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

F* [fstar-lang.org]

- **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* [fstar-lang.org]

- **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), ...

```
// 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))
                                               [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 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

- $\texttt{Tot}$, $\texttt{Lemma}$, $\texttt{Pure}$, … are just some **effects** amongst many
    - $\texttt{Tot t}$
    - $\texttt{Lemma (requires pre}_{\texttt{Lemma}}\texttt{) (ensures post}_{\texttt{Lemma}}\texttt{)}$
    - $\texttt{Pure t (requires pre}_{\texttt{Pure}}\texttt{) (ensures post}_{\texttt{Pure}}\texttt{)}$
    - $\texttt{Div t (requires pre}_{\texttt{Div}}\texttt{) (ensures post}_{\texttt{Div}}\texttt{)}$
    - $\texttt{Exc t (requires pre}_{\texttt{Exc}}\texttt{) (ensures post}_{\texttt{Exc}}\texttt{)}$
    - $\texttt{ST t (requires pre}_{\texttt{ST}}\texttt{) (ensures post}_{\texttt{ST}}\texttt{)}$
    - …
- **Monad morphs.** $\texttt{Pure} \leadsto \{\texttt{Div}, \texttt{Exc}, \texttt{ST}\}$; $\texttt{Exc} \leadsto \texttt{STExc}$; …
- Systematically derived from **WP-calculi** [POPL 2017]

Outline

* F* overview
- Monotonic state by example
- Key ideas behind our general extension to Hoare-style logics
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- Glimpse of 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**

$$\texttt{insert v; complex\_procedure(); assert } (\texttt{v} \in \texttt{get()})$$

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

$$\{\lambda\,\texttt{s}\,.\,\texttt{v} \in \texttt{s}\}\ \texttt{complex\_procedure()}\ \{\lambda\,\texttt{s}\,.\,\texttt{v} \in \texttt{s}\}$$

  - likely that we have to **carry** $\lambda\,\texttt{s}\,.\,\texttt{v} \in \texttt{s}$ **through** the proof of c_p
  - **does not guarantee** that $\lambda\,\texttt{s}\,.\,\texttt{v} \in \texttt{s}$ holds at every point in c_p
  - **sensitive** to proving that c_p maintains $\lambda\,\texttt{s}\,.\,\texttt{w} \in \texttt{s}$ for some w

- However, if c_p **never removes**, then $\lambda\,\texttt{s}\,.\,\texttt{v} \in \texttt{s}$ is **stable**, and we would like the program logic to give us $\texttt{v} \in \texttt{get()}$ "**for free**"

Monotonicity in program verification

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

$$\texttt{insert v};\ \texttt{complex\_procedure}();\ \texttt{assert}\ (\texttt{v} \in \texttt{get}())$$

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

$$\{\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}\}\ \texttt{complex\_procedure}()\ \{\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}\}$$

  - likely that we have to **carry** $\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}$ **through** the proof of c_p
  - **does not guarantee** that $\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}$ holds at every point in c_p
  - **sensitive** to proving that c_p maintains $\lambda\, \texttt{s}\,.\,\texttt{w} \in \texttt{s}$ for some $\texttt{w}$

- However, if c_p **never removes**, then $\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}$ is **stable**, and we would like the program logic to give us $\texttt{v} \in \texttt{get}()$ "**for free**"

Monotonicity in program verification

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

$$\texttt{insert v; complex\_procedure(); assert } (\mathrm{v} \in \mathrm{get}())$$

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

$$\{\lambda\,\mathtt{s}\,.\,\mathtt{v} \in \mathtt{s}\}\ \texttt{complex\_procedure()}\ \{\lambda\,\mathtt{s}\,.\,\mathtt{v} \in \mathtt{s}\}$$

  - likely that we have to **carry** $\lambda\,\mathtt{s}\,.\,\mathtt{v} \in \mathtt{s}$ **through** the proof of c_p
  - **does not guarantee** that $\lambda\,\mathtt{s}\,.\,\mathtt{v} \in \mathtt{s}$ holds at every point in c_p
  - **sensitive** to proving that c_p maintains $\lambda\,\mathtt{s}\,.\,\mathtt{w} \in \mathtt{s}$ for some w

- However, if c_p **never removes**, then $\lambda\,\mathtt{s}\,.\,\mathtt{v} \in \mathtt{s}$ is **stable**, and we would like the program logic to give us $\mathrm{v} \in \mathrm{get}()$ "**for free**"

Monotonicity in program verification

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

$$\texttt{insert v; complex\_procedure(); assert } (\texttt{v} \in \texttt{get()})$$

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

$$\{\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}\}\ \texttt{complex\_procedure()}\ \{\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}\}$$

  - likely that we have to **carry** $\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}$ **through** the proof of c_p
  - **does not guarantee** that $\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}$ holds at every point in c_p
  - **sensitive** to proving that c_p maintains $\lambda\, \texttt{s}\,.\,\texttt{w} \in \texttt{s}$ for some w

- However, if c_p **never removes**, then $\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}$ is **stable**, and we would like the program logic to give us $\texttt{v} \in \texttt{get()}$ "**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**!
  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: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: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: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
* Monotonic state by example
* Key ideas behind our general extension to Hoare-style logics
* Accommodating monotonic state in F*
* Some examples of monotonic state at work
* Glimpse of 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

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

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

$$\forall\, \mathrm{s}\, \mathrm{s}'.\ \mathtt{p}\ \mathrm{s}\ \wedge\ \mathtt{rel}\ \mathrm{s}\ \mathrm{s}' \implies \mathtt{p}\ \mathrm{s}'$$

- **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

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

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

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

- **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
    $$\forall\, s\, e'\, s'.\ (e, s) \leadsto^{*} (e', s') \implies \mathtt{rel}\ s\ s'$$
  - a stateful predicate p is **stable** (wrt. `rel`) when
    $$\forall\, s\, s'.\ p\ s\ \wedge\ \mathtt{rel}\ s\ s' \implies p\ s'$$
- **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

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

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

$$\forall\, \mathrm{s}\, \mathrm{s}'.\ \mathrm{p}\ \mathrm{s}\ \wedge\ \texttt{rel}\ \mathrm{s}\ \mathrm{s}' \implies \mathrm{p}\ \mathrm{s}'$$

- **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
    $$\forall\, s\, e'\, s'.\ (e, s) \leadsto^* (e', s') \implies \mathtt{rel}\ s\ s'$$
  - a stateful predicate `p` is **stable** (wrt. `rel`) when
    $$\forall\, s\, s'.\ \mathtt{p}\ s\ \wedge\ \mathtt{rel}\ s\ s' \implies \mathtt{p}\ s'$$
- **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
    $$\forall\, \mathtt{s}\, \mathtt{e}'\, \mathtt{s}'.\ (\mathtt{e}, \mathtt{s}) \leadsto^* (\mathtt{e}', \mathtt{s}') \implies \mathtt{rel}\ \mathtt{s}\ \mathtt{s}'$$
  - a stateful predicate `p` is **stable** (wrt. `rel`) when
    $$\forall\, \mathtt{s}\, \mathtt{s}'.\ \mathtt{p}\ \mathtt{s} \wedge \mathtt{rel}\ \mathtt{s}\ \mathtt{s}' \implies \mathtt{p}\ \mathtt{s}'$$
- **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
    $$\forall\, \mathrm{s}\, \mathrm{e}'\, \mathrm{s}'.\ (\mathrm{e}, \mathrm{s}) \leadsto^* (\mathrm{e}', \mathrm{s}') \implies \mathtt{rel}\ \mathrm{s}\ \mathrm{s}'$$
  - a stateful predicate p is **stable** (wrt. `rel`) when
    $$\forall\, \mathrm{s}\, \mathrm{s}'.\ \mathtt{p}\ \mathrm{s}\ \wedge\ \mathtt{rel}\ \mathrm{s}\ \mathrm{s}' \implies \mathtt{p}\ \mathrm{s}'$$
- **Our solution:** extend Hoare-style program logics (e.g., F*) with
  - a means to **witness** the validity of p s in some state s
  - a means for turning a p into a **state-independent proposition**
  - a means to **recall** the validity of p s′ in any future state s′
- Provides a **unifying account** of the existing *ad hoc* uses in F*

Outline

* F* overview

- Monotonic state by example
- Key ideas behind our general extension to Hoare-style logics
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- Glimpse of 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**

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

  where

$$\mathtt{pre} : \mathtt{state} \to \mathrm{Type} \qquad \mathtt{post} : \mathtt{state} \to \mathtt{t} \to \mathtt{state} \to \mathrm{Type}$$

- ST is an abstract pre-postcondition refinement of

$$\mathtt{st}\ \mathtt{t} \stackrel{\mathsf{def}}{=} \mathtt{state} \to \mathtt{t} * \mathtt{state}$$

- The global state **actions** have types

$$\mathtt{get} : \mathtt{unit} \to \mathtt{ST}\ \mathtt{state}\ (\mathtt{requires}\ (\lambda\,\_\,.\top))\ (\mathtt{ensures}\ (\lambda\,\mathtt{s}_0\ \mathtt{s}\ \mathtt{s}_1\,.\,\mathtt{s}_0 = \mathtt{s} = \mathtt{s}_1))$$

$$\mathtt{put} : \mathtt{s}{:}\mathtt{state} \to \mathtt{ST}\ \mathtt{unit}\ (\mathtt{requires}\ (\lambda\,\_\,.\top))\ (\mathtt{ensures}\ (\lambda\,\_\,\_\ \mathtt{s}_1\,.\,\mathtt{s}_1 = \mathtt{s}))$$

- **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**

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

  where

$$\texttt{pre} : \texttt{state} \to \texttt{Type} \qquad \texttt{post} : \texttt{state} \to \texttt{t} \to \texttt{state} \to \texttt{Type}$$

- ST is an abstract pre-postcondition refinement of

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

- The global state **actions** have types

$$\texttt{get} : \texttt{unit} \to \texttt{ST state}\ (\texttt{requires}\ (\lambda\,\_\,.\top))\ (\texttt{ensures}\ (\lambda\,\texttt{s}_0\ \texttt{s}\ \texttt{s}_1\,.\,\texttt{s}_0 = \texttt{s} = \texttt{s}_1))$$

$$\texttt{put} : \texttt{s}{:}\texttt{state} \to \texttt{ST unit}\ (\texttt{requires}\ (\lambda\,\_\,.\top))\ (\texttt{ensures}\ (\lambda\,\_\,\_\,\texttt{s}_1\,.\,\texttt{s}_1 = \texttt{s}))$$

- **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**

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

  where

$$\mathtt{pre} : \mathtt{state} \to \mathtt{Type} \qquad \mathtt{post} : \mathtt{state} \to \mathtt{t} \to \mathtt{state} \to \mathtt{Type}$$

- ST is an abstract pre-postcondition refinement of

$$\mathtt{st}\ \mathtt{t} \stackrel{\mathsf{def}}{=} \mathtt{state} \to \mathtt{t} * \mathtt{state}$$

- The global state **actions** have types

$$\mathtt{get} : \mathtt{unit} \to \mathtt{ST}\ \mathtt{state}\ (\mathtt{requires}\ (\lambda\,\_.\top))\ (\mathtt{ensures}\ (\lambda\,\mathtt{s}_0\ \mathtt{s}\ \mathtt{s}_1\,.\,\mathtt{s}_0 = \mathtt{s} = \mathtt{s}_1))$$

$$\mathtt{put} : \mathtt{s}{:}\mathtt{state} \to \mathtt{ST}\ \mathtt{unit}\ (\mathtt{requires}\ (\lambda\,\_.\top))\ (\mathtt{ensures}\ (\lambda\,\_\,\_\,\mathtt{s}_1\,.\,\mathtt{s}_1 = \mathtt{s}))$$

- **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**

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

  where

$$\mathtt{pre} : \mathtt{state} \to \mathtt{Type} \qquad \mathtt{post} : \mathtt{state} \to \mathtt{t} \to \mathtt{state} \to \mathtt{Type}$$

- ST is an abstract pre-postcondition refinement of

$$\mathtt{st}\ \mathtt{t} \stackrel{\mathsf{def}}{=} \mathtt{state} \to \mathtt{t} * \mathtt{state}$$

- The global state **actions** have types

$$\mathtt{get} : \mathtt{unit} \to \mathtt{ST}\ \mathtt{state}\ (\mathtt{requires}\ (\lambda\, \_ . \top))\ (\mathtt{ensures}\ (\lambda\, \mathtt{s}_0\ \mathtt{s}\ \mathtt{s}_1 \,.\, \mathtt{s}_0 = \mathtt{s} = \mathtt{s}_1))$$

$$\mathtt{put} : \mathtt{s}{:}\mathtt{state} \to \mathtt{ST}\ \mathtt{unit}\ (\mathtt{requires}\ (\lambda\, \_ . \top))\ (\mathtt{ensures}\ (\lambda\, \_\ \_\ \mathtt{s}_1 \,.\, \mathtt{s}_1 = \mathtt{s}))$$

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

New: Monotonic global state in F*

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

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

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

$$\mathtt{get} : \mathtt{unit} \to \mathtt{MST}\ \mathtt{state}\ (\mathtt{requires}\ (\lambda\, \_ . \top))\ (\mathtt{ensures}\ (\lambda\, \mathtt{s}_0\, \mathtt{s}\, \mathtt{s}_1 \,.\, \mathtt{s}_0 = \mathtt{s} = \mathtt{s}_1))$$

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

$$\mathtt{put} : \mathtt{s}{:}\mathtt{state} \to \mathtt{MST}\ \mathtt{unit}\ (\mathtt{requires}\ (\lambda\, \mathtt{s}_0 \,.\, \mathtt{rel}\ \mathtt{s}_0\ \mathtt{s}))\ (\mathtt{ensures}\ (\lambda\, \_\, \_\, \mathtt{s}_1 \,.\, \mathtt{s}_1 = \mathtt{s}))$$

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

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

New: Monotonic global state in F*

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

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

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

$$\begin{aligned}\texttt{get} : \texttt{unit} \to \texttt{MST}\ \texttt{state}\ &(\texttt{requires}\ (\lambda\ \_ .\top))\\ &(\texttt{ensures}\ (\lambda\ \texttt{s}_0\ \texttt{s}\ \texttt{s}_1\ .\ \texttt{s}_0 = \texttt{s} = \texttt{s}_1))\end{aligned}$$

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

$$\begin{aligned}\texttt{put} : \texttt{s}{:}\texttt{state} \to \texttt{MST}\ \texttt{unit}\ &(\texttt{requires}\ (\lambda\ \texttt{s}_0\ .\ \texttt{rel}\ \texttt{s}_0\ \texttt{s}))\\ &(\texttt{ensures}\ (\lambda\ \_\ \_\ \texttt{s}_1\ .\ \texttt{s}_1 = \texttt{s}))\end{aligned}$$

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

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

New: Monotonic global state in F*

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

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

- The **get** action is typed as in $\mathtt{ST}$

$$\begin{aligned}\mathtt{get} : \mathtt{unit} \to \mathtt{MST}\ \mathtt{state}\ &(\mathtt{requires}\ (\lambda\, \_ . \top))\\ &(\mathtt{ensures}\ (\lambda\, \mathtt{s}_0\ \mathtt{s}\ \mathtt{s}_1 \,.\, \mathtt{s}_0 = \mathtt{s} = \mathtt{s}_1))\end{aligned}$$

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

$$\begin{aligned}\mathtt{put} : \mathtt{s}{:}\mathtt{state} \to \mathtt{MST}\ \mathtt{unit}\ &(\mathtt{requires}\ (\lambda\, \mathtt{s}_0 \,.\, \mathtt{rel}\ \mathtt{s}_0\ \mathtt{s}))\\ &(\mathtt{ensures}\ (\lambda\, \_\ \_\ \mathtt{s}_1 \,.\, \mathtt{s}_1 = \mathtt{s}))\end{aligned}$$

- So intuitively, $\mathtt{MST}$ is an **abstract** pre-postcondition refinement of

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

New: Monotonic global state in F*

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

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

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

$$\mathtt{get} : \mathtt{unit} \rightarrow \mathtt{MST}\ \mathtt{state}\ (\mathtt{requires}\ (\lambda\, \_ . \top))\ (\mathtt{ensures}\ (\lambda\, \mathtt{s}_0\, \mathtt{s}\, \mathtt{s}_1 \,.\, \mathtt{s}_0 = \mathtt{s} = \mathtt{s}_1))$$

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

$$\mathtt{put} : \mathtt{s}{:}\mathtt{state} \rightarrow \mathtt{MST}\ \mathtt{unit}\ (\mathtt{requires}\ (\lambda\, \mathtt{s}_0 \,.\, \mathtt{rel}\ \mathtt{s}_0\ \mathtt{s}))\ (\mathtt{ensures}\ (\lambda\, \_\, \_\, \mathtt{s}_1 \,.\, \mathtt{s}_1 = \mathtt{s}))$$

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

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

New: Monotonic global state in F*

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

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

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

$$\begin{aligned} \mathtt{get} : \mathtt{unit} \to \mathtt{MST}\ \mathtt{state}\ & (\mathtt{requires}\ (\lambda\, \_ . \top)) \\ & (\mathtt{ensures}\ (\lambda\, \mathtt{s}_0\, \mathtt{s}\, \mathtt{s}_1 \,.\, \mathtt{s}_0 = \mathtt{s} = \mathtt{s}_1)) \end{aligned}$$

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

$$\begin{aligned} \mathtt{put} : \mathtt{s}{:}\mathtt{state} \to \mathtt{MST}\ \mathtt{unit}\ & (\mathtt{requires}\ (\lambda\, \mathtt{s}_0 \,.\, \mathtt{rel}\ \mathtt{s}_0\ \mathtt{s})) \\ & (\mathtt{ensures}\ (\lambda\, \_\, \_\, \mathtt{s}_1 \,.\, \mathtt{s}_1 = \mathtt{s})) \end{aligned}$$

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

$$\mathtt{mst}\ \mathtt{t} \stackrel{\mathsf{def}}{=} \mathtt{s}_0{:}\mathtt{state} \to \mathtt{t} * \mathtt{s}_1{:}\mathtt{state}\{\mathtt{rel}\ \mathtt{s}_0\ \mathtt{s}_1\}$$

New: Recalling a Witness

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

$$\texttt{witnessed} : (\texttt{state} \to \mathrm{Type}) \to \mathrm{Type}$$

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

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

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

$$[\![\texttt{witnessed p}]\!](\texttt{s}) \stackrel{\text{def}}{=} \texttt{p}\ \text{`}\texttt{stable\_from}\text{`}\ \texttt{s}$$
$$\stackrel{\text{def}}{=} \forall\, \texttt{s}'\,.\, \texttt{rel s s}' \implies [\![\texttt{p s}']\!](\texttt{s})$$

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

New: Recalling a Witness

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

$$\mathtt{witnessed} : (\mathtt{state} \to \mathtt{Type}) \to \mathtt{Type}$$

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

$$\begin{aligned}\mathtt{wk} : \mathtt{p{,}q{:}}(\mathtt{state} \to \mathtt{Type}) \to \mathtt{Lemma}\ &(\mathtt{requires}\ (\forall\, \mathtt{s}\,.\, \mathtt{p\ s} \implies \mathtt{q\ s}))\\ &(\mathtt{ensures}\ (\mathtt{witnessed\ p} \implies \mathtt{witnessed\ q}))\end{aligned}$$

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

$$\begin{aligned}[\![\mathtt{witnessed\ p}]\!](\mathtt{s}) &\stackrel{\text{def}}{=} \mathtt{p}\ \text{`}\mathtt{stable\_from}\text{`}\ \mathtt{s}\\ &\stackrel{\text{def}}{=} \forall\, \mathtt{s}'\,.\, \mathtt{rel\ s\ s}' \implies [\![\mathtt{p\ s}']\!](\mathtt{s})\end{aligned}$$

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

New: Recalling a Witness

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

$$\mathtt{witnessed} : (\mathtt{state} \to \mathtt{Type}) \to \mathtt{Type}$$

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

$$\begin{aligned} \mathtt{wk} : \mathtt{p},\mathtt{q}{:}(\mathtt{state} \to \mathtt{Type}) \to \mathtt{Lemma}\ &(\mathtt{requires}\ (\forall\, \mathtt{s}\,.\,\mathtt{p}\ \mathtt{s} \implies \mathtt{q}\ \mathtt{s})) \\ &(\mathtt{ensures}\ (\mathtt{witnessed}\ \mathtt{p} \implies \mathtt{witnessed}\ \mathtt{q})) \end{aligned}$$

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

$$\begin{aligned} [\![\mathtt{witnessed}\ \mathtt{p}]\!](\mathtt{s}) \ &\stackrel{\text{def}}{=}\ \mathtt{p}\ \text{`}\mathtt{stable\_from}\text{`}\ \mathtt{s} \\ &\stackrel{\text{def}}{=}\ \forall\, \mathtt{s}'\,.\,\mathtt{rel}\ \mathtt{s}\ \mathtt{s}' \implies [\![\mathtt{p}\ \mathtt{s}']\!](\mathtt{s}) \end{aligned}$$

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

New: Recalling a Witness

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

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

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

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

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

$$\begin{aligned} [\![\texttt{witnessed p}]\!](\texttt{s}) &\stackrel{\text{def}}{=} \texttt{p}\ \text{`}\texttt{stable\_from}\text{`}\ \texttt{s} \\ &\stackrel{\text{def}}{=} \forall\, \texttt{s}'\,.\,\texttt{rel s s}' \implies [\![\texttt{p s}']\!](\texttt{s}) \end{aligned}$$

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

New: Recalling a Witness

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

$$\texttt{witnessed} : (\texttt{state} \to \texttt{Type}) \to \texttt{Type}$$

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

$$\begin{aligned}\texttt{wk} : \texttt{p},\texttt{q}{:}(\texttt{state} \to \texttt{Type}) \to \texttt{Lemma}\ &(\texttt{requires}\ (\forall\, \texttt{s}\,.\,\texttt{p s} \implies \texttt{q s}))\\ &(\texttt{ensures}\ (\texttt{witnessed p} \implies \texttt{witnessed q}))\end{aligned}$$

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

$$\begin{aligned}[\![\texttt{witnessed p}]\!](\texttt{s}) &\stackrel{\text{def}}{=} \texttt{p}\ \text{`}\texttt{stable\_from}\text{`}\ \texttt{s}\\ &\stackrel{\text{def}}{=} \forall\, \texttt{s}'\,.\,\texttt{rel s s}' \implies [\![\texttt{p s}']\!](\texttt{s})\end{aligned}$$

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

New: Recalling a Witness

- … Hoare-style logics are essentially **world/state-indexed**, so

- we include a **stateful introduction rule** for `witnessed`

$$\begin{aligned}\texttt{witness} \;:\; & p{:}(\texttt{state} \rightarrow \texttt{Type}_0)\\ & \rightarrow \mathsf{MST}\ \texttt{unit}\ (\texttt{requires}\ (\lambda\, s_0\, .\, p\ \text{`}\texttt{stable\_from}\text{'}\ s_0))\\ & \qquad (\texttt{ensures}\ (\lambda\, s_0\, \_\, s_1\, .\, s_0 = s_1 \wedge \texttt{witnessed}\ p))\end{aligned}$$

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

$$\begin{aligned}\texttt{recall} \;:\; & p{:}(\texttt{state} \rightarrow \texttt{Type}_0)\\ & \rightarrow \mathsf{MST}\ \texttt{unit}\ (\texttt{requires}\ (\lambda\, \_\, .\, \texttt{witnessed}\ p))\\ & \qquad (\texttt{ensures}\ (\lambda\, s_0\, \_\, s_1\, .\, s_0 = s_1 \wedge p\ \text{`}\texttt{stable\_from}\text{'}\ s_1))\end{aligned}$$

New: Recalling a Witness

- … Hoare-style logics are essentially **world/state-indexed**, so

- we include a **stateful introduction rule** for `witnessed`

$$\begin{array}{lll}\texttt{witness} & : & \texttt{p}{:}(\texttt{state} \to \texttt{Type}_0) \\ & & \to \texttt{MST}\ \texttt{unit}\ (\texttt{requires}\ (\lambda\, \texttt{s}_0\, .\, \texttt{p}\ \text{`}\texttt{stable\_from}\text{`}\ \texttt{s}_0)) \\ & & \qquad\qquad (\texttt{ensures}\ (\lambda\, \texttt{s}_0\ \_\ \texttt{s}_1\, .\, \texttt{s}_0 = \texttt{s}_1 \wedge \texttt{witnessed}\ \texttt{p}))\end{array}$$

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

$$\begin{array}{lll}\texttt{recall} & : & \texttt{p}{:}(\texttt{state} \to \texttt{Type}_0) \\ & & \to \texttt{MST}\ \texttt{unit}\ (\texttt{requires}\ (\lambda\, \_\, .\, \texttt{witnessed}\ \texttt{p})) \\ & & \qquad\qquad (\texttt{ensures}\ (\lambda\, \texttt{s}_0\ \_\ \texttt{s}_1\, .\, \texttt{s}_0 = \texttt{s}_1 \wedge \texttt{p}\ \text{`}\texttt{stable\_from}\text{`}\ \texttt{s}_1))\end{array}$$

New: Recalling a Witness

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

$$\begin{array}{lll} \texttt{witness} & : & \texttt{p}{:}(\texttt{state} \to \mathrm{Type}_0) \\ & & \to \texttt{MST}\ \texttt{unit}\ (\texttt{requires}\ (\lambda\, \texttt{s}_0\, .\, \texttt{p}\ \text{`}\texttt{stable\_from}\text{`}\ \texttt{s}_0)) \\ & & \qquad\qquad (\texttt{ensures}\ (\lambda\, \texttt{s}_0\ \_\ \texttt{s}_1\, .\, \texttt{s}_0 = \texttt{s}_1 \wedge \texttt{witnessed}\ \texttt{p})) \end{array}$$

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

$$\begin{array}{lll} \texttt{recall} & : & \texttt{p}{:}(\texttt{state} \to \mathrm{Type}_0) \\ & & \to \texttt{MST}\ \texttt{unit}\ (\texttt{requires}\ (\lambda\, \_\, .\, \texttt{witnessed}\ \texttt{p})) \\ & & \qquad\qquad (\texttt{ensures}\ (\lambda\, \texttt{s}_0\ \_\ \texttt{s}_1\, .\, \texttt{s}_0 = \texttt{s}_1 \wedge \texttt{p}\ \text{`}\texttt{stable\_from}\text{`}\ \texttt{s}_1)) \end{array}$$

Outline

* F* overview
* Monotonic state by example
* Key ideas behind our general extension to Hoare-style logics
* Accommodating monotonic state in F*
* Some examples of monotonic state at work
* Glimpse of 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**

$$\texttt{insert v; complex\_procedure(); assert } (\mathrm{v} \in \mathrm{get}())$$

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

$$\texttt{insert v; witness } (\lambda\, \mathrm{s}\, .\, \mathrm{v} \in \mathrm{s})\texttt{; c\_p(); recall } (\lambda\, \mathrm{s}\, .\, \mathrm{v} \in \mathrm{s})\texttt{; assert } (\mathrm{v} \in \mathrm{get}())$$

  - For **any other** `w`, wrapping

$$\texttt{insert w; [ ]; assert } (\mathrm{w} \in \mathrm{get}())$$

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

$$\texttt{insert w; witness } (\lambda\, \mathrm{s}\, .\, \mathrm{w} \in \mathrm{s})\texttt{; [ ]; recall } (\lambda\, \mathrm{s}\, .\, \mathrm{w} \in \mathrm{s})\texttt{; assert } (\mathrm{w} \in \mathrm{get}())$$

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

$$\texttt{create 0; incr(); witness } (\lambda\, \mathrm{c}\, .\, \mathrm{c} > 0)\texttt{; c\_p(); recall } (\lambda\, \mathrm{c}\, .\, \mathrm{c} > 0)$$

The motivating example revisited

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

$$\texttt{insert v};\ \texttt{complex\_procedure}();\ \texttt{assert}\ (\texttt{v} \in \texttt{get}())$$

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

$$\texttt{insert v};\ \texttt{witness}\ (\lambda\, \texttt{s}\,.\, \texttt{v} \in \texttt{s});\ \texttt{c\_p}();\ \texttt{recall}\ (\lambda\, \texttt{s}\,.\, \texttt{v} \in \texttt{s});\ \texttt{assert}\ (\texttt{v} \in \texttt{get}())$$

  - For **any other** `w`, wrapping

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

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

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

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

$$\texttt{create 0};\ \texttt{incr}();\ \texttt{witness}\ (\lambda\, \texttt{c}\,.\, \texttt{c} > 0);\ \texttt{c\_p}();\ \texttt{recall}\ (\lambda\, \texttt{c}\,.\, \texttt{c} > 0)$$

The motivating example revisited

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

$$\texttt{insert v; complex\_procedure(); assert } (v \in \text{get}())$$

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

$$\texttt{insert v; witness } (\lambda\, s\, .\, v \in s)\texttt{; c\_p(); recall } (\lambda\, s\, .\, v \in s)\texttt{; assert } (v \in \text{get}())$$

  - For **any other** w, wrapping

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

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

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

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

$$\texttt{create 0; incr(); witness } (\lambda\, c\, .\, c > 0)\texttt{; c\_p(); recall } (\lambda\, c\, .\, c > 0)$$

The motivating example revisited

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

$$\texttt{insert v};\ \texttt{complex\_procedure}();\ \texttt{assert}\ (\texttt{v} \in \texttt{get}())$$

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

$$\texttt{insert v};\ \texttt{witness}\ (\lambda\,\texttt{s}\,.\,\texttt{v} \in \texttt{s});\ \texttt{c\_p}();\ \texttt{recall}\ (\lambda\,\texttt{s}\,.\,\texttt{v} \in \texttt{s});\ \texttt{assert}\ (\texttt{v} \in \texttt{get}())$$

  - For **any other** `w`, wrapping

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

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

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

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

$$\texttt{create 0};\ \texttt{incr}();\ \texttt{witness}\ (\lambda\,\texttt{c}\,.\,\texttt{c} > 0);\ \texttt{c\_p}();\ \texttt{recall}\ (\lambda\,\texttt{c}\,.\,\texttt{c} > 0)$$

The motivating example revisited

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

$$\texttt{insert v; complex\_procedure(); assert } (\mathrm{v} \in \mathrm{get}())$$

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

$$\texttt{insert v; witness } (\lambda\, \mathrm{s}\,.\, \mathrm{v} \in \mathrm{s})\texttt{; c\_p(); recall } (\lambda\, \mathrm{s}\,.\, \mathrm{v} \in \mathrm{s})\texttt{; assert } (\mathrm{v} \in \mathrm{get}())$$

  - For **any other** `w`, wrapping

$$\texttt{insert w; [ ]; assert } (\mathrm{w} \in \mathrm{get}())$$

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

$$\texttt{insert w; witness } (\lambda\, \mathrm{s}\,.\, \mathrm{w} \in \mathrm{s})\texttt{; [ ]; recall } (\lambda\, \mathrm{s}\,.\, \mathrm{w} \in \mathrm{s})\texttt{; assert } (\mathrm{w} \in \mathrm{get}())$$

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

$$\texttt{create 0; incr(); witness } (\lambda\, \mathrm{c}\,.\, \mathrm{c} > 0)\texttt{; c\_p(); recall } (\lambda\, \mathrm{c}\,.\, \mathrm{c} > 0)$$

ML-style typed references (local state)

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

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

where

```
type cell =
  | Unused : cell
  | Used : a:Type → v:a → cell
```

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

```
let heap_inclusion (H h₀ _) (H h₁ _) = ∀id. match h₀ id, h₁ id with
  | Used a _, Used b _ → a = b
  | Unused, Used _ _ → ⊤
  | Unused, Unused → ⊤
  | Used _ _, Unused → ⊥
```

ML-style typed references (local state)

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

$$\texttt{type heap} =$$
$$\mid \texttt{H} : h{:}(\mathbb{N} \to \texttt{cell}) \to ctr{:}\mathbb{N}\{\forall\, n\,.\, ctr \leq n \implies h\; n = \texttt{Unused}\} \to \texttt{heap}$$

where

$$\texttt{type cell} =$$
$$\mid \texttt{Unused} : \texttt{cell}$$
$$\mid \texttt{Used} : a{:}\texttt{Type} \to v{:}a \to \texttt{cell}$$

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

$$\texttt{let heap\_inclusion}\ (\texttt{H}\ h_0\ \_)\ (\texttt{H}\ h_1\ \_) = \forall\, id\,.\ \texttt{match}\ h_0\ id\,, h_1\ id\ \texttt{with}$$
$$\mid \texttt{Used}\ a\ \_\,, \texttt{Used}\ b\ \_\ \to\ a = b$$
$$\mid \texttt{Unused}\,, \texttt{Used}\ \_\ \_\ \to\ \top$$
$$\mid \texttt{Unused}\,, \texttt{Unused}\ \to\ \top$$
$$\mid \texttt{Used}\ \_\ \_\,, \texttt{Unused}\ \to\ \bot$$

ML-style typed references (local state)

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

  $\mathtt{type}\ \mathtt{heap} =$

  $\quad |\ \mathtt{H} : \mathtt{h}{:}(\mathbb{N} \to \mathtt{cell}) \to \mathtt{ctr}{:}\mathbb{N}\{\forall\, \mathtt{n}\,.\, \mathtt{ctr} \leq \mathtt{n} \implies \mathtt{h}\ \mathtt{n} = \mathtt{Unused}\} \to \mathtt{heap}$

  where

  $\mathtt{type}\ \mathtt{cell} =$

  $\quad |\ \mathtt{Unused} : \mathtt{cell}$

  $\quad |\ \mathtt{Used} : \mathtt{a}{:}\mathrm{Type} \to \mathtt{v}{:}\mathtt{a} \to \mathtt{cell}$

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

  $\mathtt{let}\ \mathtt{heap\_inclusion}\ (\mathtt{H}\ \mathtt{h}_0\ \_)\ (\mathtt{H}\ \mathtt{h}_1\ \_) = \forall\, \mathtt{id}\,.\, \mathtt{match}\ \mathtt{h}_0\ \mathtt{id}\,, \mathtt{h}_1\ \mathtt{id}\ \mathtt{with}$

  $\quad |\ \mathtt{Used}\ \mathtt{a}\ \_\,, \mathtt{Used}\ \mathtt{b}\ \_\ \to\ \mathtt{a} = \mathtt{b}$

  $\quad |\ \mathtt{Unused}\,, \mathtt{Used}\ \_\ \_\ \to\ \top$

  $\quad |\ \mathtt{Unused}\,, \mathtt{Unused}\ \to\ \top$

  $\quad |\ \mathtt{Used}\ \_\ \_\,, \mathtt{Unused}\ \to\ \bot$

ML-style typed references (local state)

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

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

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

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

where

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

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

ML-style typed references (local state)

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

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

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

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

where

```
let contains (H h _) id a =
  match h id 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 new **local state effect**

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

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

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

  where

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

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

Adding untyped and monotonic references

- **Untyped references** (`uref`) with strong updates
  - Used `heap` cells are extended with **tags**

    where
    $$\mid \texttt{Used} : \mathrm{a{:}Type} \rightarrow \mathrm{v{:}a} \rightarrow \mathrm{t{:}}\texttt{tag} \rightarrow \texttt{cell}$$
    $$\texttt{type}\ \texttt{tag}\ =\ \texttt{Typed} : \texttt{tag}\ \mid\ \texttt{Untyped} : \texttt{tag}$$
  - actions corresponding to `uref`s have **weaker types** than for `ref`s

- **Monotonic references** (`mref a rel`)
  - Used `heap` cells are extended with **typed tags**

    where
    $$\mid \texttt{Used} : \mathrm{a{:}Type} \rightarrow \mathrm{v{:}a} \rightarrow \mathrm{t{:}}\texttt{tag a} \rightarrow \texttt{cell}$$
    $$\texttt{type}\ \texttt{tag a}\ =\ \texttt{Typed} : \texttt{rel{:}preorder a} \rightarrow \texttt{tag a}\ \mid\ \texttt{Untyped} : \texttt{tag a}$$
  - `mref`s 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**

    $$\mid \ \mathtt{Used} : \mathtt{a{:}Type} \to \mathtt{v{:}a} \to \mathtt{t{:}tag} \to \mathtt{cell}$$

    where

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

  - actions corresponding to `uref`s have **weaker types** than for `ref`s

- **Monotonic references** (`mref a rel`)
  - Used heap cells are extended with **typed tags**

    $$\mid \ \mathtt{Used} : \mathtt{a{:}Type} \to \mathtt{v{:}a} \to \mathtt{t{:}tag}\ \mathtt{a} \to \mathtt{cell}$$

    where

    $$\mathtt{type}\ \mathtt{tag}\ \mathtt{a} \ = \ \mathtt{Typed} : \mathtt{rel{:}preorder}\ \mathtt{a} \to \mathtt{tag}\ \mathtt{a} \ \mid \ \mathtt{Untyped} : \mathtt{tag}\ \mathtt{a}$$

  - `mref`s 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**

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

    where

    $$\mathtt{type\ tag} \ = \ \mathtt{Typed} : \mathtt{tag} \ | \ \mathtt{Untyped} : \mathtt{tag}$$

  - actions corresponding to `uref`s have **weaker types** than for `ref`s

- **Monotonic references** (`mref a rel`)
  - Used heap cells are extended with **typed tags**

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

    where

    $$\mathtt{type\ tag\ a} \ = \ \mathtt{Typed} : \mathtt{rel{:}preorder\ a} \rightarrow \mathtt{tag\ a} \ | \ \mathtt{Untyped} : \mathtt{tag\ a}$$

  - `mref`s 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**

    $$| \ \texttt{Used} : \mathsf{a}{:}\mathsf{Type} \to \texttt{v}{:}\mathsf{a} \to \texttt{t:tag} \to \texttt{cell}$$

    where

    $$\texttt{type tag} \ = \ \texttt{Typed} : \texttt{tag} \ | \ \texttt{Untyped} : \texttt{tag}$$

  - actions corresponding to `uref`s have **weaker types** than for `ref`s

- **Monotonic references** (`mref a rel`)
  - Used heap cells are extended with **typed tags**

    $$| \ \texttt{Used} : \mathsf{a}{:}\mathsf{Type} \to \texttt{v}{:}\mathsf{a} \to \texttt{t:tag a} \to \texttt{cell}$$

    where

    $$\texttt{type tag a} \ = \ \texttt{Typed} : \texttt{rel:preorder a} \to \texttt{tag a} \ | \ \texttt{Untyped} : \texttt{tag a}$$

  - `mref`s provide **more flexibility** with ref.-wise monotonicity

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

Outline

- F* overview
- Monotonic state by example
- Key ideas behind our general extension to Hoare-style logics
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- Glimpse of 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
- **Logical consistency** shown via cut elimination
- Using an **instrumented operational semantics**, where

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

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

- **Strong normalisation** shown via type-erasure and $\top\top$-lifting
- Hoare-style **total correctness** via SN, progress, and preservation

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

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

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

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

$\texttt{witnessed}\ W' \vdash \mathit{post}\ s\ v\ s'$

Glimpse of meta-theory

- A small **dependently typed** λ**-calculus** with `Tot` and `MST` effects
- **Logical consistency** shown via cut elimination
- Using an **instrumented operational semantics**, where

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

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

- **Strong normalisation** shown via type-erasure and $\top\top$-lifting
- Hoare-style **total correctness** via SN, progress, and preservation

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

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

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

$\mathtt{witnessed}\, W' \vdash \mathtt{rel}\ s\ s'$ **and** $W \subseteq W'$ **and**

$\mathtt{witnessed}\, W' \vdash \mathit{post}\ s\ v\ s'$

Glimpse of meta-theory

- A small **dependently typed λ-calculus** with `Tot` and `MST` effects
- **Logical consistency** shown via cut elimination
- Using an **instrumented operational semantics**, where

$$
\begin{array}{rcl}
(\texttt{witness}\ p\,,\, s\,,\, W) & \rightsquigarrow & (\texttt{return}\ ()\,,\, s\,,\, W \cup \{p\}) \\
(\texttt{recall}\ p\,,\, s\,,\, W) & \rightsquigarrow & (\texttt{return}\ ()\,,\, s\,,\, W)
\end{array}
$$

- **Strong normalisation** shown via type-erasure and $\top\top$-lifting
- Hoare-style **total correctness** via SN, progress, and preservation

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

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

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

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

$\texttt{witnessed}\ W' \vdash \mathit{post}\ s\ v\ s'$

Glimpse of meta-theory

- A small **dependently typed λ-calculus** with `Tot` and `MST` effects
- **Logical consistency** shown via cut elimination
- Using an **instrumented operational semantics**, where

$$\begin{array}{rcl}(\texttt{witness}\ p\,,\, s\,,\, W) & \rightsquigarrow & (\texttt{return}\ ()\,,\, s\,,\, W \cup \{p\}) \\ (\texttt{recall}\ p\,,\, s\,,\, W) & \rightsquigarrow & (\texttt{return}\ ()\,,\, s\,,\, W)\end{array}$$

- **Strong normalisation** shown via type-erasure and $\top\top$-lifting
- Hoare-style **total correctness** via SN, progress, and preservation

$$\begin{array}{ll}\textbf{if} & \vdash e : \texttt{MST}\ t\ \mathit{pre}\ \mathit{post} \quad \textbf{and} \\ & \vdash (s, W)\ \mathsf{wf} \quad \textbf{and} \quad \texttt{witnessed}\ W \vdash \mathit{pre}\ s \\ \textbf{then} & (e, s, W) \rightsquigarrow^* (\texttt{return}\ v, s', W') \quad \textbf{and} \quad \vdash v : t \quad \textbf{and} \\ & \texttt{witnessed}\ W' \vdash \texttt{rel}\ s\ s' \quad \textbf{and} \quad W \subseteq W' \quad \textbf{and} \\ & \texttt{witnessed}\ W' \vdash \mathit{post}\ s\ v\ s'\end{array}$$

Glimpse of meta-theory

- A small **dependently typed λ-calculus** with $\mathtt{Tot}$ and $\mathtt{MST}$ effects
- **Logical consistency** shown via cut elimination
- Using an **instrumented operational semantics**, where

$$
\begin{array}{rcl}
(\mathtt{witness}\ p\,,\, s\,,\, W) & \rightsquigarrow & (\mathtt{return}\ ()\,,\, s\,,\, W \cup \{p\}) \\
(\mathtt{recall}\ p\,,\, s\,,\, W) & \rightsquigarrow & (\mathtt{return}\ ()\,,\, s\,,\, W)
\end{array}
$$

- **Strong normalisation** shown via type-erasure and $\top\top$-lifting
- Hoare-style **total correctness** via SN, progress, and preservation

$$
\begin{array}{ll}
\textbf{if} & \vdash e : \mathtt{MST}\ t\ \mathit{pre}\ \mathit{post} \quad \textbf{and} \\
& \vdash (s, W)\ \mathrm{wf} \quad \textbf{and} \quad \mathtt{witnessed}\, W \vdash \mathit{pre}\ s \\
\textbf{then} & (e, s, W) \rightsquigarrow^{*} (\mathtt{return}\ v, s', W') \quad \textbf{and} \quad \vdash v : t \quad \textbf{and} \\
& \mathtt{witnessed}\, W' \vdash \mathtt{rel}\ s\ s' \quad \textbf{and} \quad W \subseteq W' \quad \textbf{and} \\
& \mathtt{witnessed}\, W' \vdash \mathit{post}\ s\ v\ s'
\end{array}
$$

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 **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.