

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*** 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]
 - miTLS, HACL*, Vale, ...
 - Microsoft Research (US, UK, India), INRIA (Paris), ...

- **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 [\[project-everest.github.io\]](https://project-everest.github.io)
 - miTLS, HACL*, Vale, ...
 - Microsoft Research (US, UK, India), INRIA (Paris), ...

```
// Dependent (inductive) types
```

```
type vector 'a : nat -> Type =
```

```
| Nil : vector 'a 0
```

```
| Cons : #n:nat -> 'a -> vector 'a n -> vector 'a (n + 1)
```

```
// Dependent (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, total) 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)
```



```
// Dependent (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, total) 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
```

```
let in_range_index (min:nat) (max:nat) = i:nat{min <= i /\ i <= max}  
  
val lkp : #a:Type -> #n:nat -> vector a n -> in_range_index 1 n -> Tot a  
let rec lkp #a #n xs i =  
  match xs with  
  | Cons x xs' -> if i = 1 then x else lkp xs' (i - 1)
```

```
// Dependent (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, total) 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
```

```
let in_range_index (min:nat) (max:nat) = i:nat{min <= i /\ i <= max}  
  
val lkp : #a:Type -> #n:nat -> vector a n -> in_range_index 1 n -> Tot a  
let rec lkp #a #n xs i =  
  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:Type) (#n:nat) (#m:nat) (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 (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, total) 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
```

```
let in_range_index (min:nat) (max:nat) = i:nat{min <= i /\ i <= max}  
  
val lkp : #a:Type -> #n:nat -> vector a n -> in_range_index 1 n -> Tot a  
let rec lkp #a #n xs i =  
  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:Type) (#n:nat) (#m:nat) (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)
```

```
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 rec lemma #a #n #m xs ys =  
  match xs with  
  | Nil -> ()  
  | Cons x xs' -> lemma xs' ys
```

// Dependent (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, total) 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

```
let in_range_index (min:nat) (max:nat) = i:nat{min <= i /\ i <= max}  
  
val lkp : #a:Type -> #n:nat -> vector a n -> in_range_index 1 n -> Tot a  
let rec lkp #a #n xs i =  
  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:Type) (#n:nat) (#m:nat) (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)

```
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 rec lemma #a #n #m xs ys =  
  match xs with  
  | Nil -> ()  
  | Cons x xs' -> lemma xs' ys
```

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

```
val take : #a:Type -> n:nat -> #m:nat -> zs:vector a m -> Pure (vector a n) (requires (n <= m))  
                                          (ensures (fun xs -> xs `is_prefix_of` zs))  
  
let rec take #a n #m 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 sum : i:nat -> n:nat{i <= n} -> Tot 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 h0 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} -> Tot 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 h0 r h1 -> sel h1 r = sum 1 n))
```

```
val sum_loop : i:nat -> n:nat -> r:ref nat -> ST unit (requires (fun h0 -> (1 <= i ∧ i <= n) ∧
                                                                    (i = 1 ==> sel h0 r = 0) ∧
                                                                    (i > 1 ==> sel h0 r = sum 1 (i - 1))))
                                   (ensures (fun h0 - 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} -> Tot 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))
  [SMPat (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 h0 -> 1 <= n))
  (ensures (fun h0 r h1 -> sel h1 r = sum 1 n))
```

```
val sum_loop : i:nat -> n:nat -> r:ref nat -> ST unit (requires (fun h0 -> (1 <= i ∧ i <= n) ∧
  (i = 1 ==> sel h0 r = 0) ∧
  (i > 1 ==> sel h0 r = sum 1 (i - 1))))
  (ensures (fun h0 - 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)
```


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** [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**

`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** ($v \in \text{get}()$)

- 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 $v \in \text{get}()$ “for free”

Monotonicity in programming

- **Programming** also relies on **monotonicity**, even if you don't realise it!
- Consider ML-style typed references $r:\text{ref } a$
 - 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^* 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, ...
 - 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
 - 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** **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
 - 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 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
 - 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 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
 - 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 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
 - 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** **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
 - 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**

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

where

$pre : state \rightarrow Type \quad post : state \rightarrow t \rightarrow state \rightarrow Type$

- ST is an abstract pre-postcondition refinement of

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

- The global state **actions** have types

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

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

- 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**

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

where

$pre : state \rightarrow Type \quad post : state \rightarrow t \rightarrow state \rightarrow Type$

- ST is an abstract pre-postcondition refinement of

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

- The global state **actions** have types

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

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

- 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**

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

where

$pre : state \rightarrow Type \quad post : state \rightarrow t \rightarrow state \rightarrow Type$

- ST is an abstract pre-postcondition refinement of

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

- The global state **actions** have types

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

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

- 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**

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

where

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

- ST is an abstract pre-postcondition refinement of

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

- The global state **actions** have types

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

- 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**

`MST #state #rel t (requires pre) (ensures post)`

- The `get` action is typed as in ST

`get : unit → MST state (requires (λ_.T))
(ensures (λ s0 s s1 . s0 = s = s1))`

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

`put : s:state → MST unit (requires (λ s0 . rel s0 s))
(ensures (λ _ s1 . s1 = s))`

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

`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**

`MST #state #rel t (requires pre) (ensures post)`

- The `get` action is typed as in ST

`get : unit → MST state (requires (λ_.T))
(ensures (λ s0 s s1 . s0 = s = s1))`

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

`put : s:state → MST unit (requires (λ s0 . rel s0 s))
(ensures (λ _ s1 . s1 = s))`

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

`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**

`MST #state #rel t (requires pre) (ensures post)`

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

`get : unit → MST state (requires (λ_.T))
(ensures (λ s0 s s1 . s0 = s = s1))`

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

`put : s:state → MST unit (requires (λ s0 . rel s0 s))
(ensures (λ _ s1 . s1 = s))`

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

`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**

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

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

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

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

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

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

$$\text{mst } t \stackrel{\text{def}}{=} \text{s}_0:\text{state} \rightarrow t * \text{s}_1:\text{state} \{ \text{rel } s_0 s_1 \}$$

New: Monotonic global state in F^*

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

MST #state **#red** t (requires pre) (ensures post)

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

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

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

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

- 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: 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**

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

- 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**

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

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**

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

- 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**

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

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**

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

- 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**

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

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**

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

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

[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

```
witness : p:(state  $\rightarrow$  Type0)  
          $\rightarrow$  MST unit (requires ( $\lambda s_0.$  p 'stable_from' s0))  
                   (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' s1))
```

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

```
witness : p:(state  $\rightarrow$  Type0)  
          $\rightarrow$  MST unit (requires ( $\lambda s_0.$  p 'stable_from' s0))  
                   (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' s1))
```

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

```
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

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

witness : $p:(\text{state} \rightarrow \text{Type}_0)$
→ MST unit (requires ($\lambda s_0. p \text{ 'stable_from' } s_0$))
 (ensures ($\lambda s_0 - s_1. s_0 = s_1 \wedge \text{witnessed } p$)))

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

recall : $p:(\text{state} \rightarrow \text{Type}_0)$
→ MST unit (requires ($\lambda _. \text{witnessed } p$))
 (ensures ($\lambda s_0 - s_1. s_0 = s_1 \wedge p \text{ 'stable_from' } s_1$)))

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**

```
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 (λ s. v ∈ s); c_p(); recall (λ s. v ∈ 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 (λ s. w ∈ s); [ ]; recall (λ s. w ∈ s); assert (w ∈ get())
```

- Monotonic counters** are analogous, by picking \mathbb{N} and \leq , e.g.,
`create 0; incr(); witness (λ c. c > 0); c_p(); recall (λ 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 { ∀ 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**)

```
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

```
type heap =
```

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

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 _) =
```

```
   $\forall r. \text{match } h_0\ r, h_1\ r \text{ with}$ 
```

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

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

```
    | 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: $\mathbb{N}\{\forall r. \text{ctr} \leq r \implies h\ r = \text{Unused}\}$   $\rightarrow$  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 -) =
```

```
 $\forall r. \text{match } h_0\ r, h_1\ r \text{ with}$ 
```

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

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

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


ML-style typed references (local state)

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

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

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

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

where

```
let contains (H h _) r a =  
  match h r with  
  | Used b _ → a = b  
  | Unused → ⊥
```

- 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**

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

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

```
abstract type ref a = r:ℕ{witnessed (λ h. contains h r a)}
```

where

```
let contains (H h _) r a =  
  match h r with  
  | Used b _ → a = b  
  | Unused → ⊥
```

- 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**

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

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

```
abstract type ref a = r:ℕ{witnessed (λ h. contains h r a)}
```

where

```
let contains (H h _) r a =  
  match h r with  
  | Used b _ → a = b  
  | Unused  → ⊥
```

- 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 ! (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 := (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** (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 !** ($r:\text{ref a}$) : **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 :=** ($r:\text{ref a}$) ($v:a$) : **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** ($\#a:\text{Type}$) ($v:a$) : **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 !** ($r:\text{ref a}$) : **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 :=** ($r:\text{ref a}$) ($v:a$) : **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** ($\#a:\text{Type}$) ($v:a$) : **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 !** ($r:\text{ref } a$) : **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 :=** ($r:\text{ref } a$) ($v:a$) : **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

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**

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

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` \rightarrow 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.

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** λ -calculus with **Tot** and **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 TT-lifting
- **Logical consistency** of pre-post cond. logic via cut elimination
- Hoare-style **total correctness** via SN, progress, and preservation

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

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

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

witnessed $W' \vdash$ *rel* $s \ s'$ and $W \subseteq W'$ and

witnessed $W' \vdash$ *post* $s \ v \ s'$

Glimpse of meta-theory

- A small **dependently typed** λ -calculus with **Tot** and **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 TT-lifting
- Logical consistency of pre-post cond. logic via cut elimination
- Hoare-style total correctness via SN, progress, and preservation

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

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

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

witnessed $W' \vdash$ *rel s s'* and $W \subseteq W'$ and

witnessed $W' \vdash$ *post s v s'*

Glimpse of meta-theory

- A small **dependently typed** λ -calculus with **Tot** and **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

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

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

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

witnessed $W' \vdash$ *rel s s'* and $W \subseteq W'$ and

witnessed $W' \vdash$ *post s v s'*

Glimpse of meta-theory

- A small **dependently typed** λ -calculus with **Tot** and **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

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

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

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

witnessed $W' \vdash$ *rel s s'* and $W \subseteq W'$ and

witnessed $W' \vdash$ *post s v s'*

Glimpse of meta-theory

- A small **dependently typed** λ -calculus with **Tot** and **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

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

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

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

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

witnessed $W' \vdash \textit{post } s \ v \ s'$

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?

D. Ahman, C. Fournet, C. Hrițcu, K. Maillard, A. Rastogi, N. Swamy.

Recalling a Witness: Foundations and Applications of Monotonic State

Proc. ACM Program. Lang., volume 2, issue POPL, article 65, 2018.