Embracing monotonicity in

Danel Ahman @ INRIA Paris

based on a joint POPL 2018 paper with

Cătălin Hrițcu and Kenji Maillard @ INRIA Paris
Cédric Fournet, Aseem Rastogi, and Nikhil Swamy @ MSR

Software Science Departmental Seminar, TUT
February 12, 2018
and embracing monotonicity (in it)

Danel Ahman @ INRIA Paris

based on a joint POPL 2018 paper with

Cătălin Hrițcu and Kenji Maillard @ INRIA Paris
Cédric Fournet, Aseem Rastogi, and Nikhil Swamy @ MSR

Software Science Departmental Seminar, TUT
February 12, 2018
Outline

* F* overview

- Monotonicity (monotonic state) in programming and verification
- Key ideas behind our general extension to Hoare-style logics
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- Glimpse of the meta-theory and correctness results
  - More examples of monotonic state at work (see our paper)
  - Monadic reification and reflection (see our paper)
Outline

* F* overview

  - Monotonicity (monotonic state) in programming and verification
  - Key ideas behind our general extension to Hoare-style logics
  - Accommodating monotonic state in F*
  - Some examples of monotonic state at work
  - Glimpse of the meta-theory and correctness results
  - More examples of monotonic state at work (see our paper)
  - Monadic reification and reflection (see our paper)
**F**

- **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
    - tactics and metaprogramming (WIP)
  - a **semi-automated verifier** of imperative programs
    - Dafny, Why3, FramaC, ...
    - Z3-based SMT-automation (for discharging VCs)

- **F**'s development is driven by **Project Everest**
  - miTLS, HACL*, Vale, ...
  - Microsoft Research (US, UK, India), INRIA (Paris), ...

---

[project-everest.github.io]
F*

- 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
    - tactics and metaprogramming (WIP)
  - a semi-automated verifier of imperative programs
    - Dafny, Why3, FramaC, ...
    - Z3-based SMT-automation (for discharging VCs)
- F*’s development is driven by Project Everest
  - miTLS, HACL*, Vale, ...
  - Microsoft Research (US, UK, India), INRIA (Paris), ...

[fstar-lang.org]
// Dependent types, recursive functions, and type inference

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

let rec concat #a #n #m (xs:vector a n) (ys:vector a m) : Tot (vector a (n + m)) =
  match xs with
  | Nil -> ys
  | Cons #n x xs' -> Cons x (concat xs' ys)
// Dependent types, recursive functions, and type inference

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

let rec concat #a #n #m (xs:vector a n) (ys:vector a m) : Tot (vector a (n + m)) =
  match xs with
  | Nil -> ys
  | Cons #n x xs' -> Cons x (concat xs' ys)

// Refinement types

let in_range_index min max : Type = i:nat{min <= i \ i <= max}

let rec lkp #a #n (xs:vector a n) (i:in_range_index 1 n) : Tot a =
  match xs with
  | Cons x xs' -> if i = 1 then x else lkp xs' (i - 1)
// Dependent types, recursive functions, and type inference

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

let rec concat #a #n #m (xs:vector a n) (ys:vector a m) : Tot (vector a (n + m)) =
  match xs with
  | Nil -> ys
  | Cons #n x xs' -> Cons x (concat xs' ys)

// Refinement types

let in_range_index min max : Type = i:nat{min <= i ∧ i <= max}

let rec lkp #a #n (xs:vector a n) (i:in_range_index 1 n) : Tot a =
  match xs with
  | Cons x xs' -> if i = 1 then x else lkp xs' (i - 1)

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

type is_prefix_of #a #n #m (xs:vector a n) (zs:vector a m{n <= m}) : Type₀ =
  forall (i:nat) . (1 <= i ∧ i <= n) ==> lkp xs i == lkp zs i
// Dependent types, recursive functions, and type inference

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

let rec concat #a #n #m (xs:vector a n) (ys:vector a m) : Tot (vector a (n + m)) =
  match xs with
  | Nil -> ys
  | Cons #n x xs' -> Cons x (concat xs' ys)

// Refinement types

let in_range_index min max : Type = i:nat{min <= i \&\& i <= max}

let rec lkp #a #n (xs:vector a n) (i:in_range_index 1 n) : Tot a =
  match xs with
  | Cons x xs' -> if i = 1 then x else lkp xs' (i - 1)

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

type is_prefix_of #a #n #m (xs:vector a n) (zs:vector a m{n <= m}) : Type0 =
  forall (i:nat) . (1 <= i \&\& i <= n) ==> lkp xs i == lkp zs i

// Extrinsic reasoning (using separate lemmas)

let rec lemma #a #n #m (xs:vector a n) (ys:vector a m) : Lemma (xs `is_prefix_of` (concat xs ys)) =
  match xs with
  | Nil -> ()
  | Cons x xs' -> lemma xs' ys
// Dependent types, recursive functions, and type inference

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

let rec concat #a #n #m (xs:vector a n) (ys:vector a m) : Tot (vector a (n + m)) =
  match xs with
  | Nil -> ys
  | Cons #n x xs' -> Cons x (concat xs' ys)

// Refinement types

let in_range_index min max : Type = i:nat{min <= i \&\& i <= max}

let rec lkp #a #n (xs:vector a n) (i:in_range_index 1 n) : Tot a =
  match xs with
  | Cons x xs' -> if i = 1 then x else lkp xs' (i - 1)

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

type is_prefix_of #a #n #m (xs:vector a n) (zs:vector a m{\{n <= m\}}) : Type0 =
  forall (i:nat). (1 <= i \&\& i <= n) ==> lkp xs i == lkp zs i

// Extrinsic reasoning (using separate lemmas)

let rec lemma #a #n #m (xs:vector a n) (ys:vector a m) : Lemma (xs `is_prefix_of` (concat xs xs)) =
  match xs with
  | Nil -> ()
  | Cons x xs' -> lemma xs' ys

// Intrinsic reasoning (making lemmas part of definitions, e.g., using pre- and postconditions)

let rec take #a n #m (zs:vector a m) : Pure (vector a n) (requires (n <= m))
  (ensures (fun xs -> xs `is_prefix_of` zs)) =
  if n > 0 then match zs with
  | Cons z zs' -> let n':nat = n - 1 in Cons z (take n' zs')
  else Nil
// Heaps, ML-style typed references, and Hoare logic

open FStar.Heap
open FStar.ST

let rec program n =
  let r = alloc 0 in
  sum_loop 1 n r;
  r

and sum_loop i n r =
  if i < n then (r := !r + i; sum_loop (i + 1) n r)
  else (r := !r + n)
// Heaps, ML-style typed references, and Hoare logic

open FStar.Heap
open FStar.ST

val program : n:nat -> ST (ref nat) (requires (fun h₀ -> 1 <= n))
  (ensures (fun _ r h₁ -> sel h₁ r = sum 1 n))

let rec program n =
  let r = alloc 0 in
  sum_loop 1 n r;
  r

and sum_loop i n r =
  if i < n then (r := !r + i; sum_loop (i + 1) n r)
  else (r := !r + n)
// Heaps, ML-style typed references, and Hoare logic

open FStar.Heap
open FStar.ST

val sum : i:nat -> n:nat{i <= n} -> GTot nat (decreases (n - i))

let rec sum i n = 
  if i < n then i + sum (i + 1) n
  else n

val program : n:nat -> ST (ref nat) (requires (fun h₀ -> 1 <= n))
  (ensures (fun _ r h₁ -> sel h₁ r = sum 1 n))

let rec program n = 
  let r = alloc 0 in
  sum_loop 1 n r;
  r

and sum_loop i n r = 
  if i < n then (r := !r + i; sum_loop (i + 1) n r)
  else (r := !r + n)
// Heaps, ML-style typed references, and Hoare logic

open FStar.Heap
open FStar.ST

val sum : i:nat -> n:nat{i <= n} -> GTot nat (decreases (n - i))

let rec sum i n =  
      if i < n then i + sum (i + 1) n 
      else n

val program : n:nat -> ST (ref nat) (requires (fun h_0 -> 1 <= n))  
               (ensures (fun _ r h_1 -> sel h_1 r = sum 1 n))

val sum_loop : i:nat -> n:nat -> r:ref nat -> ST unit (requires (fun h_0 -> (1 <= i ∧ i <= n) ∧  
                                                                  (i = 1 ==> sel h_0 r = 0) ∧  
                                                                  (i > 1 ==> sel h_0 r = sum 1 (i - 1)))  
                                                                  (ensures (fun _ _ h_1 -> sel h_1 r = sum 1 n))

let rec program n =  
      let r = alloc 0 in  
      sum_loop 1 n r;  
      r

and sum_loop i n r =  
      if i < n then (r := !r + i; sum_loop (i + 1) n r)  
      else (r := !r + n)
open FStar.Heap
open FStar.ST

val sum : i:nat -> n:nat{i <= n} -> GTot nat (decreases (n - i))

let rec sum i n = 
  if i < n then i + sum (i + 1) n
  else n

val sum_plus_lemma : i:nat -> n:nat -> Lemma (requires (i <= n))
  (ensures (sum i (n + 1) = sum i n + (n + 1)))
  (decreases (n - i))
  [SMTPat (sum i n)]

let rec sum_plus_lemma i n = 
  if i < n then sum_plus_lemma (i + 1) n
  else ()

val program : n:nat -> ST (ref nat) (requires (fun h_0 -> 1 <= n))
  (ensures (fun _ r h_1 -> sel h_1 r = sum 1 n))

val sum_loop : i:nat -> n:nat -> r:ref nat -> ST unit (requires (fun h_0 -> (1 <= i ∧ i <= n) ∧
  (i = 1 ==> sel h_0 r = 0) ∧
  (i > 1 ==> sel h_0 r = sum 1 (i - 1)))
  (ensures (fun _ _ h_1 -> sel h_1 r = sum 1 n))

let rec program n = 
  let r = alloc 0 in
  sum_loop 1 n r;
  r

and sum_loop i n r = 
  if i < n then (r := !r + i; sum_loop (i + 1) n r) 
  else (r := !r + n)
F* – not just a pure programming language

- Tot, Lemma, Pure, ... are just some **effects** amongst many
  - Tot \( t \)
    - Lemma \( (\text{requires } \text{pre}_{\text{Lemma}}) (\text{ensures } \text{post}_{\text{Lemma}}) \)
    - Pure \( t (\text{requires } \text{pre}_{\text{Pure}}) (\text{ensures } \text{post}_{\text{Pure}}) \)
    - Div \( t (\text{requires } \text{pre}_{\text{Div}}) (\text{ensures } \text{post}_{\text{Div}}) \)
    - Exc \( t (\text{requires } \text{pre}_{\text{Exc}}) (\text{ensures } \text{post}_{\text{Exc}}) \)
    - ST \( t (\text{requires } \text{pre}_{\text{ST}}) (\text{ensures } \text{post}_{\text{ST}}) \)
    - ...

- **Monad morphs.**  Pure \( \rightsquigarrow \{ \text{Div}, \text{Exc}, \text{ST} \} \);  Exc \( \rightsquigarrow \text{STExc} \);  ...

- Systematically derived from **WP-calculi**  [POPL 2017]
Outline

* F* overview

- Monotonicity (monotonic state) in programming and verification
- Key ideas behind our general extension to Hoare-style logics
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- Glimpse of the meta-theory and correctness results
- More examples of monotonic state at work (see our paper)
- Monadic reification and reflection (see our paper)
Monotonicity in program verification

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

```plaintext
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**

```plaintext
{λs.v ∈ s} complex_procedure() {λs.v ∈ s}
```

- Likely that we have to **carry** \(λs.v ∈ s\) **through** the proof of \(c_p\)
- **Does not guarantee** that \(λs.v ∈ s\) holds at every point in \(c_p\)
- **Sensitive** to proving that \(c_p\) maintains \(λs.w ∈ s\) for some \(w\)

- However, if \(c_p\) **never removes**, then \(λs.v ∈ s\) 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

\[
\text{insert } v; \text{ complex\_procedure()}; \text{ assert } (v \in \text{get()})
\]

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

\[
\{\lambda s. v \in s\} \text{ complex\_procedure() } \{\lambda s. v \in s\}
\]

- likely that we have to carry \(\lambda s. v \in s\) through the proof of \(\text{c\_p}\)
- does not guarantee that \(\lambda s. v \in s\) holds at every point in \(\text{c\_p}\)
- sensitive to proving that \(\text{c\_p}\) maintains \(\lambda s. w \in s\) for some \(w\)

- However, if \(\text{c\_p}\) 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

\[
\text{insert } v; \text{ complex\_procedure}(); \text{ assert } (v \in \text{get}())
\]

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

\[
\{\lambda s. v \in s\} \text{ complex\_procedure}() \{\lambda s. v \in s\}
\]

• 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

\[
\text{insert } v; \text{ complex\_procedure(); assert } \ (v \in \text{get()})
\]

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

\[
\{ \lambda s. v \in s \} \text{ complex\_procedure()} \ {\lambda s. v \in s}
\]

  • 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 programming

- **Programming** also relies on **monotonicity**, even if you don’t realise it!

  Consider ML-style typed references `r:ref a`
  - `r` is a proof of existence of an `a`-typed value in the heap

  Correctness relies on **monotonicity**!
  - **i**) Allocation stores an `a`-typed value in the heap
  - **ii**) Writes don’t change type and there is no deallocation
  - **iii**) 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:ref a`
  - `r` is a proof of existence of an `a`-typed value in the heap

- Correctness relies on monotonicity!
  - i) Allocation stores an `a`-typed value in the heap
  - ii) Writes don’t change type and there is no deallocation
  - iii) 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 ref a`
  - `r` is a **proof of existence** of an `a`-typed value in the heap

- Correctness relies on **monotonicity**!
  - i) Allocation **stores** an `a`-typed value in the heap
  - ii) Writes **don’t change type** and there is **no deallocation**
  - iii) 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!
  - i) Allocation stores an \( a \)-typed value in the heap
  - ii) Writes don’t change type and there is no deallocation
  - iii) 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 (\texttt{ref t}) and untyped references (\texttt{uref})
  - more flexibility with monotonic references (\texttt{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** \((\text{ref } t)\) and **untyped references** \((\text{uref})\)
  - more flexibility with **monotonic references** \((\text{mref } t \text{ rel})\)

- 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 ($\text{ref } t$) and untyped references ($\text{uref}$)
  - more flexibility with monotonic references ($\text{mref } t \text{ rel}$)

- 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* overview
  - Monotonicity (monotonic state) in programming and verification
  - Key ideas behind our general extension to Hoare-style logics
  - Accommodating monotonic state in F*
  - Some examples of monotonic state at work
  - Glimpse of the meta-theory and correctness results
  - More examples of monotonic state at work (see our paper)
  - Monadic reification and reflection (see our 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, \ldots
  - a stateful program \( e \) is monotonic (wrt. \( rel \)) when
    \[
    \forall s e' s'. (e, s) \leadsto^* (e', s') \implies rel s s'
    \]
  - a stateful predicate \( p \) is stable (wrt. \( rel \)) when
    \[
    \forall s s'. p s \land rel s s' \implies p s'
    \]
- Our solution: extend Hoare-style program logics (e.g., F*) with
  - i) a means to witness the validity of \( p s \) in some state \( s \)
  - ii) a means for turning a \( p \) into a state-independent proposition
  - iii) 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 \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program \( e \) is monotonic (wrt. \( \text{rel} \)) when
    \[
    \forall s e' s'. (e, s) \leadsto^* (e', s') \implies \text{rel } s s'
    \]
  - a stateful predicate \( p \) is stable (wrt. \( \text{rel} \)) when
    \[
    \forall s s'. p s \land \text{rel } s s' \implies p s'
    \]
  - **Our solution**: extend Hoare-style program logics (e.g., F*) with
    i) a means to witness the validity of \( p s \) in some state \( s \)
    ii) a means for turning a \( p \) into a state-independent proposition
    iii) 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** \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, ... 
  - a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    \[
    \forall s e' s'. (e, s) \leadsto^* (e', s') \implies \text{rel} s s'
    \]
  - a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
    \[
    \forall s s'. p s \land \text{rel} s s' \implies p s'
    \]
- **Our solution**: extend Hoare-style program logics (e.g., F*) with
  i) a means to **witness** the validity of \( p s \) in some state \( s \)
  ii) a means for turning a \( p \) into a **state-independent proposition**
  iii) 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** \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, …
  - a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    \[
    \forall s \ e' \ s'. \ (e, s) \leadsto^* (e', s') \implies \text{rel} \ s \ s'
    \]
  - a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
    \[
    \forall s \ s'. \ p \ s \land \text{rel} \ s \ s' \implies p \ s'
    \]

- **Our solution**: extend Hoare-style program logics (e.g., F*) with
  i) a means to **witness** the validity of \( p \ s \) in some state \( s \)
  ii) a means for turning a \( p \) into a state-independent proposition
  iii) 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** $\text{rel}$ on states
    - set inclusion, heap inclusion, increasing counter values, . . .
  - a stateful program $e$ is **monotonic** (wrt. $\text{rel}$) when
    $$\forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s s'$$
  - a stateful predicate $p$ is **stable** (wrt. $\text{rel}$) when
    $$\forall s s'. p s \land \text{rel } s s' \implies p s'$$

- Our solution: extend Hoare-style program logics (e.g., F*) with
  1. a means to **witness** the validity of $p s$ in some state $s$
  2. a means for turning a $p$ into a state-independent proposition
  3. 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** \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, . . .
  - a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    \[
    \forall s \ e' \ s'. \ (e, s) \sim^* (e', s') \implies \text{rel} \ s \ s'
    \]
  - a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
    \[
    \forall s \ s'. \ p \ s \land \text{rel} \ s \ s' \implies p \ s'
    \]
- **Our solution**: extend Hoare-style program logics (e.g., F*) with
  - i) a means to **witness** the validity of \( p \ s \) in some state \( s \)
  - ii) a means for turning a \( p \) into a **state-independent proposition**
  - iii) 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** \( \text{rel} \) on states
    
    • set inclusion, heap inclusion, increasing counter values, \ldots

  • a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    
    \[
    \forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel} \ s \ s'
    \]

  • a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
    
    \[
    \forall s s'. p \ s \land \text{rel} \ s \ s' \implies p \ s'
    \]

• **Our solution:** extend Hoare-style program logics (e.g., F*) with
  
  i) a means to **witness** the validity of \( p \ s \) in some state \( s \)
  
  ii) a means for turning a \( p \) into a **state-independent proposition**
  
  iii) 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* overview

- Monotonicity (monotonic state) in programming and verification
- Key ideas behind our general extension to Hoare-style logics
- **Accommodating monotonic state in F***
- Some examples of monotonic state at work
- Glimpse of the meta-theory and correctness results
- More examples of monotonic state at work (see our paper)
- Monadic reification and reflection (see our 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

\[
\text{pre} : \text{state } \to \text{Type} \quad \text{post} : \text{state } \to t \to \text{state } \to \text{Type}
\]

- ST is an abstract pre-postcondition refinement of

\[
\text{st } t \ \text{def} = \text{state } \to t \ast \text{state}
\]

- The global state actions have types

\[
\text{get} : \text{unit } \to \text{ST } \text{state } (\text{requires } (\lambda \ _ . \ Top)) \ (\text{ensures } (\lambda s_0 s s_1 . s_0 = s = s_1))
\]

\[
\text{put} : s:\text{state } \to \text{ST } \text{unit } (\text{requires } (\lambda \ _ . \ Top)) \ (\text{ensures } (\lambda _ s s_1 . s_1 = s))
\]

- 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

\[
\text{pre} : \text{state} \rightarrow \text{Type} \quad \text{post} : \text{state} \rightarrow t \rightarrow \text{state} \rightarrow \text{Type}
\]

- ST is an abstract pre-postcondition refinement of

\[
\text{st} \ t \ \overset{\text{def}}{=} \text{state} \rightarrow t \ast \text{state}
\]

- The global state actions have types

\[
\begin{align*}
\text{get} &: \text{unit} \rightarrow \text{ST} \ \text{state} \ (\text{requires} \ (\lambda \_ . \top)) \ (\text{ensures} \ (\lambda s_0 s s_1 . s_0 = s = s_1)) \\
\text{put} &: \text{state} \rightarrow \text{ST} \ \text{unit} \ (\text{requires} \ (\lambda \_ . \top)) \ (\text{ensures} \ (\lambda \_ s_1 . s_1 = s))
\end{align*}
\]

- 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**

\[
\text{ST} \ #\text{state} \ t \ (\text{requires pre}) \ (\text{ensures post})
\]

where

\[
\text{pre} : \text{state} \to \text{Type} \quad \text{post} : \text{state} \to t \to \text{state} \to \text{Type}
\]

- \(\text{ST}\) is an abstract pre-postcondition refinement of

\[
\text{st} \ t \ \overset{\text{def}}{=} \text{state} \to t \ast \text{state}
\]

- The global state **actions** have types

\[
\begin{align*}
\text{get} : \text{unit} \to \text{ST} \ \text{state} \ (\text{requires } (\lambda_.\top)) \ (\text{ensures } (\lambda s_0 s s_1. s_0 = s = s_1)) \\
\text{put} : s : \text{state} \to \text{ST} \ \text{unit} \ (\text{requires } (\lambda_.\top)) \ (\text{ensures } (\lambda_. s_1. s_1 = s))
\end{align*}
\]

- 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 \texttt{comp. type}
  \[
  \text{ST} \#\text{state} \ t \ (\text{requires} \ \text{pre}) \ (\text{ensures} \ \text{post})
  \]
  
  where
  \[
  \text{pre} : \text{state} \rightarrow \text{Type} \quad \text{post} : \text{state} \rightarrow t \rightarrow \text{state} \rightarrow \text{Type}
  \]

- \text{ST} is an abstract pre-postcondition refinement of
  \[
  \text{st} \ t \ \overset{\text{def}}{=} \text{state} \rightarrow t * \text{state}
  \]

- The global state \texttt{actions} have types
  \[
  \text{get} : \text{unit} \rightarrow \text{ST} \ \text{state} \ (\text{requires} \ (\lambda \_ . \top)) \ (\text{ensures} \ (\lambda s_0 s s_1 . s_0 = s = s_1))
  \]
  \[
  \text{put} : s : \text{state} \rightarrow \text{ST} \ \text{unit} \ (\text{requires} \ (\lambda \_ . \top)) \ (\text{ensures} \ (\_ \_ s_1 . s_1 = s))
  \]

- \texttt{Refs.} and \texttt{local state} are defined in F* using \texttt{monotonicity}
New: Monotonic global state in F*

- We capture monotonic state with a new computational type

\[
\text{MST } \#\text{state } \#\text{rel } t \ (\text{requires pre}) \ (\text{ensures post})
\]

- The `get` action is typed as in ST

\[
\text{get} : \text{unit} \rightarrow \text{MST state} \ (\text{requires } (\lambda_.\top))
\]
\[
(\text{ensures } (\lambda s_0 ss_1. s_0 = s = s_1))
\]

- 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

\[
\text{mst } t \overset{\text{def}}{=} s_0:\text{state} \rightarrow t * s_1:\text{state}\{\text{rel } s_0 s_1}\]
New: Monotonic global state in F*

- We capture monotonic state with a new computational type

\[
\text{MST \#state \#rel t (requires pre) (ensures post)}
\]

- The `get` action is typed as in ST

\[
\text{get : unit } \to \text{MST state (requires (λ_.T))}
\]
\[
\text{(ensures (λs_0 ss_0 . s_0 = s = s_1))}
\]

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

\[
\text{put : s:state } \to \text{MST unit (requires (λs_0 . rel s_0 s))}
\]
\[
\text{(ensures (λ__s_1 . s_1 = s))}
\]

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

\[
\text{mst t}^{\text{def}} = s_0 : \text{state } \to t \ast s_1 : \text{state \{rel s_0 s_1\}}
\]
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 \textbf{get} action is typed as in ST

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

• To ensure monotonicity, the \textbf{put} action gets a precondition

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

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

\[ \text{mst} \ t \overset{\text{def}}{=} s_0 : \text{state} \rightarrow t * s_1 : \text{state}\{\text{rel} \ s_0 s_1\} \]
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 \text{get} action is typed as in \text{ST}

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

- To ensure \textit{monotonicity}, the \text{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, \text{MST} is an \textit{abstract} pre-postcondition refinement of

\[ \text{mst} \ t \ \overset{\text{def}}{=} \ s_0:\text{state} \rightarrow t * s_1:\text{state}\{\text{rel} \ s_0 \ s_1\} \]
New: Monotonic global state in F*

- We capture monotonic state with a new computational type
  \[ \text{MST} \#\text{state} \#\text{rel} \ t \ (\text{requires pre}) (\text{ensures post}) \]

- The \textbf{get} action is typed as in ST
  \[
  \begin{align*}
  \text{get} &: \text{unit} \to \text{MST} \ \text{state} \\
  (\text{requires} &: \lambda_.\top) \\
  (\text{ensures} &: \lambda s_0 s s_1. s_0 = s = s_1)
  \end{align*}
  \]

- To ensure \textbf{monotonicity}, the \textbf{put} action gets a precondition
  \[
  \begin{align*}
  \text{put} &: \text{s:state} \to \text{MST} \ \text{unit} \\
  (\text{requires} &: \lambda s_0 . \text{rel} s_0 s) \\
  (\text{ensures} &: \lambda _. \text{s}_1. s_1 = s)
  \end{align*}
  \]

- So intuitively, \textbf{MST} is an \textbf{abstract} pre-postcondition refinement of
  \[
  \text{mst} \ t \overset{\text{def}}{=} s_0: \text{state} \to t \ast s_1: \text{state} \{\text{rel} s_0 s_1\}
  \]
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

\[ [\text{witnessed} p](s) \overset{\text{def}}{=} p \text{ 'stable from' } s \]

\[ \overset{\text{def}}{=} \forall s'. \text{rel} s s' \implies [p s'](s) \]

- As usual, for natural deduction, need world-indexed sequents

[Simpson’94; Russo’96; Basin, Matthews, Vigano’98]
New: Recalling a Witness

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

\[
\text{witnessed} : (\text{state} \to \text{Type}) \to \text{Type}
\]

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

\[
wk : p, q : (\text{state} \to \text{Type}) \to \text{Lemma} (\text{requires} (\forall s. p s \implies q s))
\]

\[
(\text{ensures} (\text{witnessed} p \implies \text{witnessed} q))
\]

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

\[
[witnessed p](s) \overset{\text{def}}{=} p \ '\text{stable from} \ ' s
\]

\[
\overset{\text{def}}{=} \forall s'. \text{rel} s s' \implies [p s'](s)
\]

- As usual, for natural deduction, need **world-indexed sequents**

[Simpson’94; Russo’96; Basin, Matthews, Vigano’98]
New: Recalling a Witness

- We extend F* with a **logical capability**
  
  \[
  \text{witnessed} : (\text{state} \to \text{Type}) \to \text{Type}
  \]

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

  \[
  \text{wk} : p, q : (\text{state} \to \text{Type}) \to \text{Lemma} (\text{requires} (\forall s. p s \implies q s))
  \]

  \[
  (\text{ensures} (\text{witnessed} p \implies \text{witnessed} q))
  \]

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

  \[
  \llbracket \text{witnessed} p \rrbracket (s) \overset{\text{def}}{=} p \text{ \textbackslash 'stable\_from\textbackslash '} s
  \]

  \[
  \overset{\text{def}}{=} \forall s'. \text{rel} s s' \implies \llbracket p s' \rrbracket (s)
  \]

- As usual, for natural deduction, need **world-indexed sequents**
  
  [Simpson’94; Russo’96; Basin, Matthews, Vigano’98]
New: Recalling a Witness

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

  \[ \text{witnessed} : (\text{state} \to \text{Type}) \to \text{Type} \]

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

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

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

  \[ [\text{witnessed } p](s) \overset{\text{def}}{=} p \text{ `stable from` } s \]
  \[ \overset{\text{def}}{=} \forall s'. \text{rel } s s' \implies [p s'](s) \]

- As usual, for natural deduction, need **world-indexed sequents**

  [Simpson’94; Russo’96; Basin, Matthews, Vigano’98]
New: Recalling a Witness

• But, wait a minute . . .

• . . . Hoare-style logics are essentially world/state-indexed, so

we include a stateful introduction rule for witnessed

\[
\text{witness} : \ p : (\text{state} \rightarrow \text{Type}_0) \\
\rightarrow \ \text{MST unit} \ (\text{requires} \ (\lambda s_0.p \ '\text{stable\_from}' \ s_0)) \\
(\text{ensures} \ (\lambda s_0\ s_1. s_0 = s_1 \land \text{witnessed } p))
\]

• and a stateful elimination rule for witnessed

\[
\text{recall} : \ p : (\text{state} \rightarrow \text{Type}_0) \\
\rightarrow \ \text{MST unit} \ (\text{requires} \ (\lambda _. \text{witnessed } p)) \\
(\text{ensures} \ (\lambda s_0\ s_1. s_0 = s_1 \land p \ '\text{stable\_from}' \ s_1))
\]
New: Recalling a Witness

- But, wait a minute . . .
- . . . Hoare-style logics are essentially world/state-indexed, so

- we include a stateful introduction rule for witnessed

  \[
  \text{witness} : \ p : (\text{state} \to \text{Type}_0) \\
  \rightarrow \text{MST unit} \ (\text{requires} \ (\lambda s_0. p \ '\text{stable_from}' \ s_0)) \\
  \ (\text{ensures} \ (\lambda s_0 s_1. s_0 = s_1 \land \text{witnessed} \ p))
  \]

- and a stateful elimination rule for witnessed

  \[
  \text{recall} : \ p : (\text{state} \to \text{Type}_0) \\
  \rightarrow \text{MST unit} \ (\text{requires} \ (\lambda _. \ \text{witnessed} \ p)) \\
  \ (\text{ensures} \ (\lambda s_0 s_1. s_0 = s_1 \land p \ '\text{stable_from}' \ s_1))
  \]
New: Recalling a Witness

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

\[
\text{witness} : \ p : (\text{state} \to \text{Type}_0) \\
\rightarrow \ \text{MST unit (requires } \lambda s_0. p \text{ 'stable_from' } s_0) \\
(\text{ensures } \lambda s_0 \_ s_1. s_0 = s_1 \land \text{witnessed } p))
\]

- and a stateful elimination rule for witnessed

\[
\text{recall} : \ p : (\text{state} \to \text{Type}_0) \\
\rightarrow \ \text{MST unit (requires } \lambda \_. \text{witnessed } p) \\
(\text{ensures } \lambda s_0 \_ s_1. s_0 = s_1 \land p \text{ 'stable_from' } s_1))
\]
New: Recalling a Witness

• But, wait a minute . . .

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

\[
\begin{align*}
\text{witness} & : \ p : (\text{state} \to \text{Type}_0) \\
& \to \text{MST unit} \ (\text{requires} (\lambda s_0. p \ '\text{stable\_from'} \ s_0)) \\
& \quad \quad \quad \ (\text{ensures} (\lambda s_0 \ s_1. s_0 = s_1 \land \text{witnessed} \ p))
\end{align*}
\]

• and a stateful elimination rule for witnessed

\[
\begin{align*}
\text{recall} & : \ p : (\text{state} \to \text{Type}_0) \\
& \to \text{MST unit} \ (\text{requires} (\lambda \_ . \text{witnessed} \ p)) \\
& \quad \quad \quad \ (\text{ensures} (\lambda s_0 \ s_1. s_0 = s_1 \land p \ '\text{stable\_from'} \ s_1))
\end{align*}
\]
Outline

* F* overview

- Monotonicity (monotonic state) in programming and verification
- Key ideas behind our general extension to Hoare-style logics
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- Glimpse of the meta-theory and correctness results
- More examples of monotonic state at work (see our paper)
- Monadic reification and reflection (see our paper)
The motivating example revisited

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

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

- we pick set **inclusion** $\subseteq$ as our preorder rel on states

- we prove the **assertion** by inserting a witness and recall

```latex
insert v; witness (\lambda s. v \in s); c_p(); recall (\lambda s. v \in s); assert (v \in \text{get()})
```

- for any other $w$, wrapping

```latex
insert w; []; assert (w \in \text{get()})
```

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

```latex
insert w; witness (\lambda s. w \in s); []; recall (\lambda s. w \in s); assert (w \in \text{get()})
```

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

```latex
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**

  \[
  \text{insert } v; \text{complex\_procedure}(); \text{assert } (v \in \text{get}())
  \]

- we pick **set inclusion \( \subseteq \)** as our preorder \( \text{rel on states} \)

- we prove the assertion by inserting a witness and recall

  \[
  \text{insert } v; \text{witness } (\lambda s. v \in s); \text{c\_p}(); \text{recall } (\lambda s. v \in s); \text{assert } (v \in \text{get}())
  \]

- for any other \( w \), wrapping

  \[
  \text{insert } w; [ ]; \text{assert } (w \in \text{get}())
  \]

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

  \[
  \text{insert } w; \text{witness } (\lambda s. w \in s); [ ]; \text{recall } (\lambda s. w \in s); \text{assert } (w \in \text{get}())
  \]

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

  \[
  \text{create } 0; \text{incr}(); \text{witness } (\lambda c. c > 0); \text{c\_p}(); \text{recall } (\lambda c. c > 0)
  \]
The motivating example revisited

- Recall the program operating on the **set-valued state**
  
  \[
  \text{insert } v; \text{ complex\_procedure()}; \text{ assert } (v \in \text{get()})
  \]

- we pick **set inclusion** \( \subseteq \) as our preorder rel on states

- we **prove the assertion** by inserting a witness and recall
  
  \[
  \text{insert } v; \text{ witness } (\lambda s. v \in s); \text{ cp()}; \text{ recall } (\lambda s. v \in s); \text{ assert } (v \in \text{get()})
  \]

- for any other \( w \), wrapping
  
  \[
  \text{insert } w; \text{ [ ]}; \text{ assert } (w \in \text{get()})
  \]

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

  \[
  \text{insert } w; \text{ witness } (\lambda s. w \in s); \text{ [ ]}; \text{ recall } (\lambda s. w \in s); \text{ assert } (w \in \text{get()})
  \]

- **Monotonic counters** are analogous, by picking \( \mathbb{N} \) and \( \leq \), e.g.,
  
  \[
  \text{create } 0; \text{ incr(); witness } (\lambda c. c > 0); \text{ cp(); recall } (\lambda c. c > 0)
  \]
The motivating example revisited

• Recall the program operating on the \textbf{set-valued state}

\begin{verbatim}
insert v; complex_procedure(); assert (v \in get())
\end{verbatim}

• we pick \textbf{set inclusion} $\subseteq$ as our preorder rel on states

• we \textbf{prove the assertion} by inserting a witness and recall

\begin{verbatim}
insert v; witness (\lambda s. v \in s); c_p(); recall (\lambda s. v \in s); assert (v \in get())
\end{verbatim}

• for any other $w$, wrapping

\begin{verbatim}
insert w; [ ]; assert (w \in get())
\end{verbatim}

around the program is handled \textit{similarly} \textbf{easily} by

\begin{verbatim}
insert w; witness (\lambda s. w \in s); [ ]; recall (\lambda s. w \in s); assert (w \in get())
\end{verbatim}

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

\begin{verbatim}
create 0; incr(); witness (\lambda c. c > 0); c_p(); recall (\lambda c. c > 0)
\end{verbatim}
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

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

where

    type cell =

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

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

```plaintext
let heap.inclusion (H h₀ _) (H h₁ _) =

∀ r.match h₀ r, h₁ r with

  | Unused, _ → ⊤
  | Used a _, Used b _ → a = b
  | Used _, Unused → ⊥
ML-style typed references (local state)

- First, we define a type of **heaps** as a finite map
  
  ```plaintext
type heap =
  | H : h:(N -> cell) -> ctr:N{∀r.ctr ≤ r => h r = Unused} -> heap

where

type cell =
  | Unused : cell
  | Used : a:Type -> v:a -> cell
  ```

- Next, we define a **preorder** on heaps (**heap inclusion**)
  
  ```plaintext
let heap_inclusion (H h₀ _) (H h₁ _) =
  ∀r.match h₀ r, h₁ r with
  | Unused, _ → ⊤
  | Used a _, Used b _ → a = b
  | Used _, Unused → ⊥
  ```
ML-style typed references (local state)

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

\[
\begin{align*}
type \ heap &= \\
| \ H : h : (\mathbb{N} \rightarrow cell) \rightarrow ctr : \mathbb{N} \{ r . ctr \leq r \implies h r = Unused \} \rightarrow heap
\end{align*}
\]

where

\[
\begin{align*}
type \ cell &= \\
| \ Unused : cell \\
| \ Used : a : Type \rightarrow v : a \rightarrow cell
\end{align*}
\]

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

\[
\begin{align*}
let \ heap\_inclusion \ (H \ h_0 \ _) \ (H \ h_1 \ _) &= \\
\forall r . match \ h_0 \ r , h_1 \ r \ with \\
| \ Unused , _ \ \rightarrow \ \top \\
| \ Used \ a \ _, \ Used \ b \ _ \ \rightarrow \ a = b \\
| \ Used \ _ \ , \ Unused \ \rightarrow \ \bot
\end{align*}
\]
ML-style typed references (local state)

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

\[ \text{MLST } t \text{ pre post } \overset{\text{def}}{=} \text{MST } \#\text{heap } \#\text{heap_inclusion } t \text{ pre post} \]

- Next, we define the type of references using monotonicity

\[
\text{abstract type ref } a = r:N\{\text{witnessed } (\lambda h. \text{contains } h r a)\}
\]

where

\[
\text{let contains } (H h _) r a =
\]

\[
\begin{align*}
\text{match } h r \text{ with} \\
| \text{Used } b _ & \rightarrow a = b \\
| \text{Unused } & \rightarrow \bot
\end{align*}
\]

- **Important:** contains is stable wrt. heap_inclusion
ML-style typed references (local state)

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

\[
\text{MLST } t \ pre \ post \ \overset{\text{def}}{=} \text{MST } \#\text{heap } \#\text{heap_inclusion } t \ pre \ post
\]

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

\[
\text{abstract type } \text{ref } a = r: N \{ \text{witnessed } (\lambda h. \text{contains } h \ r \ a) \}
\]

where

\[
\text{let contains } (H \ h \ _) \ r \ a =
\]

\[
\text{match } h \ r \ \text{with}
\]

| Used \ b \ _ \ → \ a = b |
| Unused \ → \ \perp |

- Important: contains is **stable** wrt. heap_inclusion
ML-style typed references (local state)

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

  $$\text{MLST } t \text{ pre post} \overset{\text{def}}{=} \text{MST } \#\text{heap } \#\text{heap_inclusion } t \text{ pre post}$$

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

  $$\text{abstract type } \text{ref } a = r: \mathbb{N}\{\text{witnessed } (\lambda h. \text{contains } h \ r \ a)\}$$

  where

  $$\text{let contains } (H \ h \ _) \ r \ a =$$

  $$\text{match } h \ r \ \text{with}$$

  $$| \text{Used } b \_ \rightarrow a = b$$

  $$| \text{Unused } \rightarrow \bot$$

- **Important:** contains is **stable** wrt. heap_inclusion
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 ! (r:ref a) : MLST a (req. (⊤)) (ens. (...)) = ...
  - recall that the given ref. is in the heap
  - get the current heap
  - select the given reference from the heap

- let := (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
Finally, we define $\text{MLST}$’s actions using $\text{MST}$’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 ! (r:ref a): MLST a (req. (⊤)) (ens. (...)) = ...**
  - recall that the given ref. is in the heap
  - get the current heap
  - select the given reference from the heap

- **let := (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
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 ! (r:ref a): MLST a (req. (⊤)) (ens. (...)) = ...**
  - **recall** that the given ref. is in the heap
  - **get** the current heap
  - **select** the given reference from the heap

- **let := (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: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 ! (r:ref a) : MLST a (req. (⊤)) (ens. (...)) = ...
    - recall that the given ref. is in the heap
    - get the current heap
    - select the given reference from the heap

  - let := (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
Adding untyped and monotonic references

- **Untyped references** (uref) with strong updates
  - Used heap cells are extended with tags
    
    | Used : a:Type → v:a → t:tag → 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
    
    | Used : a:Type → v:a → t:tag a → 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
    
    | Used : a:Type → v:a → t:tag → 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**
    
    | Used : a:Type → v:a → t:tag a → 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**
    - \( \text{Used} : a : \text{Type} \rightarrow v : a \rightarrow t : \text{tag} \rightarrow \text{cell} \)
    - \( \text{type tag} = \text{Typed} : \text{tag} \mid \text{Untyped} : \text{tag} \)
  - actions corresponding to `urefs` have **weaker types** than for `refs`

- **Monotonic references** (`mref a rel`)
  - Used heap cells are extended with **typed tags**
    - \( \text{Used} : a : \text{Type} \rightarrow v : a \rightarrow t : \text{tag} a \rightarrow \text{cell} \)
    - \( \text{type tag a} = \text{Typed} : \text{rel:preorder} a \rightarrow \text{tag a} \mid \text{Untyped} : \text{tag a} \)
  - `mrefs` provide **more flexibility** with ref.-wise monotonicity
Adding untyped and monotonic references

- **Untyped references** (uref) with strong updates
  - Used heap cells are extended with **tags**
    
    \[
    \text{Used : a:Type} \to v:a \to t:tag \to \text{cell}
    \]

    where
    
    \[
    \text{type tag} = \text{Typed : tag} \mid \text{Untyped : tag}
    \]

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

- **Monotonic references** (mref a rel)
  - Used heap cells are extended with **typed tags**
    
    \[
    \text{Used : a:Type} \to v:a \to t:tag \to \text{cell}
    \]

    where
    
    \[
    \text{type tag a} = \text{Typed : rel:preorder a} \to \text{tag a} \mid \text{Untyped : tag a}
    \]

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

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

- F* overview
  - Monotonicity (monotonic state) in programming and verification
  - Key ideas behind our general extension to Hoare-style logics
  - Accommodating monotonic state in F*
  - Some examples of monotonic state at work

- Glimpse of the meta-theory and correctness results
  - More examples of monotonic state at work (see our paper)
  - Monadic reification and reflection (see our paper)
Glimpse of meta-theory

- A small **dependently typed** \(\lambda\)-calculus with \(\text{Tot}\) and \(\text{MST}\) effects

- Using an **instrumented operational semantics**, where

\[
\begin{align*}
\text{(witness } p, s, W) & \leadsto (\text{return } (), s, W \cup \{p\}) \\
\text{(recall } p, s, W) & \leadsto (\text{return } (), s, W)
\end{align*}
\]

- **Strong normalisation** via type-erasure and \(\top\top\)-lifting

- **Logical consistency** of pre-post cond. logic via cut elimination

- **Hoare-style total correctness** via SN, progress, and preservation

\[
\begin{align*}
\text{if } & \vdash e : \text{MST } t \text{ pre post } \quad \text{and} \\
& \vdash (s, W) \text{ wf } \quad \text{and} \quad \text{witnessed } W \vdash \text{pre } s \\
\text{then } & (e, s, W) \leadsto^* (\text{return } v, s', W') \quad \text{and} \quad \vdash v : t \quad \text{and} \\
& \text{witnessed } W' \vdash \text{rel } s \quad s' \quad \text{and} \quad W \subseteq W' \quad \text{and} \\
& \text{witnessed } W' \vdash \text{post } s \quad v \quad s'
\end{align*}
\]
Glimpse of meta-theory

- A small **dependently typed $\lambda$-calculus** with $\text{Tot}$ and $\text{MST}$ effects
- Using an **instrumented operational semantics**, where
  
  \[
  (\text{witness } p, s, W) \rightsquigarrow (\text{return } (), s, W \cup \{p\})
  \]
  
  \[
  (\text{recall } p, s, W) \rightsquigarrow (\text{return } (), s, W)
  \]

- **Strong normalisation** via type-erasure and $\top \top$-lifting
- Logical consistency of pre-post cond. logic via cut elimination
- **Hoare-style total correctness** via SN, progress, and preservation

\[
\text{if } \vdash e : \text{MST } t \text{ pre post } \text{ and } \\
\vdash (s, W) \text{ wf } \text{ and } \text{witnessed } W \vdash \text{ pre } s
\]

\[
\text{then } (e, s, W) \rightsquigarrow^* (\text{return } v, s', W') \text{ and } \vdash v : t \text{ and } \\
\text{witnessed } W' \vdash \text{ rel } s s' \text{ and } W \subseteq W' \text{ and } \\
\text{witnessed } W' \vdash \text{ post } s v s'
\]
Glimpse of meta-theory

- A small dependently typed $\lambda$-calculus with $\text{Tot}$ and $\text{MST}$ effects

- Using an instrumented operational semantics, where
  
  $$(\text{witness } p, s, W) \leadsto (\text{return } (), s, W \cup \{p\})$$
  
  $$\quad\quad\quad\quad (\text{recall } p, s, W) \leadsto (\text{return } (), s, W)$$

- Strong normalisation via type-erasure and $\top\top$-lifting

- Logical consistency of pre-post cond. logic via cut elimination

- Hoare-style total correctness via SN, progress, and preservation

  if $\vdash e : \text{MST } t \text{ pre post}$ and

  $\vdash (s,W) \text{ wf }$ and witnessed $W \vdash \text{ pre } s$

  then $(e,s,W) \leadsto^* (\text{return } v, s', W')$ and $\vdash v : t$ and

  witnessed $W' \vdash \text{ rel } s s'$ and $W \subseteq W'$ and

  witnessed $W' \vdash \text{ post } s v s'$
Glimpse of meta-theory

- A small dependently typed $\lambda$-calculus with \textit{Tot} and \textit{MST} effects
  
- Using an instrumented operational semantics, where

  $(\text{witness } p, s, W) \rightsquigarrow (\text{return } (), s, W \cup \{p\})$

  $(\text{recall } p, s, W) \rightsquigarrow (\text{return } (), s, W)$

- Strong normalisation via type-erasure and $\top\top$-lifting

- Logical consistency of pre-post cond. logic via cut elimination

  - Hoare-style total correctness via SN, progress, and preservation
    
    \[
    \text{if } \vdash \ e : \text{MST } t \ \text{pre } \text{post } \ \text{and} \\
    \vdash (s, W) \text{ wf } \ \text{and } \ \text{witnessed } W \vdash \text{pre } s \\
    \text{then } (e, s, W) \rightsquigarrow^* (\text{return } v, s', W') \ \text{and } \ \vdash v : t \ \text{and} \\
    \text{witnessed } W' \vdash \text{rel } s \ s' \ \text{and } \ W \subseteq W' \ \text{and} \\
    \text{witnessed } W' \vdash \text{post } s \ v \ s'
    \]
Glimpse of meta-theory

- A small dependently typed $\lambda$-calculus with $\text{Tot}$ and $\text{MST}$ effects
- Using an instrumented operational semantics, where
  
  $\text{witness } p, s, W \xrightarrow{\text{return } ()} s, W \cup \{p\}$
  
  $\text{recall } p, s, W \xrightarrow{\text{return } ()} s, W$

- Strong normalisation via type-erasure and $\top \top$-lifting
- Logical consistency of pre-post cond. logic via cut elimination
- Hoare-style total correctness via SN, progress, and preservation

\[
\begin{align*}
\text{if } & \vdash e : \text{MST } t \text{ pre post } \quad \text{and} \\
& \vdash (s, W) \text{ wf } \quad \text{and} \quad \text{witnessed } W \vdash \text{pre } s \\
\text{then } & (e, s, W) \xrightarrow{\ast} (\text{return } v, s', W') \quad \text{and} \quad \vdash v : t \quad \text{and} \\
& \text{witnessed } W' \vdash \text{rel } s s' \quad \text{and} \quad W \subseteq W' \quad \text{and} \\
& \text{witnessed } W' \vdash \text{post } s v s'
\end{align*}
\]
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**
  • details of the **meta-theory** for **MST**
  • first steps towards **monadic reification** for **MST** (rel. reasoning)

• Ongoing: taking the **modality** aspect of witnessed seriously
  • to remove instrumentation from op. sem., and
  • to improve support for monadic reification
Thank you for your attention!

Questions?


Recalling a Witness: Foundations and Applications of Monotonic State