Embracing monotonicity in

Danel Ahman @ INRIA Paris

joint work with

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

(based on our POPL 2018 paper)

ICE-TCS Seminar
January 29, 2018
Outline

• F*

• Monotonic state by example

• Key ideas behind our general framework

• Accommodating monotonic state in F*

• Some examples of monotonic state at work

  • More examples of monotonic state at work (see the paper)

  • Monadic reification and reflection (see the paper)

  • Meta-theory and correctness results (see the paper)
Outline

• F*

• Monotonic state by example

• Key ideas behind our general framework

• Accommodating monotonic state in F*

• Some examples of monotonic state at work

• More examples of monotonic state at work (see the paper)

• Monadic reification and reflection (see the paper)

• Meta-theory and correctness results (see the paper)
F*

F* is

- a functional programming language
  - ML, OCaml, F#, Haskell, . . .
  - extracted to OCaml or F#; subset compiled to efficient C code
- an interactive proof assistant
  - Agda, Coq, Lean, Isabelle/HOL, . . .
  - interactive modes for Emacs and Atom
- a semi-automated verifier of imperative programs
  - Dafny, Why3, FramaC, . . .
  - Z3-based SMT-automation; tactics and metaprogramming (WIP)

Application-driven development

- Project Everest [project-everest.github.io]
- Microsoft Research (US, UK, India), INRIA (Paris), . . .
- miTLS, HACL*, Vale, . . .
F*

- **F** is
  - a functional programming language
    - ML, OCaml, F#, Haskell, ...
    - extracted to OCaml or F#; subset compiled to efficient C code
  - an interactive proof assistant
    - Agda, Coq, Lean, Isabelle/HOL, ...
    - interactive modes for Emacs and Atom
  - a semi-automated verifier of imperative programs
    - Dafny, Why3, FramaC, ...
    - Z3-based SMT-automation; tactics and metaprogramming (WIP)

- Application-driven development
  - Project Everest  
    [project-everest.github.io]
  - Microsoft Research (US, UK, India), INRIA (Paris), ...
  - miTLS, HACL*, Vale, ...
F* – a prog. lang./proof assistant/verifier

module Talk

// 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 n' 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{m <= 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 n' x xs' -> lemma xs' ys

// Intrinsic reasoning (making lemmas part of definitions)

val take : #a:Type -> #n:nat -> zs:vector a n -> m:nat -> Pure (vector a m) (requires (m <= n))
   (ensures (fun xs -> xs `is_prefix_of` zs))
let rec take #a #n #z m =  
   if m > 0 then match zs with | Cons z' zs' -> let m' : nat = m - 1 in Cons z (take zs' m')  
   else Nil
F* – not just a pure programming language

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

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

- Systematically derived from WP-calculi (see POPL’17 paper)
Outline

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

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

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

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

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

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

- However, if `c_p` **never removes**, then `λs. v ∈ s` is **stable**, and we would like the program logic to give us `v ∈ get()` **“for free”**
Monotonicity in program verification

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

  ```
  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**

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

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

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

- Consider a program operating on set-valued state

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

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

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

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

- However, if \(c_p\) never removes, then \(\lambda s. v \in s\) is stable, and we would like the program logic to give us \(v \in \text{get}()\) “for free”
Monotonicity in program verification

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

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

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

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

  - 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 realize 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 \( r: \text{ref } a \)
  - \( r \) 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: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 (\texttt{ref t}) and untyped references (\texttt{uref})
  - more flexibility with monotonic references (\texttt{mref t rel})

- See our POPL 2018 paper for more
  - temporarily violating monotonicity via snapshots
  - two substantial case studies in F*
    - a secure file-transfer application
    - Ariadne state continuity protocol [Strackx, Piessens 2016]
  - pointers to other works in F* relying on monotonicity for
    - sophisticated region-based memory models [fstar-lang.org]
    - crypto and TLS verification [project-everest.github.io]
Outline

- F*

- Monotonic state by example

- Key ideas behind our general framework

- Accommodating monotonic state in F*

- Some examples of monotonic state at work

- More examples of monotonic state at work (see the paper)

- Monadic reification and reflection (see the paper)

- Meta-theory and correctness results (see the paper)
Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** `rel` on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program `e` is **monotonic** (wrt. `rel`) when
    \[ \forall s e' s'. (e, s) \leadsto^* (e', s') \implies rel s s' \]
  - a stateful predicate `p` is **stable** (wrt. `rel`) when
    \[ \forall s s'. p s \land rel s s' \implies p s' \]

- **Our solution**: extend Hoare-style program logics (e.g., F*) with
  - 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
    \[ \forall s \ e' \ s'. \ (e, s) \rightsquigarrow^* (e', s') \ \Rightarrow \ rel \ s \ s' \]
  • a stateful predicate `p` is **stable** (wrt. `rel`) when
    \[ \forall s \ s'. \ p \ s \wedge \ rel \ s \ s' \ \Rightarrow \ p \ s' \]

• **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** \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, . . .

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

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

- **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** \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, . . .
  - a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    \[
    \forall s \ e' s'. \ (e, s) \leadsto^* (e', s') \implies \text{rel} \ s \ s'
    \]
- a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
  \[
  \forall s s'. \ s \ p \land \text{rel} \ s \ s' \implies p \ s'
  \]

- **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
    \[
    \forall s e' s'. \ (e, s) \sim^* (e', s') \implies \text{rel} \; s \; s'
    \]
  - a stateful predicate p is **stable** (wrt. rel) when
    \[
    \forall s s'. \ p \; s \land \text{rel} \; s \; s' \implies p \; s'
    \]

- **Our solution**: extend Hoare-style program logics (e.g., F*) with
  - 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 $\text{rel}$ on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program $e$ is **monotonic** (wrt. $\text{rel}$) when
    $$\forall s, e', s'. (e, s) \leadsto^* (e', s') \implies \text{rel } s \ s'$$
  - a stateful predicate $p$ is **stable** (wrt. $\text{rel}$) when
    $$\forall s, s'. p \ s \land \text{rel } s \ s' \implies p \ s'$$

- **Our solution**: extend Hoare-style program logics (e.g., F*) with
  - 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 \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, ...  
  - a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    \[
    \forall s\ e\ s'. \ (e, s) \xrightarrow{\ast} (e', s') \implies \text{rel} \ s \ s'
    \]
  - a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
    \[
    \forall s\ s'. \ p \ s \land \text{rel} \ s \ s' \implies p \ s'
    \]

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

- Provides a **unifying account** of the existing ad hoc uses in F*
Outline

- F*

- Monotonic state by example

- Key ideas behind our general framework

- Accommodating monotonic state in F*

- Some examples of monotonic state at work

- More examples of monotonic state at work (see the paper)

- Monadic reification and reflection (see the paper)

- Meta-theory and correctness results (see the paper)
Recap: Ordinary global state in F*

- F* supports Hoare-style reasoning about state via the comp. type
  \[ \text{ST}_{\text{state}} \ t \ (\text{requires} \ \text{pre}) \ (\text{ensures} \ \text{post}) \]

  where

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

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

- The global state \textbf{actions} have types
  \[
  \begin{align*}
  \text{get} : \text{unit} & \to \text{ST} \ \text{state} \ (\text{requires} \ (\lambda \_ . \top)) \ (\text{ensures} \ (\lambda s_0 \ s s_1 . s_0 = s = s_1)) \\
  \text{put} : \ s : \text{state} & \to \text{ST} \ \text{unit} \ (\text{requires} \ (\lambda \_ . \top)) \ (\text{ensures} \ (\lambda \_ \_ s_1 . s_1 = s))
  \end{align*}
  \]

- Refs. and \textbf{local state} are defined in F* using \textbf{monotonicity}
Recap: Ordinary global state in F*

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

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

where

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

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

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

- The global state actions have types

\[
\text{get} : \text{unit} \to \text{ST} \ \text{state} \ (\text{requires} \ (\lambda _.-.\top)) \ (\text{ensures} \ (\lambda s_0 \ s \ s_1. s_0 = s = s_1))
\]
\[
\text{put} : \text{s:state} \to \text{ST} \ \text{unit} \ (\text{requires} \ (\lambda _.-.\top)) \ (\text{ensures} \ (\lambda _-_s_1. s_1 = s))
\]

- Refs. and local state are defined in F* using monotonicity
Recap: Ordinary global state in F*

- F* supports Hoare-style reasoning about state via the \texttt{comp. type}

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

where

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

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

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

- The global state \textbf{actions} have types

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

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

- Refs. and local state are defined in F* using monotonicity
Recap: Ordinary global state in F*

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

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

where

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

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

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

- The global state **actions** have types

\[ \text{get : unit} \rightarrow \text{ST state (requires (λ_.T)) (ensures (λs_0 \; s \; s_1. s_0 = s = s_1))} \]
\[ \text{put : s:state} \rightarrow \text{ST unit (requires (λ_.T)) (ensures (λ_\_s_1. s_1 = s))} \]

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

- We capture monotonic state with a new computational type

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

- The \texttt{get} action is typed as in \texttt{ST}

  \[
  \text{get : unit} \rightarrow \text{MST state (requires (λ_ _ . T))}
  \]
  \[
  (\text{ensures (λ s_0 s s_1 . s_0 = s = s_1)})
  \]

- To ensure \textit{monotonicity}, the \texttt{put} action gets a precondition

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

- So intuitively, \texttt{MST} is an \texttt{abstract} pre-postcondition refinement of

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

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

- The `get` action is typed as in ST
  \[ \text{get} : \text{unit} \rightarrow \text{MST state} \ (\text{requires } (\lambda \_. \top)) \]
  \[ (\text{ensures } (\lambda s_0 s s_1 . s_0 = s = s_1)) \]

- To ensure monotonicity, the `put` action gets a precondition
  \[ \text{put} : s : \text{state} \rightarrow \text{MST unit} \ (\text{requires } (\lambda s_0 . \text{rel } s_0 \ s)) \]
  \[ (\text{ensures } (\lambda \_. \_. s_1 . s_1 = s)) \]

- So intuitively, MST is an abstract pre-postcondition refinement of
  \[ \text{mst } t \overset{\text{def}}{=} s_0 : \text{state} \rightarrow t \ast s_1 : \text{state} \{ \text{rel } s_0 \ s_1 \} \]
New: Monotonic global state in F*

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

- The get action is typed as in ST
  \[
  \text{get} : \text{unit} \rightarrow \text{MST state} (\text{requires } (\lambda _. \top))
  
  (\text{ensures } (\lambda s_0 s s_1 . s_0 = s = s_1))
  \]

- To ensure monotonicity, the put action gets a precondition
  \[
  \text{put} : s : \text{state} \rightarrow \text{MST unit} (\text{requires } (\lambda s_0 . \text{rel } s_0 s))
  
  (\text{ensures } (\lambda _. s_1 . s_1 = s))
  \]

- So intuitively, \text{MST} is an abstract pre-postcondition refinement of
  \[
  \text{mst } t \overset{\text{def}}{=} s_0 : \text{state} \rightarrow t * s_1 : \text{state} \{\text{rel } s_0 s_1\}
  \]
New: Monotonic global state in F*

- We capture monotonic state with a new computational type

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

- The \texttt{get} action is typed as in \texttt{ST}

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

- To ensure \textit{monotonicity}, the \texttt{put} action gets a precondition

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

- So intuitively, \texttt{MST} is an \textit{abstract} pre-postcondition refinement of

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

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

- The get action is typed as in ST
  \[
  \text{get} : \text{unit} \to \text{MST state} \ (\text{requires } (\lambda _\_ . \top)) \\
  (\text{ensures } (\lambda s_0 s s_1 . s_0 = s = s_1))
  \]

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

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

- We extend F* with a logical capability

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

- Together with a weakening principle (functoriality)

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

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

- Intuitively, think of it as a necessity modality

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

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

- But, wait a minute …
New: Recalling a Witness

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

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

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

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

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

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

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

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

- But, wait a minute ...
New: Recalling a Witness

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

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

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

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

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

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

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

- As usual, for natural deduction, need world-indexed sequents
- But, wait a minute ...
New: Recalling a Witness

- We extend F* with a **logical capability**
  
  \[\text{witnessed} : (\text{state} \to \text{Type}) \to \text{Type}\]
  
  together with a **weakening principle** (functoriality)
  
  \[\text{wk} : p,q : (\text{state} \to \text{Type}) \to \text{Lemma} \ (\text{requires} \ (\forall s. p\ s \implies q\ s)) \]
  \[\quad (\text{ensures} \ (\text{witnessed} p \implies \text{witnessed} q))\]

- Intuitively, think of it as a **necessity modality**
  
  \[\llbracket\text{witnessed} p\rrbracket(s) \overset{\text{def}}{=} \forall s'. \text{rel} s\ s' \implies \llbracket p\ s'\rrbracket(s)\]

- As usual, for natural deduction, need **world-indexed sequents**
New: Recalling a Witness

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

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

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

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

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

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

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

- But, wait a minute . . .
New: Recalling a Witness

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

- we include a *stateful introduction rule* for witnessed

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

- and a *stateful elimination rule* for witnessed

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

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

• we include a **stateful introduction rule** for witnessed

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

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

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

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

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

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

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

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

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

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

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

- For any other \(w\), wrapping
  
  \[
  \text{insert } w; [ ]; \text{ assert } (w \in \text{get}())
  \]
  
  around the program is handled **similarly easily** by
  
  \[
  \text{insert } w; \text{ witness } (\lambda s. w \in s); [ ]; \text{ recall } (\lambda s. w \in s); \text{ assert } (w \in \text{get}())
  \]

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

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

  ```
  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 ∈ s); c_p(); recall (\lambda 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 (\lambda s. w ∈ s); [ ]; recall (\lambda s. w ∈ 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**
  \[
  \text{insert } v; \text{ complex\_procedure}(); \text{ assert } (v \in \text{get}())
  \]

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

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

- For any other \(w\), wrapping
  \[
  \text{insert } w; [ ]; \text{ assert } (w \in \text{get}())
  \] around the program is handled **similarly easily** by
  \[
  \text{insert } w; \text{ witness } (\lambda s. w \in s); [ ]; \text{ recall } (\lambda s. w \in s); \text{ assert } (w \in \text{get}())
  \]

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

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

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

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

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

  around the program is handled similarly easily by

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

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

- We **prove the assertion** by inserting a witness and recall
  
  ```
  insert v; witness (λs.v ∈ s); c.p(); recall (λs.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 (λs.w ∈ s); [ ]; recall (λs.w ∈ s); assert (w ∈ get())
  ```

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

```haskell
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**)

```haskell
let heap inclusion (H h0 _) (H h1 _) = ∀ id.match h0 id, h1 id with
  | Used a _, Used b _ → a = b
  | Unused, Used _ _ → True
  | Unused, Unused → True
  | Used _ _, Unused → ⊥
```
ML-style typed references (local state)

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

```plaintext
type heap =
    | H : h : (N → cell) → ctr : N { ∀ 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)

```plaintext
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)

● As a result, we can define new local state effect

\[
\text{MLST } t \text{ pre post } \overset{\text{def}}{=} \text{MST}_{\text{heap,heap inclusion}} t \text{ pre post}
\]

● Next, we define the type of references using monotonicity

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

where

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

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

\[
| \text{Used } b _ \rightarrow a = b
\]

\[
| \text{Unused } \rightarrow \bot
\]

● Important: contains is stable wrt. heap inclusion
ML-style typed references (local state)

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

\[
\text{MLST } t \text{ pre post } \overset{\text{def}}{=} \text{MST}_{\text{heap, heap\_inclusion}} t \text{ pre post}
\]

- Next, we define the type of references using monotonicity

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

where

\[
\text{let contains } (H \ h \ _) \text{ id } a = \\
\text{match } h \text{ id with} \\
| \text{Used } b \ _ \rightarrow a = b \\
| \text{Unused} \rightarrow \bot
\]

- Important: contains is stable wrt. heap\_inclusion
ML-style typed references (local state)

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

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

- Next, we define the type of references using monotonicity

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

where

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

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

\[
| \text{Used } b \_ \rightarrow a = b
\]

\[
| \text{Unused } \rightarrow \bot
\]

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

- Finally, we define MLST’s actions using MST’s actions
  - let alloc (a:Type) (v:a):MLST (ref a) ... = ...
    - get the current heap
    - create a fresh ref., and add it to the heap
    - put the updated heap back
    - witness that the created ref. is in the heap
  - let read (r:ref a):MLST a (req. (⊤)) (ens. (...)) = ...
    - recall that the given ref. is in the heap
    - get the current heap
    - select the given reference from the heap
  - let write (r:ref a) (v:a):MLST unit ... = ...
    - recall that the given ref. is in the heap
    - get the current heap
    - update the heap with the given value at the given ref.
    - put the updated heap back
ML-style typed references (local state)

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

- let alloc (a: Type) (v:a) : MLST (ref a) ... = ...
  - get the current heap
  - create a fresh ref., and add it to the heap
  - put the updated heap back
  - witness that the created ref. is in the heap

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

- let write (r:ref a) (v:a) : MLST unit ... = ...
  - recall that the given ref. is in the heap
  - get the current heap
  - update the heap with the given value at the given ref.
  - put the updated heap back
Finally, we define MLST’s actions using MST’s actions

- let alloc (a:Type) (v:a): MLST (ref a) ... = ...
  - get the current heap
  - create a fresh ref., and add it to the heap
  - put the updated heap back
  - witness that the created ref. is in the heap

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

- let write (r:ref a) (v:a): MLST unit ... = ...
  - recall that the given ref. is in the heap
  - get the current heap
  - update the heap with the given value at the given ref.
  - put the updated heap back
Adding untyped and monotonic references

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

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

- Further, all three can be extended with **manually managed** refs.
Adding untyped and monotonic references

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

- **Monotonic references** \((\text{mref a rel})\)
  - Used heap cells are extended with **typed tags**
    
    \[
    \text{Used} : \text{Type} \rightarrow \\mathit{v}:\text{a} \rightarrow \text{t}:\text{tag a} \rightarrow \text{cell}
    \]
    
    where
    \[
    \text{type tag a} = \ \text{Typed : rel:preorder a} \rightarrow \text{tag a} \ | \ \text{Untyped : tag a}
    \]
    
    - \text{mrefs} provide **more flexibility** with ref.-wise monotonicity

- Further, all three can be extended with **manually managed refs.**
Adding untyped and monotonic references

- **Untyped references** ([uref]) with strong updates
  - Used heap cells are extended with **tags**
    
    ```
    Used : a:Type → v:a → t:tag → cell
    where
    type tag = Typed : tag | Untyped : tag
    ```

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

- **Monotonic references** ([mref a rel])
  - Used heap cells are extended with **typed tags**
    
    ```
    Used : a:Type → v:a → t:tag a → cell
    where
    type tag a = Typed : rel:preorder a → tag a | Untyped : tag a
    ```

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

- Further, all three can be extended with manually managed refs.
Adding untyped and monotonic references

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

    \[
    \text{type tag} = \text{Typed} : \text{tag} \mid \text{Untyped} : \text{tag}
    \]
  - actions corresponding to \(\text{urefs}\) have **weaker types** than for \(\text{refs}\)

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

    \[
    \text{type tag a} = \text{Typed} : \text{rel:preorder} a \to \text{tag} a \mid \text{Untyped} : \text{tag} a
    \]
  - \(\text{mrefs}\) provide **more flexibility** with ref.-wise monotonicity

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

- Monotonicity
  - can be distilled into a **simple** and **general** framework
  - is **useful** for **programming** (refs.) and **verification** (Prj. Everest)

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

- Monotonicity
  - can be distilled into a **simple** and **general** framework
  - is **useful** for **programming** (refs.) and **verification** (Prj. Everest)

- See our POPL 2018 paper for
  - further **examples** and **case studies**
  - **meta-theory** and **total correctness** for **MST**
    - based on an instrumented operational semantics
      \[(\text{witness } x.\varphi, s, W) \leadsto (\text{return } (), s, W \cup \{x.\varphi\})\]
    - and cut elimination for the **witnessed-logic**
  - **first steps towards** **monadic reification** for **MST**
    - useful for extrinsic reasoning, e.g., for relational properties
    - but have to be careful when breaking abstraction
Thank you for your attention!

Questions?
Appendix: Mon. reification and reflection

- In F* every abstract ST computation
  
  \[ e : \text{ST} \ t \ (\text{requires} \ \text{pre}) \ (\text{ensures} \ \text{post}) \]

  can be reified into its underlying Pure representation

  \[ \text{reify} \ e : s_0 : \text{state} \to \text{Pure} \ (t \ast \text{state}) \ (\text{requires} \ (\text{pre} \ s_0)) \]
  
  \[ (\text{ensures} \ (\lambda (x, s_1). \text{post} \ s_0 \ast x \ast s_1)) \]

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

- Useful for extrinsic reasoning, e.g., for relational properties

- We also need it for MST!
Appendix: Mon. reification and reflection

- In F* every abstract ST computation
  
  \[ e : ST \ t (\text{requires pre}) (\text{ensures post}) \]

  can be reified into its underlying Pure representation

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

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

- Useful for extrinsic reasoning, e.g., for relational properties

- We also need it for MST!
Appendix: Mon. reification and reflection

- In F* every abstract ST computation

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

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

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

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

- Useful for **extrinsic reasoning**, e.g., for relational properties

- We also need it for **MST**!
Appendix: Mon. reification and reflection

- We cannot simply turn an abstract MST computation
  
  \[ e : \text{MST} \quad \text{t} \quad (\text{requires pre}) \quad (\text{ensures post}) \]

  into a state-passing function

  \[ s_0 : \text{state} \to \text{Pure} \quad (t \ast s_1 : \text{state}\{\text{rel} \ s_0 \ s_1\}) \quad (\text{req.} \ (\text{pre} \ s_0)) \]

  \[ (\text{ens.} \ (\lambda (x, s_1). \text{post} \ s_0 \ x \ s_1)) \]

- For example, consider the recalling action

  \[ \text{recall} : p : (\text{state} \to \text{Type}) \to \text{MST} \quad \text{unit} \quad (\text{requires} \ (\lambda \_. \text{witnessed} \ p)) \]

  \[ (\text{ensures} \ (\lambda s_0 \ s_1. s_0 = s_1 \land p \ s_1)) \]

  which we would like to reduce as

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

  but we cannot prove \( p \ s_0 \) from \text{witnessed} \( p \) in the pure logic
Appendix: Mon. reification and reflection

- We cannot simply turn an abstract MST computation

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

into a state-passing function

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

- For example, consider the recalling action

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

which we would like to reduce as

\[ \text{reify (recall p) } \sim \lambda s_0. \text{return } ((), s_0) \]

but we cannot prove \( p s_0 \) from \( \text{witnessed p} \) in the pure logic
Appendix: Mon. reification and reflection

- We cannot simply turn an abstract MST computation
  \[ e : \text{MST } t \text{ (requires pre) (ensures post)} \]
  into a state-passing function
  \[ s_0 : \text{state} \rightarrow \text{Pure } (t \ast s_1 : \text{state}\{\text{rel } s_0 s_1\}) \text{ (req. (pre } s_0)) \]
  \[ \quad(\text{ens. } (\lambda (x, s_1). \text{post } s_0 x s_1)) \]

- For example, consider the recalling action
  \[ \text{recall : } p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit (requires } (\lambda \_ . \text{witnessed } p)) \]
  \[ \quad(\text{ensures } (\lambda s_0 s_1. s_0 = s_1 \land p s_1)) \]
  which we would like to reduce as
  \[ \text{reify } \text{(recall } p) \rightsquigarrow \lambda s_0 . \text{return } ((), s_0) \]
  but we cannot prove \( p s_0 \) from \text{witnessed } p \text{ in the pure logic}
Appendix: Mon. reification and reflection

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

- This works but leads to **duplication** of pre- and postconditions!

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

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

where $@$ is the **standard translation** of modal logic.
Appendix: Mon. reification and reflection

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

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

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

where $@$ is the standard translation of modal logic
Appendix: Mon. reification and reflection

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

\[
\begin{align*}
s_0:& \text{state} \rightarrow \text{Pure} \ (t * s_1: \text{state}\{\text{rel } s_0 s_1\}) \ (\text{req. } (\text{pre } s_0 @ s_0)) \\
& \quad \ (\text{ens. } (\lambda (x, s_1). \text{post } s_0 x s_1 @ s_1))
\end{align*}
\]

where @ is the standard translation of modal logic