Recalling a Witness

Foundations and Applications of Monotonic State

Danel Ahman

Prosecco Team at Inria Paris

joint work with

Cătălin Hrițcu and Kenji Maillard @ Inria Paris

Cédric Fournet, Aseem Rastogi, and Nikhil Swamy @ MSR

HOPE 2017

September 3, 2017

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F*
- Examples of monotonic state at work
- A glimpse of the meta-theory

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F*
- Examples of monotonic state at work
- A glimpse of the meta-theory

Monotonic state and 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
    - **sensitive** to proving that c_p maintains $\lambda\, \texttt{s}\,.\, \texttt{w} \in \texttt{s}$ for some other w
    - **does not guarantee** that $\lambda\, \texttt{s}\,.\, \texttt{v} \in \texttt{s}$ holds at every point in c_p

- However, if c_p **only inserts**, 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**"

Monotonic state and 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
    - **sensitive** to proving that c_p maintains $\lambda\, \texttt{s}\,.\,\texttt{w} \in \texttt{s}$ for some other $\texttt{w}$
    - **does not guarantee** that $\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}$ holds at every point in c_p

- However, if c_p **only inserts**, 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**"

Monotonic state and 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
    - **sensitive** to proving that c_p maintains $\lambda\,\texttt{s}\,.\,\texttt{w} \in \texttt{s}$ for some other w
    - **does not guarantee** that $\lambda\,\texttt{s}\,.\,\texttt{v} \in \texttt{s}$ holds at every point in c_p

- However, if c_p **only inserts**, 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**"

Monotonic state and 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
    - **sensitive** to proving that c_p maintains $\lambda\, \texttt{s}\,.\,\texttt{w} \in \texttt{s}$ for some other w
    - **does not guarantee** that $\lambda\, \texttt{s}\,.\,\texttt{v} \in \texttt{s}$ holds at every point in c_p

- However, if c_p **only inserts**, 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 is really useful!

- To come later in this talk
  - reasoning about **monotonic counters**
  - implementing **typed** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)

- For other examples of the usefulness of monotonicity,

  Recalling a Witness:
  Foundations and Applications of Monotonic State

  (arXiv:1707.02466)

  which includes

  - a secure **file-transfer** application
  - pointers to works using monotonicity in **crypto** and **TLS verif.**
  - Ariadne **state continuity** protocol [Strackx, Piessens 2016]

Monotonicity is really useful!

- To come later in this talk
  - reasoning about **monotonic counters**
  - implementing **typed** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)

- For other examples of the usefulness of monotonicity,

  Recalling a Witness:
  Foundations and Applications of Monotonic State
  (arXiv:1707.02466)

  which includes

  - a secure **file-transfer** application
  - pointers to works using monotonicity in **crypto** and **TLS verif.**
  - Ariadne **state continuity** protocol [Strackx, Piessens 2016]

Monotonicity is really useful!

- To come later in this talk
  - reasoning about **monotonic counters**
  - implementing **typed** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)

- For other examples of the usefulness of monotonicity,

  Recalling a Witness:
  Foundations and Applications of Monotonic State

  (arXiv:1707.02466)

  which includes
  - a secure **file-transfer** application
  - pointers to works using monotonicity in **crypto** and **TLS verif.**
  - Ariadne **state continuity** protocol [Strackx, Piessens 2016]

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F*
- Examples of monotonic state at work
- A glimpse of the meta-theory

Overview of our solution

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

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

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

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

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

Overview of our solution

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

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

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

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

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

Overview of our solution

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

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

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

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

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

Overview of our solution

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

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

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

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

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

Overview of our solution

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

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

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

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

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

Overview of our solution

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

$$(s, e) \leadsto^* (s', e') \implies \texttt{rel}\ s\ s'$$

  - a predicate `p` on states 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
  - means for turning a `p` into a **state-independent proposition**
  - operation to **witness** the validity of `p` s in some state s
  - operation to **recall** the validity of `p` s′ in a future state s′

- A **unifying account** of the ad hoc uses of monotonicity in F*

Overview of our solution

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

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

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

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

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

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F*
- Examples of monotonic state at work
- A glimpse of the meta-theory

Reasoning about ordinary state in F*

- An ML-like dependently typed language, aimed at verification

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

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

  where

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

  (formally, this type is derived from a WP calculus for state)

- The **get** and **put** actions are typed as follows

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

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

Reasoning about ordinary state in F*

- An ML-like dependently typed language, aimed at verification
- F* supports Hoare-style reasoning about state via the **comp. type**

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

  where

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

  (formally, this type is derived from a WP calculus for state)

- The **get** and **put** actions are typed as follows

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

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

Reasoning about ordinary state in F*

- An ML-like dependently typed language, aimed at verification
- F* supports Hoare-style reasoning about state via the **comp. type**

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

  where

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

  (formally, this type is derived from a WP calculus for state)

- The **get** and **put** actions are typed as follows

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

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

Reasoning about monotonic state in F*

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

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

  where t, pre, and post are typed as in ST

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

- To ensure **monotonicity**, the **put** action is typed as follows

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

  - thus MST is a bit like an **update monad** [A., Uustalu'14]

Reasoning about monotonic state in F*

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

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

  where t, pre, and post are typed as in ST

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

- To ensure **monotonicity**, the **put** action is typed as follows

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

  - thus MST is a bit like an **update monad** [A., Uustalu'14]

Reasoning about monotonic state in F*

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

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

  where `t`, `pre`, and `post` are typed as in `ST`

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

- To ensure **monotonicity**, the **put** action is typed as follows

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

  - thus `MST` is a bit like an **update monad** [A., Uustalu'14]

Reasoning about monotonic state in F*

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

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

  where `t`, `pre`, and `post` are typed as in `ST`

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

- To ensure **monotonicity**, the **put** action is typed as follows

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

  - thus `MST` is a bit like an **update monad** [A., Uustalu'14]

Reasoning about monotonic state in F*

- We introduce a **logical capability**

$$\mathtt{witnessed} : \mathtt{pred\ state} \to \mathtt{Type}$$

together with a **weakening** principle

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

- We introduce an operation for **witnessing** stable predicates

$$\begin{aligned}\mathtt{witness} : \mathtt{p{:}pred\ state} \to \mathtt{MST\ unit}\ &(\mathtt{requires}\ (\lambda\, \mathtt{s_0}\,.\, \mathtt{p\ s_0} \wedge \mathtt{stable\ p}))\\ &(\mathtt{ensures}\ (\lambda\, \mathtt{s_0\ \_\ s_1}\,.\, \mathtt{s_0} = \mathtt{s_1} \wedge\\ &\qquad \mathtt{witnessed\ p}))\end{aligned}$$

- We introduce an operation for **recalling** validity of predicates

$$\begin{aligned}\mathtt{recall} : \mathtt{p{:}pred\ state} \to \mathtt{MST\ unit}\ &(\mathtt{requires}\ (\lambda\, \mathtt{s_0}\,.\, \mathtt{witnessed\ p}))\\ &(\mathtt{ensures}\ (\lambda\, \mathtt{s_0\ \_\ s_1}\,.\, \mathtt{s_0} = \mathtt{s_1} \wedge \mathtt{p\ s_1}))\end{aligned}$$

Reasoning about monotonic state in F*

- We introduce a **logical capability**

$$\texttt{witnessed} : \texttt{pred state} \rightarrow \texttt{Type}$$

  together with a **weakening** principle

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

- We introduce an operation for **witnessing** stable predicates

$$\texttt{witness} : \texttt{p:pred state} \rightarrow \texttt{MST unit}\ (\texttt{requires}\ (\lambda\, \texttt{s}_0\,.\, \texttt{p s}_0 \wedge \texttt{stable p}))$$
$$(\texttt{ensures}\ (\lambda\, \texttt{s}_0\ \_\ \texttt{s}_1\,.\, \texttt{s}_0 = \texttt{s}_1 \wedge \texttt{witnessed p}))$$

- We introduce an operation for **recalling** validity of predicates

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

Reasoning about monotonic state in F*

- We introduce a **logical capability**

$$\texttt{witnessed} : \texttt{pred state} \to \texttt{Type}$$

  together with a **weakening** principle

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

- We introduce an operation for **witnessing** stable predicates

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

- We introduce an operation for **recalling** validity of predicates

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

Reasoning about monotonic state in F*

- We introduce a **logical capability**

$$\texttt{witnessed} : \texttt{pred state} \rightarrow \texttt{Type}$$

  together with a **weakening** principle

$$\begin{array}{l}\texttt{wk} : \texttt{p,q:pred state} \rightarrow \texttt{Lemma}\ (\texttt{requires}\ (\forall\, \texttt{s}\,.\,\texttt{p s} \implies \texttt{q s}))\\ \qquad\qquad\qquad\qquad\qquad\qquad (\texttt{ensures}\ (\texttt{witnessed p} \implies \texttt{witnessed q}))\end{array}$$

- We introduce an operation for **witnessing** stable predicates

$$\begin{array}{l}\texttt{witness} : \texttt{p:pred state} \rightarrow \texttt{MST unit}\ (\texttt{requires}\ (\lambda\, \texttt{s}_0\,.\,\texttt{p s}_0 \wedge \texttt{stable p}))\\ \qquad\qquad\qquad\qquad\qquad\qquad (\texttt{ensures}\ (\lambda\, \texttt{s}_0\ \_\ \texttt{s}_1\,.\,\texttt{s}_0 = \texttt{s}_1 \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{witnessed p}))\end{array}$$

- We introduce an operation for **recalling** validity of predicates

$$\begin{array}{l}\texttt{recall} : \texttt{p:pred state} \rightarrow \texttt{MST unit}\ (\texttt{requires}\ (\lambda\, \texttt{s}_0\,.\,\texttt{witnessed p}))\\ \qquad\qquad\qquad\qquad\qquad\qquad (\texttt{ensures}\ (\lambda\, \texttt{s}_0\ \_\ \texttt{s}_1\,.\,\texttt{s}_0 = \texttt{s}_1 \wedge \texttt{p s}_1))\end{array}$$

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F*
- Examples of monotonic state at work
- A glimpse of the meta-theory

The motivating example revisited

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

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

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

- **Monotonic counters** are analogous, with $\mathbb{N}$ and $\leq$

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

The motivating example revisited

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

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

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

- **Monotonic counters** are analogous, with $\mathbb{N}$ and $\leq$

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

The motivating example revisited

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

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

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

- **Monotonic counters** are analogous, with $\mathbb{N}$ and $\leq$

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

The motivating example revisited

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

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

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

- **Monotonic counters** are analogous, with $\mathbb{N}$ and $\leq$

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

The motivating example revisited

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

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

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

- **Monotonic counters** are analogous, with $\mathbb{N}$ and $\leq$

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

References: both typed and untyped

- We define **local state** using global state + monotonicity
- We define **heaps** as maps

  $\texttt{type heap} =$

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

  where

  $\quad \texttt{type cell} = \mathtt{Unused} : \mathtt{cell} \;|\; \mathtt{Used} : a{:}\mathrm{Type} \to v{:}\mathrm{a} \to t{:}\mathrm{tag} \to \mathtt{cell}$

  $\quad \texttt{type tag} \;\; = \mathtt{Typed} : \mathtt{tag} \;|\; \mathtt{Untyped} : live{:}\mathrm{bool} \to \mathtt{tag}$

- The **preorder** on heaps is given by

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

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

  $\quad | \; \mathtt{Used}\ \_\ \_\ (\mathtt{Untyped}\ l_0), \mathtt{Used}\ \_\ \_\ (\mathtt{Untyped}\ l_1) \to \neg(l_0) \implies \neg(l_1)$

  $\quad | \; \_, \_ \to \bot$

References: both typed and untyped

- We define **local state** using global state + monotonicity

- We define **heaps** as maps

```
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 → t:tag → cell
    type tag  = Typed : tag | Untyped : live:bool → tag
```

- The **preorder** on heaps is given by

```
  let rel (H h0 _) (H h1 _) = ∀id . match h0 id, h1 id with
    | Used a _ Typed, Used b _ Typed → a = b
    | Used _ _ (Untyped l0), Used _ _ (Untyped l1) → ¬(l0) ⟹ ¬(l1)
    | _, _ → ⊥
```

References: both typed and untyped

- We define **local state** using global state + monotonicity

- We define **heaps** as maps

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

  where

$$\texttt{type cell} = \texttt{Unused} : \texttt{cell} \mid \texttt{Used} : \texttt{a}{:}\mathrm{Type} \to \texttt{v}{:}\texttt{a} \to \texttt{t}{:}\texttt{tag} \to \texttt{cell}$$
$$\texttt{type tag} = \texttt{Typed} : \texttt{tag} \mid \texttt{Untyped} : \texttt{live}{:}\mathrm{bool} \to \texttt{tag}$$

- The **preorder** on heaps is given by

$$\texttt{let rel (H } \texttt{h}_0\ \_\texttt{) (H } \texttt{h}_1\ \_\texttt{)} = \forall\, \texttt{id}\,.\, \texttt{match } \texttt{h}_0\ \texttt{id}, \texttt{h}_1\ \texttt{id with}$$
$$\mid \texttt{Used a \_ Typed}, \texttt{Used b \_ Typed} \to \texttt{a} = \texttt{b}$$
$$\mid \texttt{Used \_ \_ (Untyped } \texttt{l}_0), \texttt{Used \_ \_ (Untyped } \texttt{l}_1) \to \neg(\texttt{l}_0) \implies \neg(\texttt{l}_1)$$
$$\mid \_, \_ \to \bot$$

References: both typed and untyped

- We define **local state** using global state + monotonicity

- We define **heaps** as maps

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

  where

$$\texttt{type cell} = \texttt{Unused} : \texttt{cell} \mid \texttt{Used} : \texttt{a}{:}\mathrm{Type} \to \texttt{v}{:}\texttt{a} \to \texttt{t}{:}\texttt{tag} \to \texttt{cell}$$
$$\texttt{type tag} = \texttt{Typed} : \texttt{tag} \mid \texttt{Untyped} : \texttt{live}{:}\texttt{bool} \to \texttt{tag}$$

- The **preorder** on heaps is given by

$$\texttt{let rel}\ (\texttt{H}\ \texttt{h}_0\ \_)\ (\texttt{H}\ \texttt{h}_1\ \_) = \forall\, \texttt{id}\,.\, \texttt{match}\ \texttt{h}_0\ \texttt{id}, \texttt{h}_1\ \texttt{id}\ \texttt{with}$$
$$\mid \texttt{Used a}\ \_\ \texttt{Typed}, \texttt{Used b}\ \_\ \texttt{Typed} \to \texttt{a} = \texttt{b}$$
$$\mid \texttt{Used}\ \_\ \_\ (\texttt{Untyped}\ \texttt{l}_0), \texttt{Used}\ \_\ \_\ (\texttt{Untyped}\ \texttt{l}_1) \to \neg(\texttt{l}_0) \implies \neg(\texttt{l}_1)$$
$$\mid \_, \_ \to \bot$$

References: both typed and untyped ctd.

- We define **local state** as global state + monotonicity
- We define **heaps** as maps

  $\texttt{type heap} =$

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

  where

  $\texttt{type cell} = \texttt{Unused} : \texttt{cell} \ | \ \texttt{Used} : \texttt{a}{:}\texttt{Type} \to \texttt{v}{:}\texttt{a} \to \texttt{t}{:}\texttt{tag} \to \texttt{cell}$

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

- **Typed references** are defined as

  $\texttt{abstract type ref t} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed}\ (\lambda\, \texttt{h}\,.\, \texttt{has\_used\_typed id t h})\}$

- **Untyped references** are defined as

  $\texttt{abstract type uref} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed}\ (\lambda\, \texttt{h}\,.\, \texttt{has\_used\_untyped\_live id h})\}$

References: both typed and untyped ctd.

- We define **local state** as global state + monotonicity
- We define **heaps** as maps

  $\texttt{type heap} =$

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

  where

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

  $\texttt{type tag} \;= \texttt{Typed} : \texttt{tag} \mid \texttt{Untyped} : \texttt{live}{:}\texttt{bool} \to \texttt{tag}$

- **Typed references** are defined as

  $\texttt{abstract type ref t} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed}\ (\lambda\, \texttt{h} \,.\, \texttt{has\_used\_typed id t h})\}$

- **Untyped references** are defined as

  $\texttt{abstract type uref} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed}\ (\lambda\, \texttt{h} \,.\, \texttt{has\_used\_untyped\_live id h})\}$

References: both typed and untyped ctd.

- We define **local state** as global state + monotonicity
- We define **heaps** as maps

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

  where

  $$\texttt{type cell} = \texttt{Unused} : \texttt{cell} \mid \texttt{Used} : \texttt{a}{:}\mathrm{Type} \to \texttt{v}{:}\texttt{a} \to \texttt{t}{:}\texttt{tag} \to \texttt{cell}$$
  $$\texttt{type tag} \;\; = \texttt{Typed} : \texttt{tag} \mid \texttt{Untyped} : \texttt{live}{:}\texttt{bool} \to \texttt{tag}$$

- **Typed references** are defined as

  $$\texttt{abstract type ref t} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed}\ (\lambda\, \texttt{h}\,.\, \texttt{has\_used\_typed id t h})\}$$

- **Untyped references** are defined as

  $$\texttt{abstract type uref} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed}\ (\lambda\, \texttt{h}\,.\, \texttt{has\_used\_untyped\_live id h})\}$$

References: typed and untyped ctd.

- The state actions for **typed references** use **witness** and **recall**
  - `let alloc t (v:t) : MST (ref t) ... = ...`
    - **get** the current heap (using global state `get`)
    - **create** a fresh ref., and **add** it to the heap
    - **put** the updated heap back (using global state `put`)
    - **witness** that the created ref. is in the heap
  - `let read t (r:ref t) : MST t ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap (using global state `get`)
    - **select** the given reference from the heap
  - `let write t (r:ref t) (v:t) : MST unit ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap (using global state `get`)
    - **update** the heap with the given value at the given ref.
    - **put** the updated heap back (using global state `put`)
- The actions for **untyped references** involve liveness preconditions

References: typed and untyped ctd.

- The state actions for **typed references** use **witness** and **recall**
  - `let alloc t (v:t) : MST (ref t) ... = ...`
    - **get** the current heap (using global state `get`)
    - **create** a fresh ref., and **add** it to the heap
    - **put** the updated heap back (using global state `put`)
    - **witness** that the created ref. is in the heap
  - `let read t (r:ref t) : MST t ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap (using global state `get`)
    - **select** the given reference from the heap
  - `let write t (r:ref t) (v:t) : MST unit ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap (using global state `get`)
    - **update** the heap with the given value at the given ref.
    - **put** the updated heap back (using global state `put`)
- The actions for **untyped references** involve liveness preconditions

References: typed and untyped ctd.

- The state actions for **typed references** use **witness** and **recall**
  - `let alloc t (v:t) : MST (ref t) ... = ...`
    - **get** the current heap (using global state `get`)
    - **create** a fresh ref., and **add** it to the heap
    - **put** the updated heap back (using global state `put`)
    - **witness** that the created ref. is in the heap
  - `let read t (r:ref t) : MST t ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap (using global state `get`)
    - **select** the given reference from the heap
  - `let write t (r:ref t) (v:t) : MST unit ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap (using global state `get`)
    - **update** the heap with the given value at the given ref.
    - **put** the updated heap back (using global state `put`)
- The actions for **untyped references** involve liveness preconditions

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

$\mathtt{type\ tag\ a} = \mathtt{Typed} : \mathtt{rel}{:}\mathtt{preorder\ a} \rightarrow \mathtt{tag\ a} \mid \mathtt{Untyped} : \mathtt{live}{:}\mathtt{bool} \rightarrow \mathtt{tag\ a}$

- The **global preorder** is a point-wise lifting of the individual ones

$$\begin{array}{l}
\mathtt{let\ rel}\ (\mathtt{H}\ \mathtt{h}_0\ _)\ (\mathtt{H}\ \mathtt{h}_1\ _) = \forall\, \mathtt{id}\,.\, \mathtt{match}\ \mathtt{h}_0\ \mathtt{id}, \mathtt{h}_1\ \mathtt{id}\ \mathtt{with} \\
\quad \mid \mathtt{Used}\ \mathtt{a}_0\ \mathtt{v}_0\ (\mathtt{Typed}\ \mathtt{rel}_0), \\
\quad\ \ \ \mathtt{Used}\ \mathtt{a}_1\ \mathtt{v}_1\ (\mathtt{Typed}\ \mathtt{rel}_1) \rightarrow \mathtt{a}_0 = \mathtt{a}_1 \wedge \mathtt{rel}_0 = \mathtt{rel}_1 \wedge \mathtt{rel}_0\ \mathtt{v}_0\ \mathtt{v}_1 \\
\quad \mid \ldots
\end{array}$$

- **Monotonic references** are then given as

$\mathtt{abstract\ type\ mref\ t\ rel} = \mathtt{id}{:}\mathbb{N}\{\mathtt{witnessed}\ (\lambda\, \mathtt{h}\,.\, \mathtt{has\_mref\ id\ t\ rel\ h})\}$

- State actions
  - The **write** action is constrained by `rel` of the given mref.
  - The **witness** and **recall** actions are given reference-wise

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

$\texttt{type tag a} = \texttt{Typed} : \texttt{rel:preorder a} \rightarrow \texttt{tag a} \mid \texttt{Untyped} : \texttt{live:bool} \rightarrow \texttt{tag a}$

- The **global preorder** is a point-wise lifting of the individual ones

$$\begin{array}{l}\texttt{let rel (H } h_0 \texttt{ \_) (H } h_1 \texttt{ \_)} = \forall\, \texttt{id} . \texttt{match } h_0 \texttt{ id}, h_1 \texttt{ id with} \\ \quad \mid \texttt{Used } a_0 \; v_0 \; (\texttt{Typed } rel_0), \\ \quad\;\; \texttt{Used } a_1 \; v_1 \; (\texttt{Typed } rel_1) \rightarrow a_0 = a_1 \wedge rel_0 = rel_1 \wedge rel_0 \; v_0 \; v_1 \\ \quad \mid \ldots \end{array}$$

- **Monotonic references** are then given as

$\texttt{abstract type mref t rel} = \texttt{id:}\mathbb{N}\{\texttt{witnessed } (\lambda\, \texttt{h} . \texttt{has\_mref id t rel h})\}$

- State actions
  - The **write** action is constrained by `rel` of the given mref.
  - The **witness** and **recall** actions are given reference-wise

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

$$\texttt{type tag a} = \texttt{Typed} : \texttt{rel:preorder a} \rightarrow \texttt{tag a} \mid \texttt{Untyped} : \texttt{live:bool} \rightarrow \texttt{tag a}$$

- The **global preorder** is a point-wise lifting of the individual ones

$$\begin{array}{l} \texttt{let rel (H } h_0 \texttt{ \_) (H } h_1 \texttt{ \_)} = \forall\, \texttt{id} .\, \texttt{match } h_0 \texttt{ id}, h_1 \texttt{ id with} \\ \quad \mid \texttt{Used } a_0 \; v_0 \; (\texttt{Typed } rel_0), \\ \quad\;\; \texttt{Used } a_1 \; v_1 \; (\texttt{Typed } rel_1) \rightarrow a_0 = a_1 \wedge rel_0 = rel_1 \wedge rel_0 \; v_0 \; v_1 \\ \quad \mid \ldots \end{array}$$

- **Monotonic references** are then given as

$$\texttt{abstract type mref t rel} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed } (\lambda\, \texttt{h} .\, \texttt{has\_mref id t rel h})\}$$

- State actions
  - The **write** action is constrained by `rel` of the given mref.
  - The **witness** and **recall** actions are given reference-wise

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

$\texttt{type tag a} = \texttt{Typed} : \texttt{rel:preorder a} \rightarrow \texttt{tag a} \mid \texttt{Untyped} : \texttt{live:bool} \rightarrow \texttt{tag a}$

- The **global preorder** is a point-wise lifting of the individual ones

$$
\begin{array}{l}
\texttt{let rel } (\texttt{H } \texttt{h}_0 \ \_) \ (\texttt{H } \texttt{h}_1 \ \_) = \forall \, \texttt{id} \, . \, \texttt{match } \texttt{h}_0 \ \texttt{id}, \texttt{h}_1 \ \texttt{id} \texttt{ with} \\
\quad \mid \texttt{Used } \texttt{a}_0 \ \texttt{v}_0 \ (\texttt{Typed } \texttt{rel}_0), \\
\quad \ \ \ \texttt{Used } \texttt{a}_1 \ \texttt{v}_1 \ (\texttt{Typed } \texttt{rel}_1) \rightarrow \texttt{a}_0 = \texttt{a}_1 \wedge \texttt{rel}_0 = \texttt{rel}_1 \wedge \texttt{rel}_0 \ \texttt{v}_0 \ \texttt{v}_1 \\
\quad \mid \ \ldots
\end{array}
$$

- **Monotonic references** are then given as

$\texttt{abstract type mref t rel} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed } (\lambda \, \texttt{h} \, . \, \texttt{has\_mref id t rel h})\}$

- State actions
  - The **write** action is constrained by `rel` of the given mref.
  - The **witness** and **recall** actions are given reference-wise

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

$\texttt{type tag a} = \texttt{Typed} : \texttt{rel:preorder a} \rightarrow \texttt{tag a} \mid \texttt{Untyped} : \texttt{live:bool} \rightarrow \texttt{tag a}$

- The **global preorder** is a point-wise lifting of the individual ones

$$
\begin{aligned}
&\texttt{let rel } (\texttt{H } \texttt{h}_0 \_)\ (\texttt{H } \texttt{h}_1 \_) = \forall \texttt{id}.\ \texttt{match } \texttt{h}_0 \texttt{ id}, \texttt{h}_1 \texttt{ id with} \\
&\quad \mid \texttt{Used } \texttt{a}_0\ \texttt{v}_0\ (\texttt{Typed } \texttt{rel}_0), \\
&\quad\ \ \ \texttt{Used } \texttt{a}_1\ \texttt{v}_1\ (\texttt{Typed } \texttt{rel}_1) \rightarrow \texttt{a}_0 = \texttt{a}_1 \wedge \texttt{rel}_0 = \texttt{rel}_1 \wedge \texttt{rel}_0\ \texttt{v}_0\ \texttt{v}_1 \\
&\quad \mid \ldots
\end{aligned}
$$

- **Monotonic references** are then given as

$\texttt{abstract type mref t rel} = \texttt{id}:\mathbb{N}\{\texttt{witnessed } (\lambda \texttt{h}.\ \texttt{has\_mref id t rel h})\}$

- State actions
  - The **write** action is constrained by `rel` of the given mref.
  - The **witness** and **recall** actions are given reference-wise

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F*
- Examples of monotonic state at work
- A glimpse of the meta-theory

A glimpse of the meta-theory

- We formalize MST in a small dependently typed CBV calculus

$$
\begin{aligned}
t &::= \mathsf{state} \mid x{:}t_1 \to \textbf{Tot}\ t_2 \mid x{:}t_1 \to \textbf{MST}\ t_2\ (s.\varphi_{\mathsf{pre}})\ (s.y.s'.\varphi_{\mathsf{post}}) \mid \ \ldots \\
e &::= \mathsf{get} \mid \mathsf{put}\ v \mid \mathsf{witness}\ s.\varphi \mid \mathsf{recall}\ s.\varphi \mid \ \ldots \\
\varphi &::= \mathsf{rel}\ v_1\ v_2 \mid \mathsf{witnessed}\ s.\varphi \mid \ \ldots
\end{aligned}
$$

- **Consistency** and **props. of the logic** via seq. calc. and cut-adm.

- **Operational semantics** on configurations (e, σ, W)

$$
\begin{aligned}
(\mathsf{witness}\ s.\varphi, \sigma, W) &\rightsquigarrow (\mathsf{return}\ (), \sigma, W \cup \{s.\varphi\}) \\
(\mathsf{recall}\ s.\varphi, \sigma, W) &\rightsquigarrow (\mathsf{return}\ (), \sigma, W)
\end{aligned}
$$

- **Total correctness** via progress, preservation, and SN

$$
\begin{array}{c}
\vdash e : \textbf{MST}\ t\ (s.\varphi_{\mathsf{pre}})\ (s.x.s'.\varphi_{\mathsf{post}}) \\
\mathsf{witnessed}\ W \vdash \varphi_{\mathsf{pre}}[\sigma/s]
\end{array}
\implies
\begin{array}{c}
(e, \sigma, W) \rightsquigarrow^* (\mathsf{return}\ v, \sigma', W') \quad \vdash v : t \\
W \subseteq W' \quad \mathsf{witnessed}\ W' \vdash \mathsf{rel}\ \sigma\ \sigma' \\
\mathsf{witnessed}\ W' \vdash \varphi_{\mathsf{post}}[\sigma/s, v/x, \sigma'/s']
\end{array}
$$

A glimpse of the meta-theory

- We formalize MST in a small dependently typed CBV calculus

$$
\begin{aligned}
t &::= \mathsf{state} \mid x{:}t_1 \to \mathbf{Tot}\ t_2 \mid x{:}t_1 \to \mathbf{MST}\ t_2\ (s.\varphi_{\mathsf{pre}})\ (s.y.s'.\varphi_{\mathsf{post}}) \mid \ldots \\
e &::= \mathsf{get} \mid \mathsf{put}\ v \mid \mathsf{witness}\ s.\varphi \mid \mathsf{recall}\ s.\varphi \mid \ldots \\
\varphi &::= \mathsf{rel}\ v_1\ v_2 \mid \mathsf{witnessed}\ s.\varphi \mid \ldots
\end{aligned}
$$

  - **Consistency** and **props. of the logic** via seq. calc. and cut-adm.

  - **Operational semantics** on configurations (e, σ, W)

$$
\begin{aligned}
(\mathsf{witness}\ s.\varphi, \sigma, W) &\leadsto (\mathsf{return}\ (), \sigma, W \cup \{s.\varphi\}) \\
(\mathsf{recall}\ s.\varphi, \sigma, W) &\leadsto (\mathsf{return}\ (), \sigma, W)
\end{aligned}
$$

  - **Total correctness** via progress, preservation, and SN

$$
\begin{array}{c}
\vdash e : \mathbf{MST}\ t\ (s.\varphi_{\mathsf{pre}})\ (s.x.s'.\varphi_{\mathsf{post}}) \\
\mathsf{witnessed}\ W \vdash \varphi_{\mathsf{pre}}[\sigma/s]
\end{array}
\implies
\begin{array}{c}
(e, \sigma, W) \leadsto^* (\mathsf{return}\ v, \sigma', W') \quad \vdash v : t \\
W \subseteq W' \quad \mathsf{witnessed}\ W' \vdash \mathsf{rel}\ \sigma\ \sigma' \\
\mathsf{witnessed}\ W' \vdash \varphi_{\mathsf{post}}[\sigma/s, v/x, \sigma'/s']
\end{array}
$$

A glimpse of the meta-theory

- We formalize MST in a small dependently typed CBV calculus

$$
\begin{aligned}
t &::= \text{state} \mid x{:}t_1 \to \textbf{Tot}\ t_2 \mid x{:}t_1 \to \textbf{MST}\ t_2\ (s.\varphi_{\text{pre}})\ (s.y.s'.\varphi_{\text{post}}) \mid \ \ldots \\
e &::= \text{get} \mid \text{put}\ v \mid \text{witness}\ s.\varphi \mid \text{recall}\ s.\varphi \mid \ \ldots \\
\varphi &::= \text{rel}\ v_1\ v_2 \mid \text{witnessed}\ s.\varphi \mid \ \ldots
\end{aligned}
$$

  - **Consistency** and **props. of the logic** via seq. calc. and cut-adm.

  - **Operational semantics** on configurations (e, σ, W)

$$
\begin{aligned}
(\text{witness}\ s.\varphi, \sigma, W) &\rightsquigarrow (\text{return}\ (), \sigma, W \cup \{s.\varphi\}) \\
(\text{recall}\ s.\varphi, \sigma, W) &\rightsquigarrow (\text{return}\ (), \sigma, W)
\end{aligned}
$$

  - **Total correctness** via progress, preservation, and SN

$$
\begin{array}{c}
\vdash e : \textbf{MST}\ t\ (s.\varphi_{\text{pre}})\ (s.x.s'.\varphi_{\text{post}}) \\
\text{witnessed}\ W \vdash \varphi_{\text{pre}}[\sigma/s]
\end{array}
\implies
\begin{array}{c}
(e, \sigma, W) \rightsquigarrow^* (\text{return}\ v, \sigma', W') \quad \vdash v : t \\
W \subseteq W' \quad \text{witnessed}\ W' \vdash \text{rel}\ \sigma\ \sigma' \\
\text{witnessed}\ W' \vdash \varphi_{\text{post}}[\sigma/s, v/x, \sigma'/s']
\end{array}
$$

A glimpse of the meta-theory

- We formalize `MST` in a small dependently typed CBV calculus

$$
\begin{aligned}
t &::= \mathsf{state} \mid x{:}t_1 \to \mathbf{Tot}\ t_2 \mid x{:}t_1 \to \mathbf{MST}\ t_2\ (s.\varphi_{\mathsf{pre}})\ (s.y.s'.\varphi_{\mathsf{post}}) \mid \ \ldots \\
e &::= \mathsf{get} \mid \mathsf{put}\ v \mid \mathsf{witness}\ s.\varphi \mid \mathsf{recall}\ s.\varphi \mid \ \ldots \\
\varphi &::= \mathsf{rel}\ v_1\ v_2 \mid \mathsf{witnessed}\ s.\varphi \mid \ \ldots
\end{aligned}
$$

  - **Consistency** and **props. of the logic** via seq. calc. and cut-adm.

  - **Operational semantics** on configurations (e, σ, W)

$$
\begin{aligned}
(\mathsf{witness}\ s.\varphi, \sigma, W) &\quad\rightsquigarrow\quad (\mathsf{return}\ (), \sigma, W \cup \{s.\varphi\}) \\
(\mathsf{recall}\ s.\varphi, \sigma, W) &\quad\rightsquigarrow\quad (\mathsf{return}\ (), \sigma, W)
\end{aligned}
$$

  - **Total correctness** via progress, preservation, and SN

$$
\begin{array}{c}
\vdash e : \mathbf{MST}\ t\ (s.\varphi_{\mathsf{pre}})\ (s.x.s'.\varphi_{\mathsf{post}}) \\
\mathsf{witnessed}\ W \vdash \varphi_{\mathsf{pre}}[\sigma/s]
\end{array}
\implies
\begin{array}{c}
(e, \sigma, W) \rightsquigarrow^* (\mathsf{return}\ v, \sigma', W') \quad \vdash v : t \\
W \subseteq W' \quad \mathsf{witnessed}\ W' \vdash \mathsf{rel}\ \sigma\ \sigma' \\
\mathsf{witnessed}\ W' \vdash \varphi_{\mathsf{post}}[\sigma/s, v/x, \sigma'/s']
\end{array}
$$

Conclusion

- In conclusion
  - making use of monotonicity is quite useful in verification
  - using monotonicity can be distilled into a simple interface
  - useful for both programming (refs.) and verification (crypto,TLS)

- Not in this talk (see the draft paper on arXiv)
  - temporarily **escaping the preorder** via snapshots
  - **revealing the representation** via selective monadic reification

- Future work
  - extending F* with indexed effects
  - combining preorders (e.g., ala graded monads)
  - modal aspects of `witnessed p`
  - connections with other works, e.g., Iris and [Pilkiewicz,Pottier'11]

Conclusion

- In conclusion
  - making use of monotonicity is quite useful in verification
  - using monotonicity can be distilled into a simple interface
  - useful for both programming (refs.) and verification (crypto,TLS)

- Not in this talk (see the draft paper on arXiv)
  - temporarily **escaping the preorder** via snapshots
  - **revealing the representation** via selective monadic reification

- Future work
  - extending F* with indexed effects
  - combining preorders (e.g., ala graded monads)
  - modal aspects of `witnessed p`
  - connections with other works, e.g., Iris and [Pilkiewicz,Pottier'11]

Conclusion

- In conclusion
  - making use of monotonicity is quite useful in verification
  - using monotonicity can be distilled into a simple interface
  - useful for both programming (refs.) and verification (crypto,TLS)

- Not in this talk (see the draft paper on arXiv)
  - temporarily **escaping the preorder** via snapshots
  - **revealing the representation** via selective monadic reification

- Future work
  - extending F* with indexed effects
  - combining preorders (e.g., ala graded monads)
  - modal aspects of `witnessed p`
  - connections with other works, e.g., Iris and [Pilkiewicz,Pottier'11]

Thank you!

Questions?

Recalling a Witness:
Foundations and Applications of Monotonic State

(arXiv:1707.02466)