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

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

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

```
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 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)

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 zs 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 (requires` $\texttt{pre}_{\texttt{Lemma}}$`) (ensures` $\texttt{post}_{\texttt{Lemma}}$`)`
    - `Pure t (requires` $\texttt{pre}_{\texttt{Pure}}$`) (ensures` $\texttt{post}_{\texttt{Pure}}$`)`
    - `Div t (requires` $\texttt{pre}_{\texttt{Div}}$`) (ensures` $\texttt{post}_{\texttt{Div}}$`)`
    - `Exc t (requires` $\texttt{pre}_{\texttt{Exc}}$`) (ensures` $\texttt{post}_{\texttt{Exc}}$`)`
    - `ST t (requires` $\texttt{pre}_{\texttt{ST}}$`) (ensures` $\texttt{post}_{\texttt{ST}}$`)`
    - ...
- **Monad morphs.** $\texttt{Pure} \rightsquigarrow \{\texttt{Div}, \texttt{Exc}, \texttt{ST}\}$; $\texttt{Exc} \rightsquigarrow \texttt{STExc}$; ...
- Systematically derived from **WP-calculi** (see POPL'17 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)

Monotonicity in program verification

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

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

- 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 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**

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

- 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**

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

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

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** `rel` on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program e is **monotonic** (wrt. `rel`) when

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

  - 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

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

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

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

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

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** `rel` on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program e is **monotonic** (wrt. `rel`) when

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

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

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

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

Key ideas behind our general framework

- Based on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** `rel` on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program e is **monotonic** (wrt. `rel`) when
    $$\forall\, s\, e'\, s'.\ (e, s) \rightsquigarrow^{*} (e', s') \implies \mathtt{rel}\ s\ s'$$
  - 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

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

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

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

- **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*

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

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

- 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

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

- ST is an abstract pre-postcondition refinement of

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

- The global state **actions** have types

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

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

- **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

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

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

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

- 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

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

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

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

- 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

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

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

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

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

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

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

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

- 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

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

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 $\mathtt{ST}$

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

- 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**

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

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

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

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

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

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

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

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

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

New: Recalling a Witness

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

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

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

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

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

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

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

New: Recalling a Witness

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

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

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

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

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

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

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

New: Recalling a Witness

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

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

  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**

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

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

New: Recalling a Witness

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

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

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

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

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

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

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

New: Recalling a Witness

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

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

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

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

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

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

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

New: Recalling a Witness

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

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

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

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

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

New: Recalling a Witness

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

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

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

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

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

New: Recalling a Witness

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

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

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

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

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

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**

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

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

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

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

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

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

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

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

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

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**

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

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

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

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

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

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

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

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

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

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**

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

  - 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.,

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

ML-style typed references (local state)

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

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

  where

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

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

$$\texttt{let heap\_inclusion}\ (\texttt{H}\ \texttt{h}_0\ \_)\ (\texttt{H}\ \texttt{h}_1\ \_) = \forall\, \texttt{id}\,.\, \texttt{match}\ \texttt{h}_0\ \texttt{id}\,,\texttt{h}_1\ \texttt{id}\ \texttt{with}$$
$$\mid \texttt{Used a}\ \_\,, \texttt{Used b}\ \_ \ \to\ \texttt{a} = \texttt{b}$$
$$\mid \texttt{Unused}\,, \texttt{Used}\ \_\ \_ \ \to\ \top$$
$$\mid \texttt{Unused}\,, \texttt{Unused} \ \to\ \top$$
$$\mid \texttt{Used}\ \_\ \_\,, \texttt{Unused} \ \to\ \bot$$

ML-style typed references (local state)

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

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

  where

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

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

$$\mathtt{let}\ \mathtt{heap\_inclusion}\ (\mathtt{H}\ \mathtt{h}_0\ \_)\ (\mathtt{H}\ \mathtt{h}_1\ \_) = \forall\,\mathtt{id}\,.\,\mathtt{match}\ \mathtt{h}_0\ \mathtt{id}\,,\mathtt{h}_1\ \mathtt{id}\ \mathtt{with}$$
$$\mid \mathtt{Used}\ \mathtt{a}\ \_\,,\mathtt{Used}\ \mathtt{b}\ \_\ \to\ \mathtt{a} = \mathtt{b}$$
$$\mid \mathtt{Unused}\,,\mathtt{Used}\ \_\ \_\ \to\ \top$$
$$\mid \mathtt{Unused}\,,\mathtt{Unused}\ \to\ \top$$
$$\mid \mathtt{Used}\ \_\ \_\,,\mathtt{Unused}\ \to\ \bot$$

ML-style typed references (local state)

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

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

where

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

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

$$\texttt{let heap\_inclusion}\ (\texttt{H}\ \texttt{h}_0\ \_)\ (\texttt{H}\ \texttt{h}_1\ \_) = \forall\, \texttt{id}\,.\,\texttt{match}\ \texttt{h}_0\ \texttt{id}\,,\texttt{h}_1\ \texttt{id}\ \texttt{with}$$
$$\mid \texttt{Used a}\ \_\,,\texttt{Used b}\ \_ \ \rightarrow\ \texttt{a} = \texttt{b}$$
$$\mid \texttt{Unused}\,,\texttt{Used}\ \_\ \_ \ \rightarrow\ \top$$
$$\mid \texttt{Unused}\,,\texttt{Unused} \ \rightarrow\ \top$$
$$\mid \texttt{Used}\ \_\ \_\,,\texttt{Unused} \ \rightarrow\ \bot$$

ML-style typed references (local state)

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

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

- 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**

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

- 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**

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

- 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 read` (r:`ref a`) : `MLST a` (`req.` ($\top$)) (`ens.` (...)) = ...
    - **recall** that the given ref. is in the heap
    - **get** the current heap
    - **select** the given reference from the heap
  - `let write` (r:`ref a`) (v:a) : `MLST unit` ... = ...
    - **recall** that the given ref. is in the heap
    - **get** the current heap
    - **update** the heap with the given value at the given ref.
    - **put** the updated heap back

ML-style typed references (local state)

- Finally, we define `MLST`'s **actions** using `MST`'s actions
  - `let alloc (a: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.` ($\top$)) (`ens.` (...)) $=$ ...
    - **recall** that the given ref. is in the heap
    - **get** the current heap
    - **select** the given reference from the heap
  - `let write` (`r:ref a`) (v:a) : `MLST unit ...  =  ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap
    - **update** the heap with the given value at the given ref.
    - **put** the updated heap back

ML-style typed references (local state)

- Finally, we define `MLST`'s **actions** using `MST`'s actions
  - `let alloc (a: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**

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

    where

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

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

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

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

    where

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

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

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

Adding untyped and monotonic references

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

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

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

    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**

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

    where

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

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

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

Adding untyped and monotonic references

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

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

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

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

    $$\mathtt{type\ tag\ a} \ = \ \mathtt{Typed} : \mathtt{rel{:}preorder\ a} \to \mathtt{tag\ a} \ \mid \ \mathtt{Untyped} : \mathtt{tag\ a}$$
  - `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
- Using an **instrumented operational semantics**, where

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

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

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

Glimpse of meta-theory

- A small **dependently typed λ-calculus** with `Tot` and `MST` effects
- 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

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

Glimpse of meta-theory

- A small **dependently typed λ-calculus** with `Tot` and `MST` effects
- 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 `Tot` and `MST` effects
- 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**

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

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

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.

Appendix: Mon. reification and reflection

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

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

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

$$\begin{array}{l} \mathtt{reify}\ \mathtt{e} : \mathtt{s_0}{:}\mathtt{state} \to \mathtt{Pure}\ (\mathtt{t} * \mathtt{state})\ (\mathtt{requires}\ (\mathtt{pre}\ \mathtt{s_0})) \\ \qquad\qquad\qquad\qquad\qquad\qquad (\mathtt{ensures}\ (\lambda\ (\mathtt{x}, \mathtt{s_1})\ .\ \mathtt{post}\ \mathtt{s_0}\ \mathtt{x}\ \mathtt{s_1})) \end{array}$$

  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**

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

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

  $$\begin{aligned}\texttt{reify}\ \texttt{e} : \texttt{s}_0{:}\texttt{state} \to \texttt{Pure}\ (\texttt{t} * \texttt{state})\ &(\texttt{requires}\ (\texttt{pre}\ \texttt{s}_0))\\ &(\texttt{ensures}\ (\lambda\ (\texttt{x}, \texttt{s}_1)\,.\,\texttt{post}\ \texttt{s}_0\ \texttt{x}\ \texttt{s}_1))\end{aligned}$$

  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**

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

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

  $$\mathtt{reify}\ \mathtt{e} : \mathtt{s}_0{:}\mathtt{state} \to \mathtt{Pure}\ (\mathtt{t} * \mathtt{state})\ (\mathtt{requires}\ (\mathtt{pre}\ \mathtt{s}_0))$$
  $$(\mathtt{ensures}\ (\lambda\ (\mathtt{x}, \mathtt{s}_1)\,.\,\mathtt{post}\ \mathtt{s}_0\ \mathtt{x}\ \mathtt{s}_1))$$

  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**

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

into a **state-passing function**

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

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

$$\mathtt{recall} : \mathtt{p}{:}(\mathtt{state} \to \mathtt{Type}) \to \mathtt{MST}\ \mathtt{unit}\ (\mathtt{requires}\ (\lambda\ \_\ .\ \mathtt{witnessed}\ \mathtt{p}))$$
$$(\mathtt{ensures}\ (\lambda\ \mathtt{s}_0\ \_\ \mathtt{s}_1\ .\ \mathtt{s}_0 = \mathtt{s}_1\ \wedge\ \mathtt{p}\ \mathtt{s}_1))$$

which we would like to **reduce** as

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

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

Appendix: Mon. reification and reflection

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

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

  into a **state-passing function**

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

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

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

  which we would like to **reduce** as

$$\mathtt{reify}\ (\mathtt{recall}\ \mathtt{p}) \leadsto \lambda\ \mathtt{s_0}\ .\ \mathtt{return}\ ((), \mathtt{s_0})$$

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

Appendix: Mon. reification and reflection

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

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

  into a **state-passing function**

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

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

$$\mathtt{recall} : \mathtt{p}{:}(\mathtt{state} \rightarrow \mathtt{Type}) \rightarrow \mathtt{MST}\ \mathtt{unit}\ (\mathtt{requires}\ (\lambda\,\_\,.\,\mathtt{witnessed}\ \mathtt{p}))$$
$$(\mathtt{ensures}\ (\lambda\,\mathtt{s}_0\,\_\,\mathtt{s}_1\,.\,\mathtt{s}_0 = \mathtt{s}_1 \wedge \mathtt{p}\ \mathtt{s}_1))$$

  which we would like to **reduce** as

$$\mathtt{reify}\ (\mathtt{recall}\ \mathtt{p}) \ \rightsquigarrow\ \lambda\,\mathtt{s}_0\,.\,\mathtt{return}\ ((), \mathtt{s}_0)$$

  but we cannot prove $\mathtt{p}\ \mathtt{s}_0$ from $\mathtt{witnessed}\ \mathtt{p}$ in the pure logic

Appendix: Mon. reification and reflection

- In our POPL 2018 paper, we support reification and reflection by
  - indexing $\mathtt{MST}_{\mathtt{state},\mathtt{rel},\mathtt{b}}$ with a **boolean flag** `b` (reifiable?), and
  - **guarding** the pre-postconditions of `witness` and `recall` with `b`

  so if `b = 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

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

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

Appendix: Mon. reification and reflection

- In our POPL 2018 paper, we support reification and reflection by
  - indexing $\texttt{MST}_{\texttt{state},\texttt{rel},\texttt{b}}$ with a **boolean flag** `b` (reifiable?), and
  - **guarding** the pre-postconditions of `witness` and `recall` with `b`

  so if `b = 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

$$
\begin{aligned}
s_0{:}\mathtt{state} \to \mathtt{Pure}\ (\mathtt{t} * s_1{:}\mathtt{state}\{\mathtt{rel}\ s_0\ s_1\})\ &(\mathtt{req.}\ (\mathtt{pre}\ s_0\ \mathbf{@}\ s_0))\\
&(\mathtt{ens.}\ (\lambda\ (\mathtt{x}, s_1)\ .\ \mathtt{post}\ s_0\ \mathtt{x}\ s_1\ \mathbf{@}\ s_1))
\end{aligned}
$$

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 $\mathtt{MST}_{\mathtt{state},\mathtt{rel},\mathtt{b}}$ with a **boolean flag** b (reifiable?), and
  - **guarding** the pre-postconditions of `witness` and `recall` with b

  so if `b = 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

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

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