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 solution
- Adding monotonic state to F*
- Example uses of monotonicity (as used in F*)
- A glimpse of the meta-theory

Outline

- Monotonic state and program verification by example
- Key ideas behind our solution
- Adding monotonic state to F*
- Example uses of monotonicity (as used in F*)
- 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 $\texttt{w}$
    - 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**

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

- 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
    - 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**

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

- 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
    - 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**"

Other, more substantial examples

- To come later in this talk
  - reasoning about **monotonic counters**
  - using monotonicity to implement **typed** and **untyped references**
  - more flexibility with **monotonic references**

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

Other, more substantial examples

- To come later in this talk
  - reasoning about **monotonic counters**
  - using monotonicity to implement **typed** and **untyped references**
  - more flexibility with **monotonic references**

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

Other, more substantial examples

- To come later in this talk
  - reasoning about **monotonic counters**
  - using monotonicity to implement **typed** and **untyped references**
  - more flexibility with **monotonic references**

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

Outline

- Monotonic state and program verification by example
- Key ideas behind our solution
- Adding monotonic state to F*
- Example uses of monotonicity (as used in F*)
- 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′
- We provide a **simple**, yet **general interface** for monotonicity

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 $\mathtt{p}\ \mathtt{s}$ in some state $\mathtt{s}$
  - operation to **recall** the validity of $\mathtt{p}\ \mathtt{s}'$ in a future state $\mathtt{s}'$
- We provide a **simple**, yet **general interface** for monotonicity

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′
- We provide a **simple**, yet **general interface** for monotonicity

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′
- We provide a **simple**, yet **general interface** for monotonicity

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′
- We provide a **simple**, yet **general interface** for monotonicity

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

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

  - 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′

- We provide a **simple**, yet **general interface** for monotonicity

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 $\mathtt{p}\ \mathtt{s}$ in some state $\mathtt{s}$
  - operation to **recall** the validity of $\mathtt{p}\ \mathtt{s}'$ in a future state $\mathtt{s}'$
- We provide a **simple**, yet **general interface** for monotonicity

Outline

- Monotonic state and program verification by example
- Key ideas behind our solution
- Adding monotonic state to F*
- Example uses of monotonicity (as used in F*)
- 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

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

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

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 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

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

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

Reasoning about monotonic state in F*

- We 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

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

  - 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

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

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

Reasoning about monotonic state in F*

- We 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

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

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

Reasoning about monotonic state in F*

- We introduce a **logical capability**

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

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**

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

  together with a **weakening** principle

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

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

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

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

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

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

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

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

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

Outline

- Monotonic state and program verification by example
- Key ideas behind our solution
- Adding monotonic state to F*
- Example uses of monotonicity (as used in F*)
- A glimpse of the meta-theory

The motivating example revisited

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

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

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

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

  - For any other `w`, wrapping

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

    around the program is handled similarly easily

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

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

The motivating example revisited

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

  `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`

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

  - For any other `w`, wrapping

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

    around the program is handled similarly easily

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

  `create 0; incr(); witness` $(\lambda\, \texttt{c} \,.\, \texttt{c} > 0)$`; c_p(); 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 $\leq$

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

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**

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

  - 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 $\leq$

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

References: both typed and untyped

- We define **local state** using 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

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

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

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

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

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

  $\quad | \ \texttt{Used \_ \_ (Untyped}\ \texttt{l}_0), \texttt{Used \_ \_ (Untyped}\ \texttt{l}_1) \to \neg(\texttt{l}_0) \implies \neg(\texttt{l}_1)$

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

References: both typed and untyped

- We define **local state** using 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

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

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

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

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

  $\qquad \mid \texttt{Used a}\ \_\ \texttt{Typed}, \texttt{Used b}\ \_\ \texttt{Typed} \to \texttt{a} = \texttt{b}$

  $\qquad \mid \texttt{Used}\ \_\ \_\ (\texttt{Untyped l}_0), \texttt{Used}\ \_\ \_\ (\texttt{Untyped l}_1) \to \neg(\texttt{l}_0) \implies \neg(\texttt{l}_1)$

  $\qquad \mid \_, \_ \to \bot$

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

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

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

- 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

$$\texttt{type heap} =$$

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

  where

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

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

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

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

$$\mid \texttt{Used a}\ \_\ \texttt{Typed}, \texttt{Used b}\ \_\ \texttt{Typed} \to \texttt{a} = \texttt{b}$$

$$\mid \texttt{Used}\ \_\ \_\ \texttt{(Untyped l}_0\texttt{)}, \texttt{Used}\ \_\ \_\ \texttt{(Untyped l}_1\texttt{)} \to \neg(\texttt{l}_0) \implies \neg(\texttt{l}_1)$$

$$\mid \_, \_ \to \bot$$

References: both typed and untyped ctd.

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

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

where

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

- **Typed references** are defined as

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

- **Untyped references** are defined as

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

References: both typed and untyped ctd.

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

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

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

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

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

  where

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

$$\mathtt{type\ tag} \;= \mathtt{Typed} : \mathtt{tag} \mid \mathtt{Untyped} : \mathtt{live}{:}\mathtt{bool} \rightarrow \mathtt{tag}$$

- **Typed references** are defined as

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

- **Untyped references** are defined as

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

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

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

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

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

- **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

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

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

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

- **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

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

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

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

- **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

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

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

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

- **Monotonic references** are then given as

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

- 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

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

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

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

- **Monotonic references** are then given as

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

- 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 solution
- Adding monotonic state to F*
- Example uses of monotonicity (as used in F*)
- 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)

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

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

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

A glimpse of the meta-theory

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

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

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

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

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

  - **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

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

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

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

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

  - **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)