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* [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, ...

F* – a prog. lang./proof assistant/verifier

```
module Talk

// 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 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 (nat is defined as z:int{z >= 0})

val lkp : #a:Type -> #n:nat -> vector a n -> i:nat{0 < i /\ i <= n} -> a
let rec lkp #a #n xs i =
  match xs with
  | Cons x xs -> if i = n then x else lkp xs i

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

type is_prefix_of (#a:Type) (#n:nat) (#m:nat{n <= m}) (xs:vector a n) (zs:vector a m) : Type0 =
  forall (i:nat) . (0 < i /\ i <= n) ==> lkp xs i == lkp zs i

type is_suffix_of (#a:Type) (#n:nat) (#m:nat{n <= m}) (ys:vector a n) (zs:vector a m) : Type0 =
  forall (i:nat) . (0 < i /\ i <= n) ==> lkp ys i == lkp zs (m - n + 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 lemma #a #n #m xs ys =
  match xs with
  | Nil -> ()
  | Cons x xs -> admit () // need to call an appropriate sub-lemma here

// Intrinsic reasoning (making lemmas part of definitions)

val append' : #a:Type -> #n:nat -> #m:nat -> xs:vector a n -> ys:vector a m -> Pure (vector a (n + m)) (requires (True))
                                                                                                         (ensures  (fun zs -> xs `is_prefix_of` zs
                                                                                                                           /\ ys `is_suffix_of` zs))
```

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*
- 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**

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

- 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**

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

- 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**

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

- 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 program verification

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

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

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

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

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

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

Monotonicity in programming

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

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

- Correctness relies on **monotonicity**!
  1) Allocation **stores** an `a`-typed value in the heap
  2) Writes **don't change type** and there is **no deallocation**
  3) So, given a ref. `r`, it is **guaranteed to point** to an `a`-typed value

- Baked into the memory models of most languages
- We derive them from **global state + general monotonicity**

Monotonicity in programming

- **Programming** also relies on **monotonicity**,

  even if you don't realise it!

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

- Correctness relies on **monotonicity**!
  1) Allocation **stores** an `a`-typed value in the heap
  2) Writes **don't change type** and there is **no deallocation**
  3) So, given a ref. `r`, it is **guaranteed to point** to an `a`-typed value

- Baked into the memory models of most languages

- We derive them from **global state + general monotonicity**

Monotonicity in programming

- **Programming** also relies on **monotonicity**,

  even if you don't realise it!

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

- Correctness relies on **monotonicity**!
  1) Allocation **stores** an `a`-typed value in the heap
  2) Writes **don't change type** and there is **no deallocation**
  3) So, given a ref. `r`, it is **guaranteed to point** to an `a`-typed value

- Baked into the memory models of most languages

- We derive them from **global state** + **general monotonicity**

Monotonicity in programming

- **Programming** also relies on **monotonicity**,

  even if you don't realise it!

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

- Correctness relies on **monotonicity**!
  1) Allocation **stores** an `a`-typed value in the heap
  2) Writes **don't change type** and there is **no deallocation**
  3) So, given a ref. `r`, it is **guaranteed to point** to an `a`-typed value

- Baked into the memory models of most languages
- We derive them from **global state** + **general monotonicity**

Monotonicity is really useful!

- In this talk, we will see how monotonicity gives us
  - our **motivating example** and **monotonic counters**
  - **typed references** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)
- See our POPL 2018 paper for more
  - temporarily **violating monotonicity** via snapshots
  - two substantial case studies in F*
    - a **secure file-transfer** application
    - Ariadne **state continuity** protocol [Strackx, Piessens 2016]
  - pointers to other works in F* relying on monotonicity for
    - sophisticated **region-based memory models** [`fstar-lang.org`]
    - **crypto** and **TLS verification** [`project-everest.github.io`]

Monotonicity is really useful!

- In this talk, we will see how monotonicity gives us
  - our **motivating example** and **monotonic counters**
  - **typed references** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)

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

Monotonicity is really useful!

- In this talk, we will see how monotonicity gives us
  - our **motivating example** and **monotonic counters**
  - **typed references** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)

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

Outline

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

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

- **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

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

- **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

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

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

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

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

Outline

- F*
- 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**

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

  where

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

- $\mathtt{ST}$ 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

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

- $\mathtt{ST}$ 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

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

- 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

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

- The global state **actions** have types

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

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

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

New: Monotonic global state in F*

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

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

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

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

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

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

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

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

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 ST

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

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

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

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

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

New: Monotonic global state in F*

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

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

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

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

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

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

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

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

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 $\mathtt{ST}$

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

- 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, $\mathtt{MST}$ 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 $\mathtt{ST}$

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

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

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

- 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: Recalling a Witness

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

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

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

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

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

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

- 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**)

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

- 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**

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

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

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

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

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

- 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**)

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

- 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 $\mathtt{witnessed}$

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

- and a **stateful elimination rule** for $\mathtt{witnessed}$

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

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`

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

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

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

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**

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

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

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

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

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

    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.,

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

The motivating example revisited

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

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

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

`insert v; witness (λ s . v ∈ s); c_p(); recall (λ s . v ∈ s); assert (v ∈ get())`

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

    `insert w; [ ]; assert (w ∈ get())`

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

`insert w; witness (λ s . w ∈ s); [ ]; recall (λ s . w ∈ s); assert (w ∈ get())`

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

  `create 0; incr(); witness (λ c . c > 0); c_p(); recall (λ c . c > 0)`

The motivating example revisited

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

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

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

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

  - For **any other** w, wrapping

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

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

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

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

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

The motivating example revisited

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

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

  - 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

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

    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.,

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

The motivating example revisited

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

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

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

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

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

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

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

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

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

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

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

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

  where

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

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

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

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

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

- 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**

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

- 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**

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

- 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 t ... = ...`
    - **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 t ... = ...`
    - **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**

    where
    $$| \ \texttt{Used} : \mathrm{a}{:}\mathrm{Type} \to \mathrm{v}{:}\mathrm{a} \to \texttt{t}{:}\texttt{tag} \to \texttt{cell}$$
    $$\texttt{type tag} \ = \ \texttt{Typed} : \texttt{tag} \ | \ \texttt{Untyped} : \texttt{tag}$$
  - actions corresponding to `urefs` have **weaker types** than for `refs`

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

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

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

    where

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

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

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

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

    where

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

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

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

Adding untyped and monotonic references

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

    where

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

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

  - 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

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

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

  - `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**

    where
    $$| \ \mathtt{Used} : \mathsf{a}{:}\mathsf{Type} \rightarrow \mathsf{v}{:}\mathsf{a} \rightarrow \mathsf{t}{:}\mathsf{tag} \rightarrow \mathtt{cell}$$
    $$\mathtt{type}\ \mathtt{tag} \ = \ \mathtt{Typed} : \mathtt{tag} \ | \ \mathtt{Untyped} : \mathtt{tag}$$
  - actions corresponding to `uref`s have **weaker types** than for `ref`s
- **Monotonic references** (`mref a rel`)
  - Used heap cells are extended with **typed tags**

    where
    $$| \ \mathtt{Used} : \mathsf{a}{:}\mathsf{Type} \rightarrow \mathsf{v}{:}\mathsf{a} \rightarrow \mathsf{t}{:}\mathsf{tag}\ \mathsf{a} \rightarrow \mathtt{cell}$$
    $$\mathtt{type}\ \mathtt{tag}\ \mathtt{a} \ = \ \mathtt{Typed} : \mathsf{rel}{:}\mathsf{preorder}\ \mathsf{a} \rightarrow \mathtt{tag}\ \mathtt{a} \ | \ \mathtt{Untyped} : \mathtt{tag}\ \mathtt{a}$$
  - `mref`s provide **more flexibility** with ref.-wise monotonicity
- Further, all three can be extended with **manually managed** refs.

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

$$(\texttt{witness}\ x.\varphi\ ,\ s\ ,\ W) \ \rightsquigarrow\ (\texttt{return}\ ()\ ,\ s\ ,\ W \cup \{x.\varphi\})$$

    - 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

$$(\mathtt{witness}\; x.\varphi\,,\, s\,,\, W) \rightsquigarrow (\mathtt{return}\;()\,,\, s\,,\, W \cup \{x.\varphi\})$$

    - 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**

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

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

$$\mathtt{reify}\ \mathtt{e} : \mathtt{s_0}{:}\mathtt{state} \rightarrow \mathtt{Pure}\ (\mathtt{t} * \mathtt{state})\ (\mathtt{requires}\ (\mathtt{pre}\ \mathtt{s_0}))\ (\mathtt{ensures}\ (\lambda\ (\mathtt{x}, \mathtt{s_1})\ .\ \mathtt{post}\ \mathtt{s_0}\ \mathtt{x}\ \mathtt{s_1}))$$

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

- 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**

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

  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**

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

  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

- 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

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

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

into a **state-passing function**

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

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

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

which we would like to **reduce** as

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

but we cannot prove $\texttt{p s}_0$ from $\texttt{witnessed p}$ 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

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

  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 $\texttt{MST}_{\texttt{state},\texttt{rel},\texttt{b}}$ 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

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

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 $\mathtt{MST}_{\mathtt{state},\mathtt{rel},\mathtt{b}}$ 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

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

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

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** $\mathtt{b}$ (reifiable?), and
  - **guarding** the pre-postconditions of `witness` and `recall` with $\mathtt{b}$

  so if $\mathtt{b} = \mathtt{true}$ 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

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

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