Baked into the memory models of most languages
- We derive them from **global state + general monotonicity**

Monotonicity in programming

- **Programming** also relies on **monotonicity**, even if you don't realise it!
- Consider ML-style typed **references** $r:\text{ref } a$
 - r is a **proof of existence** of an a -typed value in the heap
- Correctness relies on **monotonicity**!
 - 1) Allocation **stores** an a -typed value in the heap
 - 2) Writes **don't change type** and there is **no deallocation**
 - 3) So, given a ref. r , it is **guaranteed to point** to an a -typed value
- Baked into the memory models of most languages
- We derive them from **global state** + **general monotonicity**

Monotonicity is really useful!

- In this talk, we will see how monotonicity gives us
 - our **motivating example** and **monotonic counters**
 - **typed references** (`ref t`) and **untyped references** (`uref`)
 - more flexibility with **monotonic references** (`mref t rel`)
- See our POPL 2018 paper for more
 - temporarily **violating monotonicity** via snapshots
 - two substantial case studies in F^*
 - a **secure file-transfer** application
 - Ariadne **state continuity** protocol [Strackx, Piessens 2016]
 - pointers to other works in F^* relying on monotonicity for
 - sophisticated **region-based memory models** [fstar-lang.org]
 - **crypto** and **TLS verification** [project-everest.github.io]

Monotonicity is really useful!

- In this talk, we will see how monotonicity gives us
 - our **motivating example** and **monotonic counters**
 - **typed references** (`ref t`) and **untyped references** (`uref`)
 - more flexibility with **monotonic references** (`mref t rel`)
- See our POPL 2018 paper for more
 - temporarily **violating monotonicity** via snapshots
 - two substantial case studies in F^*
 - a **secure file-transfer** application
 - Ariadne **state continuity** protocol [Strackx, Piessens 2016]
 - pointers to other works in F^* relying on monotonicity for
 - sophisticated **region-based memory models** [fstar-lang.org]
 - **crypto** and **TLS verification** [project-everest.github.io]

Monotonicity is really useful!

- In this talk, we will see how monotonicity gives us
 - our **motivating example** and **monotonic counters**
 - **typed references** (`ref t`) and **untyped references** (`uref`)
 - more flexibility with **monotonic references** (`mref t rel`)
- See our POPL 2018 paper for more
 - temporarily **violating monotonicity** via snapshots
 - two substantial case studies in F^*
 - a **secure file-transfer** application
 - Ariadne **state continuity** protocol [Strackx, Piessens 2016]
 - pointers to other works in F^* relying on monotonicity for
 - sophisticated **region-based memory models** [fstar-lang.org]
 - **crypto** and **TLS verification** [project-everest.github.io]

Outline

- F^*
- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F^*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
 - per verification task, we **choose a preorder rel** on states
 - set inclusion, heap inclusion, increasing counter values, ...
 - a stateful program e is **monotonic** (wrt. rel) when
$$\forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s s'$$
 - a stateful predicate p is **stable** (wrt. rel) when
$$\forall s s'. p \ s \wedge \text{rel } s s' \implies p \ s'$$
- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - a means to **witness** the validity of $p \ s$ in some state s
 - a means for turning a p into a **state-independent proposition**
 - a means to **recall** the validity of $p \ s'$ in any future state s'
- Provides a **unifying account** of the existing *ad hoc* uses in F^*

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
 - per verification task, we **choose a preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counter values, ...
 - a stateful program e is **monotonic** (wrt. **rel**) when
$$\forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s \ s'$$
 - a stateful predicate p is **stable** (wrt. **rel**) when
$$\forall s s'. p \ s \wedge \text{rel } s \ s' \implies p \ s'$$
- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - a means to **witness** the validity of $p \ s$ in some state s
 - a means for turning a p into a **state-independent proposition**
 - a means to **recall** the validity of $p \ s'$ in any future state s'
- Provides a **unifying account** of the existing *ad hoc* uses in F^*

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
 - per verification task, we **choose a preorder rel** on states
 - set inclusion, heap inclusion, increasing counter values, ...
 - a stateful program e is **monotonic** (wrt. rel) when
$$\forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s s'$$
 - a stateful predicate p is **stable** (wrt. rel) when
$$\forall s s'. p \ s \wedge \text{rel } s s' \implies p \ s'$$
- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - a means to **witness** the validity of $p \ s$ in some state s
 - a means for turning a p into a **state-independent proposition**
 - a means to **recall** the validity of $p \ s'$ in any future state s'
- Provides a **unifying account** of the existing *ad hoc* uses in F^*

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
 - per verification task, we **choose a preorder rel** on states
 - set inclusion, heap inclusion, increasing counter values, ...

- a stateful program e is **monotonic** (wrt. rel) when

$$\forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s s'$$

- a stateful predicate p is **stable** (wrt. rel) when

$$\forall s s'. p s \wedge \text{rel } s s' \implies p s'$$

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - a means to **witness** the validity of $p s$ in some state s
 - a means for turning a p into a **state-independent proposition**
 - a means to **recall** the validity of $p s'$ in any future state s'
- Provides a **unifying account** of the existing *ad hoc* uses in F^*

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
 - per verification task, we **choose a preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counter values, ...

- a stateful program e is **monotonic** (wrt. **rel**) when

$$\forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s s'$$

- a stateful predicate p is **stable** (wrt. **rel**) when

$$\forall s s'. p \ s \wedge \text{rel } s s' \implies p \ s'$$

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - a means to **witness** the validity of $p \ s$ in some state s
 - a means for turning a p into a **state-independent proposition**
 - a means to **recall** the validity of $p \ s'$ in any future state s'
- Provides a **unifying account** of the existing *ad hoc* uses in F^*

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
 - per verification task, we **choose a preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counter values, ...
 - a stateful program e is **monotonic** (wrt. **rel**) when
$$\forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s s'$$
 - a stateful predicate p is **stable** (wrt. **rel**) when
$$\forall s s'. p \ s \wedge \text{rel } s s' \implies p \ s'$$
- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - a means to **witness** the validity of $p \ s$ in some state s
 - a means for turning a p into a **state-independent proposition**
 - a means to **recall** the validity of $p \ s'$ in any future state s'
- Provides a **unifying account** of the existing *ad hoc* uses in F^*

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
 - per verification task, we **choose a preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counter values, ...
 - a stateful program e is **monotonic** (wrt. **rel**) when
$$\forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s s'$$
 - a stateful predicate p is **stable** (wrt. **rel**) when
$$\forall s s'. p \ s \wedge \text{rel } s s' \implies p \ s'$$
- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - a means to **witness** the validity of $p \ s$ in some state s
 - a means for turning a p into a **state-independent proposition**
 - a means to **recall** the validity of $p \ s'$ in any future state s'
- Provides a **unifying account** of the existing *ad hoc* uses in F^*

Outline

- F^*
- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F^*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

Recap: Ordinary global state in F*

- F* supports Hoare-style reasoning about state via the **comp. type**

$$\text{ST}_{\text{state}}\ t\ (\text{requires}\ \text{pre})\ (\text{ensures}\ \text{post})$$

where

$$\text{pre} : \text{state} \rightarrow \text{Type} \qquad \text{post} : \text{state} \rightarrow t \rightarrow \text{state} \rightarrow \text{Type}$$

- ST is an abstract pre-postcondition refinement of

$$\text{st}\ t \stackrel{\text{def}}{=} \text{state} \rightarrow t * \text{state}$$

- The global state **actions** have types

$$\text{get} : \text{unit} \rightarrow \text{ST}\ \text{state}\ (\text{requires}\ (\lambda_.\top))\ (\text{ensures}\ (\lambda\ s_0\ s\ s_1.\ s_0 = s = s_1))$$
$$\text{put} : s:\text{state} \rightarrow \text{ST}\ \text{unit}\ (\text{requires}\ (\lambda_.\top))\ (\text{ensures}\ (\lambda\ __\ s_1.\ s_1 = s))$$

- Refs.** and **local state** are defined in F* using **monotonicity**

Recap: Ordinary global state in F*

- F* supports Hoare-style reasoning about state via the **comp. type**

$$ST_{\text{state}}\ t\ (\text{requires}\ \text{pre})\ (\text{ensures}\ \text{post})$$

where

$$\text{pre} : \text{state} \rightarrow \text{Type} \qquad \text{post} : \text{state} \rightarrow t \rightarrow \text{state} \rightarrow \text{Type}$$

- ST is an abstract pre-postcondition refinement of

$$\text{st}\ t \stackrel{\text{def}}{=} \text{state} \rightarrow t * \text{state}$$

- The global state **actions** have types

$$\text{get} : \text{unit} \rightarrow ST\ \text{state}\ (\text{requires}\ (\lambda _.\top))\ (\text{ensures}\ (\lambda\ s_0\ s\ s_1.\ s_0 = s = s_1))$$

$$\text{put} : s:\text{state} \rightarrow ST\ \text{unit}\ (\text{requires}\ (\lambda _.\top))\ (\text{ensures}\ (\lambda _ \ s_1.\ s_1 = s))$$

- Refs. and local state are defined in F* using **monotonicity**

Recap: Ordinary global state in F*

- F* supports Hoare-style reasoning about state via the **comp. type**

$$\text{ST}_{\text{state}} \, t \, (\text{requires } \text{pre}) \, (\text{ensures } \text{post})$$

where

$$\text{pre} : \text{state} \rightarrow \text{Type} \qquad \text{post} : \text{state} \rightarrow t \rightarrow \text{state} \rightarrow \text{Type}$$

- ST is an abstract pre-postcondition refinement of

$$\text{st } t \stackrel{\text{def}}{=} \text{state} \rightarrow t * \text{state}$$

- The global state **actions** have types

$$\text{get} : \text{unit} \rightarrow \text{ST } \text{state} \, (\text{requires } (\lambda _ . \top)) \, (\text{ensures } (\lambda s_0 \, s \, s_1 . s_0 = s = s_1))$$

$$\text{put} : s : \text{state} \rightarrow \text{ST } \text{unit} \, (\text{requires } (\lambda _ . \top)) \, (\text{ensures } (\lambda _ _ s_1 . s_1 = s))$$

- Refs. and local state are defined in F* using monotonicity

Recap: Ordinary global state in F*

- F* supports Hoare-style reasoning about state via the **comp. type**

$$ST_{\text{state}} \ t \ (\text{requires } \text{pre}) \ (\text{ensures } \text{post})$$

where

$$\text{pre} : \text{state} \rightarrow \text{Type} \qquad \text{post} : \text{state} \rightarrow t \rightarrow \text{state} \rightarrow \text{Type}$$

- ST is an abstract pre-postcondition refinement of

$$st \ t \stackrel{\text{def}}{=} \text{state} \rightarrow t * \text{state}$$

- The global state **actions** have types

$$\text{get} : \text{unit} \rightarrow ST \ \text{state} \ (\text{requires } (\lambda _ . \top)) \ (\text{ensures } (\lambda s_0 \ s \ s_1 . s_0 = s = s_1))$$

$$\text{put} : s : \text{state} \rightarrow ST \ \text{unit} \ (\text{requires } (\lambda _ . \top)) \ (\text{ensures } (\lambda _ _ s_1 . s_1 = s))$$

- Refs.** and **local state** are defined in F* using **monotonicity**

New: Monotonic global state in F*

- We capture monotonic state with a new **computational type**

$\text{MST}_{\text{state}, \text{rel}}\ t\ (\text{requires}\ \text{pre})\ (\text{ensures}\ \text{post})$

- The **get** action is typed as in ST

$\text{get} : \text{unit} \rightarrow \text{MST}\ \text{state}\ (\text{requires}\ (\lambda\ _.\text{T}))$
 $\quad\quad\quad (\text{ensures}\ (\lambda\ s_0\ s\ s_1.\ s_0 = s = s_1))$

- To ensure **monotonicity**, the **put** action gets a precondition

$\text{put} : \text{s}:\text{state} \rightarrow \text{MST}\ \text{unit}\ (\text{requires}\ (\lambda\ s_0.\ \text{rel}\ s_0\ s))$
 $\quad\quad\quad (\text{ensures}\ (\lambda\ _.\ s_1.\ s_1 = s))$

- So intuitively, MST is an **abstract** pre-postcondition refinement of

$\text{mst}\ t \stackrel{\text{def}}{=} \text{s}_0:\text{state} \rightarrow t * \text{s}_1:\text{state}\{\text{rel}\ \text{s}_0\ \text{s}_1\}$

New: Monotonic global state in F*

- We capture monotonic state with a new **computational type**

$\text{MST}_{\text{state}, \text{rel}} \ t \ (\text{requires} \ \text{pre}) \ (\text{ensures} \ \text{post})$

- The **get** action is typed as in ST

$\text{get} : \text{unit} \rightarrow \text{MST state} \ (\text{requires} \ (\lambda _ . \top))$
 $(\text{ensures} \ (\lambda s_0 \ s \ s_1 . s_0 = s = s_1))$

- To ensure **monotonicity**, the **put** action gets a precondition

$\text{put} : s : \text{state} \rightarrow \text{MST unit} \ (\text{requires} \ (\lambda s_0 . \text{rel} \ s_0 \ s))$
 $(\text{ensures} \ (\lambda _ \ s_1 . s_1 = s))$

- So intuitively, MST is an **abstract** pre-postcondition refinement of

$\text{mst } t \stackrel{\text{def}}{=} s_0 : \text{state} \rightarrow t * s_1 : \text{state} \{ \text{rel } s_0 \ s_1 \}$

New: Monotonic global state in F*

- We capture monotonic state with a new **computational type**

$\text{MST}_{\text{state}, \text{rel}} \ t \ (\text{requires} \ \text{pre}) \ (\text{ensures} \ \text{post})$

- The **get** action is typed as in **ST**

$\text{get} : \text{unit} \rightarrow \text{MST} \ \text{state} \ (\text{requires} \ (\lambda _ . \top))$
 $(\text{ensures} \ (\lambda s_0 \ s \ s_1 . s_0 = s = s_1))$

- To ensure **monotonicity**, the **put** action gets a precondition

$\text{put} : s : \text{state} \rightarrow \text{MST} \ \text{unit} \ (\text{requires} \ (\lambda s_0 . \text{rel} \ s_0 \ s))$
 $(\text{ensures} \ (\lambda _ \ s_1 . s_1 = s))$

- So intuitively, **MST** is an **abstract** pre-postcondition refinement of

$\text{mst} \ t \stackrel{\text{def}}{=} s_0 : \text{state} \rightarrow t * s_1 : \text{state} \{ \text{rel} \ s_0 \ s_1 \}$

New: Monotonic global state in F*

- We capture monotonic state with a new **computational type**

$\text{MST}_{\text{state}, \text{rel}} \tau \text{ (requires pre) (ensures post)}$

- The **get** action is typed as in **ST**

$\text{get} : \text{unit} \rightarrow \text{MST state (requires } (\lambda _ . \top) \text{)}$
 $\text{(ensures } (\lambda s_0 s s_1 . s_0 = s = s_1) \text{))}$

- To ensure **monotonicity**, the **put** action gets a precondition

$\text{put} : \text{s:state} \rightarrow \text{MST unit (requires } (\lambda s_0 . \text{rel } s_0 \text{ s}) \text{)}$
 $\text{(ensures } (\lambda _ s_1 . s_1 = s) \text{)}$

- So intuitively, MST is an **abstract** pre-postcondition refinement of

$\text{mst } \tau \stackrel{\text{def}}{=} \text{s}_0:\text{state} \rightarrow \tau * \text{s}_1:\text{state} \{ \text{rel } s_0 \text{ s}_1 \}$

New: Monotonic global state in F*

- We capture monotonic state with a new **computational type**

$$\text{MST}_{\text{state}, \text{rel}} \, t \, (\text{requires} \, \text{pre}) \, (\text{ensures} \, \text{post})$$

- The **get** action is typed as in **ST**

$$\begin{aligned} \text{get} : \text{unit} \rightarrow \text{MST} \, \text{state} \, (\text{requires} \, (\lambda _ . \top)) \\ (\text{ensures} \, (\lambda s_0 \, s \, s_1 . s_0 = s = s_1)) \end{aligned}$$

- To ensure **monotonicity**, the **put** action gets a precondition

$$\begin{aligned} \text{put} : \text{s} : \text{state} \rightarrow \text{MST} \, \text{unit} \, (\text{requires} \, (\lambda s_0 . \text{rel} \, s_0 \, s)) \\ (\text{ensures} \, (\lambda _ _ s_1 . s_1 = s)) \end{aligned}$$

- So intuitively, **MST** is an **abstract** pre-postcondition refinement of

$$\text{mst} \, t \stackrel{\text{def}}{=} \text{s}_0 : \text{state} \rightarrow t * \text{s}_1 : \text{state} \{ \text{rel} \, \text{s}_0 \, \text{s}_1 \}$$

New: Recalling a Witness

- We extend F^* with a **logical capability**

$\text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type}$

together with a **weakening principle (functoriality)**

$\text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma} (\text{requires } (\forall s. p\ s \implies q\ s))$
 $(\text{ensures } (\text{witnessed } p \implies \text{witnessed } q))$

- Intuitively, think of it as a **necessity modality**

$$\llbracket \text{witnessed } p \rrbracket(s) \stackrel{\text{def}}{=} \forall s'. \text{rel } s\ s' \implies \llbracket p\ s' \rrbracket(s)$$

- As usual, for natural deduction, need **world-indexed sequents**
- But, wait a minute ...

New: Recalling a Witness

- We extend F^* with a **logical capability**

$\text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type}$

together with a **weakening principle (functoriality)**

$\text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma } (\text{requires } (\forall s. p\ s \implies q\ s))$
 $(\text{ensures } (\text{witnessed } p \implies \text{witnessed } q))$

- Intuitively, think of it as a **necessity modality**

$$\llbracket \text{witnessed } p \rrbracket (s) \stackrel{\text{def}}{=} \forall s'. \text{rel } s\ s' \implies \llbracket p\ s' \rrbracket (s)$$

- As usual, for natural deduction, need **world-indexed sequents**
- But, wait a minute ...

New: Recalling a Witness

- We extend F^* with a **logical capability**

$\text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type}$

together with a **weakening principle (functoriality)**

$\text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma } (\text{requires } (\forall s. p\ s \implies q\ s))$
 $(\text{ensures } (\text{witnessed } p \implies \text{witnessed } q))$

- Intuitively, think of it as a **necessity modality**

$$\llbracket \text{witnessed } p \rrbracket (s) \stackrel{\text{def}}{=} \forall s'. \text{rel } s\ s' \implies \llbracket p\ s' \rrbracket (s)$$

- As usual, for natural deduction, need **world-indexed sequents**
- But, wait a minute ...

New: Recalling a Witness

- We extend F^* with a **logical capability**

$\text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type}$

together with a **weakening principle (functoriality)**

$\text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma} \left(\text{requires } (\forall s. p\ s \implies q\ s) \right)$
 $\left(\text{ensures } (\text{witnessed } p \implies \text{witnessed } q) \right)$

- Intuitively, think of it as a **necessity modality**

$$\llbracket \text{witnessed } p \rrbracket(s) \stackrel{\text{def}}{=} \forall s'. \text{rel } s\ s' \implies \llbracket p\ s' \rrbracket(s)$$

- As usual, for natural deduction, need **world-indexed sequents**
- But, wait a minute ...

New: Recalling a Witness

- We extend F^* with a **logical capability**

$\text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type}$

together with a **weakening principle (functoriality)**

$\text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma} (\text{requires } (\forall s. p\ s \implies q\ s))$
 $(\text{ensures } (\text{witnessed } p \implies \text{witnessed } q))$

- Intuitively, think of it as a **necessity modality**

$$\llbracket \text{witnessed } p \rrbracket (s) \stackrel{\text{def}}{=} \forall s'. \text{rel } s\ s' \implies \llbracket p\ s' \rrbracket (s)$$

- As usual, for natural deduction, need **world-indexed sequents**
- But, wait a minute ...

New: Recalling a Witness

- ... Hoare-style logics are essentially **world/state-indexed**, so
- we include a **stateful introduction rule** for witnessed

```
witness : p:(state  $\rightarrow$  Type0)  
          $\rightarrow$  MST unit (requires ( $\lambda s_0. p$  'stable_from'  $s_0$ ))  
                   (ensures ( $\lambda s_0 - s_1. s_0 = s_1 \wedge$  witnessed  $p$ )))
```

- and a **stateful elimination rule** for witnessed

```
recall : p:(state  $\rightarrow$  Type0)  
         $\rightarrow$  MST unit (requires ( $\lambda \_.$  witnessed  $p$ ))  
                  (ensures ( $\lambda s_0 - s_1. s_0 = s_1 \wedge p$  'stable_from'  $s_1$ )))
```

New: Recalling a Witness

- ... Hoare-style logics are essentially **world/state-indexed**, so
- we include a **stateful introduction rule** for witnessed

```
witness : p:(state → Type0)  
        → MST unit (requires (λ s0. p 'stable_from' s0))  
                   (ensures (λ s0 s1. s0 = s1 ∧ witnessed p))
```

- and a **stateful elimination rule** for witnessed

```
recall : p:(state → Type0)  
        → MST unit (requires (λ _. witnessed p))  
                   (ensures (λ s0 s1. s0 = s1 ∧ p 'stable_from' s1))
```

New: Recalling a Witness

- ... Hoare-style logics are essentially **world/state-indexed**, so
- we include a **stateful introduction rule** for witnessed

```
witness : p:(state  $\rightarrow$  Type0)  
          $\rightarrow$  MST unit (requires ( $\lambda s_0$ . p 'stable_from' s0))  
                   (ensures ( $\lambda s_0 - s_1$ . s0 = s1  $\wedge$  witnessed p)))
```

- and a **stateful elimination rule** for witnessed

```
recall : p:(state  $\rightarrow$  Type0)  
         $\rightarrow$  MST unit (requires ( $\lambda \_$ . witnessed p))  
                  (ensures ( $\lambda s_0 - s_1$ . s0 = s1  $\wedge$  p 'stable_from' s1)))
```

Outline

- F^*
- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F^*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

The motivating example revisited

- Recall the program operating on the **set-valued state**

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

- We pick **set inclusion** \subseteq as our preorder rel on states
- We **prove the assertion** by inserting a witness and recall

```
insert v; witness ( $\lambda s. v \in s$ ); c_p(); recall ( $\lambda s. v \in s$ ); assert (v ∈ get())
```

- For any other w, wrapping

```
insert w; [ ]; assert (w ∈ get())
```

around the program is handled **similarly easily** by

```
insert w; witness ( $\lambda s. w \in s$ ); [ ]; recall ( $\lambda s. w \in s$ ); assert (w ∈ get())
```

- Monotonic counters** are analogous, by picking \mathbb{N} and \leq , e.g.,

```
create 0; incr(); witness ( $\lambda c. c > 0$ ); c_p(); recall ( $\lambda c. c > 0$ )
```

The motivating example revisited

- Recall the program operating on the **set-valued state**

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

- We pick **set inclusion** \subseteq as our preorder rel on states
- We **prove the assertion** by inserting a witness and recall

```
insert v; witness ( $\lambda s. v \in s$ ); c_p(); recall ( $\lambda s. v \in s$ ); assert (v ∈ get())
```

- For any other w, wrapping

```
insert w; [ ]; assert (w ∈ get())
```

around the program is handled **similarly easily** by

```
insert w; witness ( $\lambda s. w \in s$ ); [ ]; recall ( $\lambda s. w \in s$ ); assert (w ∈ get())
```

- Monotonic counters** are analogous, by picking \mathbb{N} and \leq , e.g.,

```
create 0; incr(); witness ( $\lambda c. c > 0$ ); c_p(); recall ( $\lambda c. c > 0$ )
```


The motivating example revisited

- Recall the program operating on the **set-valued state**

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

- We pick **set inclusion** \subseteq as our preorder rel on states
- We **prove the assertion** by inserting a witness and recall

```
insert v; witness ( $\lambda s. v \in s$ ); c_p(); recall ( $\lambda s. v \in s$ ); assert (v ∈ get())
```

- For any other w, wrapping

```
insert w; [ ]; assert (w ∈ get())
```

around the program is handled **similarly easily** by

```
insert w; witness ( $\lambda s. w \in s$ ); [ ]; recall ( $\lambda s. w \in s$ ); assert (w ∈ get())
```

- Monotonic counters** are analogous, by picking \mathbb{N} and \leq , e.g.,

```
create 0; incr(); witness ( $\lambda c. c > 0$ ); c_p(); recall ( $\lambda c. c > 0$ )
```

The motivating example revisited

- Recall the program operating on the **set-valued state**

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

- We pick **set inclusion** \subseteq as our preorder rel on states
- We **prove the assertion** by inserting a witness and recall

```
insert v; witness ( $\lambda s. v \in s$ ); c_p(); recall ( $\lambda s. v \in s$ ); assert (v ∈ get())
```

- For **any other** w , wrapping

```
insert w; [ ]; assert (w ∈ get())
```

around the program is handled **similarly easily** by

```
insert w; witness ( $\lambda s. w \in s$ ); [ ]; recall ( $\lambda s. w \in s$ ); assert (w ∈ get())
```

- Monotonic counters are analogous, by picking \mathbb{N} and \leq , e.g.,

```
create 0; incr(); witness ( $\lambda c. c > 0$ ); c_p(); recall ( $\lambda c. c > 0$ )
```

The motivating example revisited

- Recall the program operating on the **set-valued state**

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

- We pick **set inclusion** \subseteq as our preorder rel on states
- We **prove the assertion** by inserting a witness and recall

```
insert v; witness ( $\lambda s. v \in s$ ); c_p(); recall ( $\lambda s. v \in s$ ); assert (v ∈ get())
```

- For **any other** w , wrapping

```
insert w; [ ]; assert (w ∈ get())
```

around the program is handled **similarly easily** by

```
insert w; witness ( $\lambda s. w \in s$ ); [ ]; recall ( $\lambda s. w \in s$ ); assert (w ∈ get())
```

- Monotonic counters** are analogous, by picking \mathbb{N} and \leq , e.g.,

```
create 0; incr(); witness ( $\lambda c. c > 0$ ); c_p(); recall ( $\lambda c. c > 0$ )
```

ML-style typed references (local state)

- First, we define a type of **heaps** as a finite map

```
type heap =
```

```
| H : h : (N → cell) → ctr : N { ∀ n. ctr ≤ n ⇒ h n = Unused } → heap
```

where

```
type cell =
```

```
| Unused : cell
```

```
| Used : a : Type → v : a → cell
```

- Next, we define a **preorder** on heaps (**heap inclusion**)

```
let heap_inclusion (H h0 _) (H h1 _) = ∀ id. match h0 id, h1 id with
```

```
| Used a _, Used b _ → a = b
```

```
| Unused, Used _ _ → ⊤
```

```
| Unused, Unused → ⊤
```

```
| Used _ _, Unused → ⊥
```

ML-style typed references (local state)

- First, we define a type of **heaps** as a finite map

```
type heap =
```

```
| H : h:( $\mathbb{N} \rightarrow \text{cell}$ )  $\rightarrow$  ctr: $\mathbb{N}\{\forall n. \text{ctr} \leq n \implies h\ n = \text{Unused}\} \rightarrow \text{heap}$ 
```

where

```
type cell =
```

```
| Unused : cell
```

```
| Used : a:Type  $\rightarrow$  v:a  $\rightarrow$  cell
```

- Next, we define a preorder on heaps (**heap inclusion**)

```
let heap_inclusion (H h0 _) (H h1 _) =  $\forall \text{id}. \text{match } h_0\ \text{id}, h_1\ \text{id} \text{ with}$ 
```

```
| Used a _, Used b _  $\rightarrow$  a = b
```

```
| Unused, Used _ _  $\rightarrow$   $\top$ 
```

```
| Unused, Unused  $\rightarrow$   $\top$ 
```

```
| Used _ _, Unused  $\rightarrow$   $\perp$ 
```

ML-style typed references (local state)

- First, we define a type of **heaps** as a finite map

`type heap =`

`| H : h:($\mathbb{N} \rightarrow \text{cell}$) \rightarrow ctr: $\mathbb{N}\{\forall n. \text{ctr} \leq n \implies h\ n = \text{Unused}\}$ \rightarrow heap`

where

`type cell =`

`| Unused : cell`

`| Used : a:Type \rightarrow v:a \rightarrow cell`

- Next, we define a **preorder** on heaps (**heap inclusion**)

`let heap_inclusion (H h0 _) (H h1 _) = $\forall \text{id}. \text{match } h_0\ \text{id}, h_1\ \text{id} \text{ with}$`

`| Used a _, Used b _ $\rightarrow a = b$`

`| Unused, Used _ _ $\rightarrow \top$`

`| Unused, Unused $\rightarrow \top$`

`| Used _ _, Unused $\rightarrow \perp$`

ML-style typed references (local state)

- As a result, we can define new **local state effect**

$$\text{MLST } t \text{ pre post} \stackrel{\text{def}}{=} \text{MST}_{\text{heap, heap_inclusion}} t \text{ pre post}$$

- Next, we define the type of **references** using monotonicity

abstract type ref a = id:N{witnessed (λh . contains h id a)}

where

let contains (H h _) id a =

match h id with

| Used b _ \rightarrow a = b

| Unused $\rightarrow \perp$

- Important: contains is **stable** wrt. heap_inclusion

ML-style typed references (local state)

- As a result, we can define new **local state effect**

$$\text{MLST } t \text{ pre post} \stackrel{\text{def}}{=} \text{MST}_{\text{heap, heap_inclusion}} t \text{ pre post}$$

- Next, we define the type of **references** using monotonicity

```
abstract type ref a = id:N{witnessed ( $\lambda h$ . contains h id a)}
```

where

```
let contains (H h _) id a =
```

```
  match h id with
```

```
    | Used b _  $\rightarrow$  a = b
```

```
    | Unused  $\rightarrow \perp$ 
```

- Important: contains is **stable** wrt. heap_inclusion

ML-style typed references (local state)

- As a result, we can define new **local state effect**

$$\text{MLST } t \text{ pre post} \stackrel{\text{def}}{=} \text{MST}_{\text{heap, heap_inclusion}} t \text{ pre post}$$

- Next, we define the type of **references** using monotonicity

```
abstract type ref a = id:N{witnessed ( $\lambda h$ . contains h id a)}
```

where

```
let contains (H h _) id a =  
  match h id with  
  | Used b _  $\rightarrow$  a = b  
  | Unused  $\rightarrow \perp$ 
```

- Important: contains is **stable** wrt. heap_inclusion

ML-style typed references (local state)

- Finally, we define **MLST's actions** using **MST's actions**

- `let alloc (a:Type) (v:a) : MLST (ref a) ... = ...`
 - get the current heap
 - create a fresh ref., and add it to the heap
 - put the updated heap back
 - witness that the created ref. is in the heap
- `let read (r:ref a) : MLST a (req. (\top)) (ens. (...)) = ...`
 - recall that the given ref. is in the heap
 - get the current heap
 - select the given reference from the heap
- `let write (r:ref a) (v:a) : MLST unit ... = ...`
 - recall that the given ref. is in the heap
 - get the current heap
 - update the heap with the given value at the given ref.
 - put the updated heap back

ML-style typed references (local state)

- Finally, we define **MLST**'s **actions** using **MST**'s actions
 - **let alloc** ($a:\text{Type}$) ($v:a$) : **MLST** ($\text{ref } a$) ... = ...
 - **get** the current heap
 - **create** a fresh ref., and **add** it to the heap
 - **put** the updated heap back
 - **witness** that the created ref. is in the heap
 - **let read** ($r:\text{ref } a$) : **MLST** a ($\text{req. } (\top)$) ($\text{ens. } (...)$) = ...
 - **recall** that the given ref. is in the heap
 - **get** the current heap
 - **select** the given reference from the heap
 - **let write** ($r:\text{ref } a$) ($v:a$) : **MLST** unit ... = ...
 - **recall** that the given ref. is in the heap
 - **get** the current heap
 - **update** the heap with the given value at the given ref.
 - **put** the updated heap back

ML-style typed references (local state)

- Finally, we define **MLST**'s **actions** using **MST**'s actions
 - **let alloc** $(a:\text{Type}) (v:a) : \text{MLST} (\text{ref } a) \dots = \dots$
 - **get** the current heap
 - **create** a fresh ref., and **add** it to the heap
 - **put** the updated heap back
 - **witness** that the created ref. is in the heap
 - **let read** $(r:\text{ref } a) : \text{MLST } a (\text{req. } (\top)) (\text{ens. } (\dots)) = \dots$
 - **recall** that the given ref. is in the heap
 - **get** the current heap
 - **select** the given reference from the heap
 - **let write** $(r:\text{ref } a) (v:a) : \text{MLST unit } \dots = \dots$
 - **recall** that the given ref. is in the heap
 - **get** the current heap
 - **update** the heap with the given value at the given ref.
 - **put** the updated heap back

ML-style typed references (local state)

- Finally, we define **MLST**'s **actions** using **MST**'s actions
 - **let alloc** $(a:\text{Type}) (v:a) : \text{MLST} (\text{ref } a) \dots = \dots$
 - **get** the current heap
 - **create** a fresh ref., and **add** it to the heap
 - **put** the updated heap back
 - **witness** that the created ref. is in the heap
 - **let read** $(r:\text{ref } a) : \text{MLST } a (\text{req. } (\top)) (\text{ens. } (\dots)) = \dots$
 - **recall** that the given ref. is in the heap
 - **get** the current heap
 - **select** the given reference from the heap
 - **let write** $(r:\text{ref } a) (v:a) : \text{MLST} \text{ unit } \dots = \dots$
 - **recall** that the given ref. is in the heap
 - **get** the current heap
 - **update** the heap with the given value at the given ref.
 - **put** the updated heap back

Adding untyped and monotonic references

- Untyped references (`uref`) with strong updates

- Used heap cells are extended with **tags**

where
$$| \text{Used} : a:\text{Type} \rightarrow v:a \rightarrow t:\text{tag} \rightarrow \text{cell}$$

$$\text{type tag} = \text{Typed} : \text{tag} \mid \text{Untyped} : \text{tag}$$

- actions corresponding to urefs have **weaker types** than for refs

- Monotonic references (`mref a rel`)

- Used heap cells are extended with **typed tags**

where
$$| \text{Used} : a:\text{Type} \rightarrow v:a \rightarrow t:\text{tag } a \rightarrow \text{cell}$$

$$\text{type tag } a = \text{Typed} : \text{rel}:\text{preorder } a \rightarrow \text{tag } a \mid \text{Untyped} : \text{tag } a$$

- `mrefs` provide **more flexibility** with ref.-wise monotonicity
- Further, all three can be extended with **manually managed** refs.

Adding untyped and monotonic references

- **Untyped references** (`uref`) with strong updates

- Used heap cells are extended with **tags**

| $\text{Used} : a:\text{Type} \rightarrow v:a \rightarrow \text{t:tag} \rightarrow \text{cell}$

where

`type` tag = Typed : tag | Untyped : tag

- actions corresponding to urefs have **weaker types** than for refs

- **Monotonic references** (`mref a rel`)

- Used heap cells are extended with **typed tags**

| $\text{Used} : a:\text{Type} \rightarrow v:a \rightarrow \text{t:tag } a \rightarrow \text{cell}$

where

`type` tag a = Typed : `rel:preorder a` → tag a | Untyped : tag a

- `mrefs` provide **more flexibility** with ref.-wise monotonicity

- Further, all three can be extended with **manually managed** refs.

Adding untyped and monotonic references

- **Untyped references** (`uref`) with strong updates

- Used heap cells are extended with **tags**

where
$$| \text{Used} : a:\text{Type} \rightarrow v:a \rightarrow \text{t:tag} \rightarrow \text{cell}$$

$$\text{type tag} = \text{Typed} : \text{tag} \mid \text{Untyped} : \text{tag}$$

- actions corresponding to urefs have **weaker types** than for refs

- **Monotonic references** (`mref a rel`)

- Used heap cells are extended with **typed tags**

where
$$| \text{Used} : a:\text{Type} \rightarrow v:a \rightarrow \text{t:tag } a \rightarrow \text{cell}$$

$$\text{type tag } a = \text{Typed} : \text{rel:preorder } a \rightarrow \text{tag } a \mid \text{Untyped} : \text{tag } a$$

- mrefs provide **more flexibility** with ref.-wise monotonicity

- Further, all three can be extended with **manually managed** refs.

Adding untyped and monotonic references

- **Untyped references** (`uref`) with strong updates

- Used heap cells are extended with **tags**

where
$$| \text{Used} : a:\text{Type} \rightarrow v:a \rightarrow \text{t:tag} \rightarrow \text{cell}$$

$$\text{type tag} = \text{Typed} : \text{tag} \mid \text{Untyped} : \text{tag}$$

- actions corresponding to urefs have **weaker types** than for refs

- **Monotonic references** (`mref a rel`)

- Used heap cells are extended with **typed tags**

where
$$| \text{Used} : a:\text{Type} \rightarrow v:a \rightarrow \text{t:tag } a \rightarrow \text{cell}$$

$$\text{type tag } a = \text{Typed} : \text{rel:preorder } a \rightarrow \text{tag } a \mid \text{Untyped} : \text{tag } a$$

- `mrefs` provide **more flexibility** with ref.-wise monotonicity
- Further, all three can be extended with **manually managed** refs.

Conclusion

- Monotonicity
 - can be distilled into a **simple** and **general** framework
 - is **useful** for **programming** (refs.) and **verification** (Prj. Everest)
- See our POPL 2018 paper for
 - further **examples** and **case studies**
 - **meta-theory** and **total correctness** for MST
 - based on an instrumented operational semantics
$$(\text{witness } x.\varphi, s, W) \rightsquigarrow (\text{return } (), s, W \cup \{x.\varphi\})$$
 - and cut elimination for the witnessed-logic
 - first steps towards **monadic reification** for MST
 - useful for extrinsic reasoning, e.g., for relational properties
 - but have to be careful when breaking abstraction

Conclusion

- Monotonicity
 - can be distilled into a **simple** and **general** framework
 - is **useful** for **programming** (refs.) and **verification** (Prj. Everest)
- See our POPL 2018 paper for
 - further **examples** and **case studies**
 - **meta-theory** and **total correctness** for **MST**
 - based on an instrumented operational semantics
$$(\text{witness } x.\varphi, s, W) \rightsquigarrow (\text{return } (), s, W \cup \{x.\varphi\})$$
 - and cut elimination for the witnessed-logic
 - first steps towards **monadic reification** for **MST**
 - useful for extrinsic reasoning, e.g., for relational properties
 - but have to be careful when breaking abstraction

Thank you for your attention!

Questions?

Appendix: Mon. reification and reflection

- In F^* every **abstract ST computation**

$$e : \text{ST } t \text{ (requires pre) (ensures post)}$$

can be **reified** into its **underlying Pure representation**

$$\text{reify } e : s_0 : \text{state} \rightarrow \text{Pure } (t * \text{state}) \text{ (requires (pre } s_0)) \\ \text{(ensures } (\lambda (x, s_1). \text{post } s_0 \ x \ s_1))$$

and vice versa using **reflection** (see our POPL 2017 paper)

- Useful for **extrinsic reasoning**, e.g., for relational properties
- We also need it for **MST**!

Appendix: Mon. reification and reflection

- In F^* every **abstract ST computation**

$$e : \text{ST } t \text{ (requires pre) (ensures post)}$$

can be **reified** into its **underlying Pure representation**

$$\text{reify } e : s_0 : \text{state} \rightarrow \text{Pure } (t * \text{state}) \text{ (requires (pre } s_0)) \\ \text{(ensures } (\lambda (x, s_1). \text{post } s_0 \ x \ s_1))$$

and vice versa using **reflection** (see our POPL 2017 paper)

- Useful for **extrinsic reasoning**, e.g., for relational properties
- We also need it for MST!

Appendix: Mon. reification and reflection

- In F^* every **abstract ST computation**

$$e : \text{ST } t \text{ (requires pre) (ensures post)}$$

can be **reified** into its **underlying Pure representation**

$$\text{reify } e : s_0 : \text{state} \rightarrow \text{Pure } (t * \text{state}) \text{ (requires (pre } s_0)) \\ \text{(ensures } (\lambda (x, s_1). \text{post } s_0 \ x \ s_1))$$

and vice versa using **reflection** (see our POPL 2017 paper)

- Useful for **extrinsic reasoning**, e.g., for relational properties
- We also need it for **MST**!

Appendix: Mon. reification and reflection

- We cannot simply turn an **abstract MST computation**

$$e : \text{MST } t \text{ (requires pre) (ensures post)}$$

into a **state-passing function**

$$s_0 : \text{state} \rightarrow \text{Pure } (t * s_1 : \text{state} \{ \text{rel } s_0 \ s_1 \}) \text{ (req. (pre } s_0))$$
$$\text{(ens. } (\lambda (x, s_1). \text{post } s_0 \ x \ s_1))$$

- For example, consider the **recalling** action

$$\text{recall} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit (requires } (\lambda _. \text{witnessed } p))$$
$$\text{(ensures } (\lambda s_0 \ s_1. s_0 = s_1 \wedge p \ s_1))$$

which we would like to **reduce** as

$$\text{reify (recall } p) \rightsquigarrow \lambda s_0. \text{return } ((), s_0)$$

but we cannot prove $p \ s_0$ from **witnessed** p in the pure logic

Appendix: Mon. reification and reflection

- We cannot simply turn an **abstract MST computation**

$$e : \text{MST } t \text{ (requires pre) (ensures post)}$$

into a **state-passing function**

$$s_0 : \text{state} \rightarrow \text{Pure } (t * s_1 : \text{state} \{ \text{rel } s_0 \ s_1 \}) \text{ (req. (pre } s_0))$$
$$(\text{ens. } (\lambda (x, s_1). \text{post } s_0 \ x \ s_1))$$

- For example, consider the **recalling** action

$$\text{recall} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit (requires } (\lambda _. \text{witnessed } p))$$
$$(\text{ensures } (\lambda s_0 \ s_1. s_0 = s_1 \wedge p \ s_1))$$

which we would like to **reduce** as

$$\text{reify (recall } p) \rightsquigarrow \lambda s_0. \text{return } ((), s_0)$$

but we cannot prove $p \ s_0$ from $\text{witnessed } p$ in the pure logic

Appendix: Mon. reification and reflection

- We cannot simply turn an **abstract MST computation**

$$e : \text{MST } t \text{ (requires pre) (ensures post)}$$

into a **state-passing function**

$$s_0 : \text{state} \rightarrow \text{Pure } (t * s_1 : \text{state} \{ \text{rel } s_0 \ s_1 \}) \text{ (req. (pre } s_0)) \\ \text{(ens. } (\lambda (x, s_1). \text{post } s_0 \ x \ s_1))$$

- For example, consider the **recalling** action

$$\text{recall} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit (requires } (\lambda _ . \text{witnessed } p)) \\ \text{(ensures } (\lambda s_0 \ s_1 . s_0 = s_1 \wedge p \ s_1))$$

which we would like to **reduce** as

$$\text{reify (recall } p) \rightsquigarrow \lambda s_0 . \text{return } ((), s_0)$$

but we cannot prove $p \ s_0$ from $\text{witnessed } p$ in the pure logic

Appendix: Mon. reification and reflection

- In our POPL 2018 paper, we support reification and reflection by
 - indexing $\text{MST}_{\text{state}, \text{rel}, \mathbf{b}}$ with a **boolean flag** \mathbf{b} (reifiable?), and
 - **guarding** the pre-postconditions of witness and recall with \mathbf{b}so if $\mathbf{b} = \text{true}$ then witness and recall are **logically no-ops**.
- This **works** but leads to **duplication** of pre- and postconditions!
- Instead, ongoing work is taking (hybrid) **modal logic** seriously

$$\mathbf{s}_0:\text{state} \rightarrow \text{Pure } (t * \mathbf{s}_1:\text{state}\{\text{rel } \mathbf{s}_0 \ \mathbf{s}_1\}) \left(\text{req. } (\text{pre } \mathbf{s}_0 \ @ \ \mathbf{s}_0) \right) \\ \left(\text{ens. } (\lambda (x, \mathbf{s}_1). \text{post } \mathbf{s}_0 \ x \ \mathbf{s}_1 \ @ \ \mathbf{s}_1) \right)$$

where $@$ is the **standard translation** of modal logic

Appendix: Mon. reification and reflection

- In our POPL 2018 paper, we support reification and reflection by
 - indexing $\text{MST}_{\text{state}, \text{rel}, \mathbf{b}}$ with a **boolean flag** \mathbf{b} (reifiable?), and
 - **guarding** the pre-postconditions of witness and recall with \mathbf{b}so if $\mathbf{b} = \text{true}$ then witness and recall are **logically no-ops**.
- This **works** but leads to **duplication** of pre- and postconditions!

- Instead, ongoing work is taking (hybrid) **modal logic** seriously

$$s_0:\text{state} \rightarrow \text{Pure } (t * s_1:\text{state}\{\text{rel } s_0 \ s_1\}) \ (\text{req. } (\text{pre } s_0 \ @ \ s_0)) \\ (\text{ens. } (\lambda (x, s_1). \text{post } s_0 \ x \ s_1 \ @ \ s_1))$$

where $@$ is the **standard translation** of modal logic

Appendix: Mon. reification and reflection

- In our POPL 2018 paper, we support reification and reflection by
 - indexing $\text{MST}_{\text{state}, \text{rel}, \mathbf{b}}$ with a **boolean flag** \mathbf{b} (reifiable?), and
 - **guarding** the pre-postconditions of witness and recall with \mathbf{b}so if $\mathbf{b} = \text{true}$ then witness and recall are **logically no-ops**.
- This **works** but leads to **duplication** of pre- and postconditions!
- Instead, ongoing work is taking (hybrid) **modal logic** seriously

$$\mathbf{s}_0:\text{state} \rightarrow \text{Pure} \left(\mathbf{t} * \mathbf{s}_1:\text{state} \{ \text{rel } \mathbf{s}_0 \ \mathbf{s}_1 \} \right) \left(\text{req.} \left(\text{pre } \mathbf{s}_0 \ @ \ \mathbf{s}_0 \right) \right) \\ \left(\text{ens.} \left(\lambda \left(\mathbf{x}, \mathbf{s}_1 \right). \text{post } \mathbf{s}_0 \ \mathbf{x} \ \mathbf{s}_1 \ @ \ \mathbf{s}_1 \right) \right)$$

where $@$ is the **standard translation** of modal logic