Leveraging monotonic state in F*

Danel Ahman @ INRIA Paris

joint work with

Cătălin Hrițcu and Kenji Maillard @ INRIA Paris

Cédric Fournet, Aseem Rastogi, and Nikhil Swamy @ MSR

EUTypes Working Meeting

January 23, 2018

Global state + monotonicity is really useful!

Its essence can be captured very neatly!

Outline

- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see POPL'18 paper)
- First steps in mon. reification and reflection (see POPL'18 paper)
- Meta-theory and correctness results (see POPL'18 paper)

Outline

- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see POPL'18 paper)
- First steps in mon. reification and reflection (see POPL'18 paper)
- Meta-theory and correctness results (see POPL'18 paper)

Monotonicity in verification

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

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

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

  $\{\lambda\, s\, .\, v \in s\}$ complex_procedure() $\{\lambda\, s\, .\, v \in s\}$

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

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

Monotonicity in verification

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

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

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

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

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

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

Monotonicity in verification

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

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

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

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

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

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

Monotonicity in verification

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

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

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

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

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

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

Monotonicity in programming

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

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

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

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

Monotonicity in programming

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

  even if you don't realise it!

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

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

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

Monotonicity in programming

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

  even if you don't realise it!

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

- Correctness relies on **monotonicity**!

  1) Allocation **stores** an `a`-typed value in the heap

  2) Writes **don't change type** and there is **no deallocation**

  3) So, given a ref. `r`, it is **guaranteed to point** to an `a`-typed value

- Baked into the memory models of most languages

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

Monotonicity in programming

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

  even if you don't realise it!

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

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

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

Monotonicity is really useful!

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

Monotonicity is really useful!

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

Monotonicity is really useful!

- In this talk
  - our **motivating example** and **monotonic counters**
  - **typed references** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)

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

Outline

- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see POPL'18 paper)
- First steps in mon. reification and reflection (see POPL'18 paper)
- Meta-theory and correctness results (see POPL'18 paper)

Key ideas behind our general framework

- We make use of **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** rel on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program e is **monotonic** (wrt. rel) when
$$\forall\, s\, e'\, s'.\ (e, s) \leadsto^{*} (e', s') \implies \mathtt{rel}\ s\ s'$$
  - a stateful predicate p is **stable** (wrt. rel) when
$$\forall\, s\, s'.\ \mathtt{p}\ s\ \wedge\ \mathtt{rel}\ s\ s' \implies \mathtt{p}\ s'$$
- **Our solution:** extend Hoare-style program logics (e.g., F*) with
  - a means to **witness** the validity of p s in some state s
  - a means for turning a p into a **state-independent proposition**
  - a means to **recall** the validity of p s′ in any future state s′
- Provides a **unifying account** of the existing *ad hoc* uses in F*

Key ideas behind our general framework

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

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

  - a stateful predicate `p` 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
  - a means to **witness** the validity of `p` s in some state s
  - a means for turning a `p` into a **state-independent proposition**
  - a means to **recall** the validity of `p` s′ in any future state s′
- Provides a **unifying account** of the existing *ad hoc* uses in F*

Key ideas behind our general framework

- We make use of **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** `rel` on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program e is **monotonic** (wrt. `rel`) when
$$\forall\, s\, e'\, s'.\ (e, s) \rightsquigarrow^* (e', s') \implies \mathtt{rel}\ s\ s'$$
  - a stateful predicate p is **stable** (wrt. `rel`) when
$$\forall\, s\, s'.\ p\ s\ \wedge\ \mathtt{rel}\ s\ s' \implies p\ s'$$
- **Our solution:** extend Hoare-style program logics (e.g., F*) with
  - a means to **witness** the validity of p s in some state s
  - a means for turning a p into a **state-independent proposition**
  - a means to **recall** the validity of p s′ in any future state s′
- Provides a **unifying account** of the existing *ad hoc* uses in F*

Key ideas behind our general framework

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

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

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

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

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

Key ideas behind our general framework

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

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

  - a stateful predicate `p` 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
  - a means to **witness** the validity of p s in some state s
  - a means for turning a p into a **state-independent proposition**
  - a means to **recall** the validity of p s′ in any future state s′
- Provides a **unifying account** of the existing *ad hoc* uses in F*

Key ideas behind our general framework

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

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

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

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

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

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

Key ideas behind our general framework

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

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

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

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

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

Outline

- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see POPL'18 paper)
- First steps in mon. reification and reflection (see POPL'18 paper)
- Meta-theory and correctness results (see POPL'18 paper)

Recap: Ordinary global state in F*

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

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

  where

$$\mathtt{pre} : \mathtt{state} \to \mathtt{Type}_0 \qquad \mathtt{post} : \mathtt{state} \to \mathtt{t} \to \mathtt{state} \to \mathtt{Type}_0$$

- ST is an abstract pre-postcondition refinement of

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

- The global state **actions** have types

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

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

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

Recap: Ordinary global state in F*

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

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

  where

$$\mathtt{pre} : \mathtt{state} \to \mathtt{Type}_0 \qquad \mathtt{post} : \mathtt{state} \to \mathtt{t} \to \mathtt{state} \to \mathtt{Type}_0$$

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

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

- The global state **actions** have types

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

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

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

Recap: Ordinary global state in F*

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

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

  where

$$\mathtt{pre} : \mathtt{state} \to \mathtt{Type}_0 \qquad \mathtt{post} : \mathtt{state} \to \mathtt{t} \to \mathtt{state} \to \mathtt{Type}_0$$

- ST is an abstract pre-postcondition refinement of

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

- The global state **actions** have types

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

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

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

Recap: Ordinary global state in F*

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

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

  where

$$\mathtt{pre} : \mathtt{state} \rightarrow \mathtt{Type}_0 \qquad \mathtt{post} : \mathtt{state} \rightarrow \mathtt{t} \rightarrow \mathtt{state} \rightarrow \mathtt{Type}_0$$

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

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

- The global state **actions** have types

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

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

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

New: Monotonic global state in F*

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

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

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

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

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

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

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

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

New: Monotonic global state in F*

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

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

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

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

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

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

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

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

New: Monotonic global state in F*

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

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

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

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

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

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

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

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

New: Monotonic global state in F*

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

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

- The **get** action is typed as in $\mathtt{ST}$

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

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

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

- So intuitively, $\mathtt{MST}$ is an **abstract** pre-postcondition refinement of

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

New: Monotonic global state in F*

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

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

- The **get** action is typed as in $\mathtt{ST}$

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

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

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

- So intuitively, $\mathtt{MST}$ is an **abstract** pre-postcondition refinement of

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

New: Recalling a Witness

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

$$\mathtt{witnessed} : (\mathtt{state} \to \mathtt{Type_0}) \to \mathtt{Type_0}$$

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

$$\mathtt{wk} : \mathtt{p,q}{:}(\mathtt{state} \to \mathtt{Type_0}) \to \mathtt{Lemma}\ (\mathtt{requires}\ (\forall\, \mathtt{s}\,.\, \mathtt{p\ s} \implies \mathtt{q\ s}))$$
$$(\mathtt{ensures}\ (\mathtt{witnessed\ p} \implies \mathtt{witnessed\ q}))$$

- Intuitively, a lot like the **necessity modality** $\Box$

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

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

New: Recalling a Witness

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

$$\texttt{witnessed} : (\texttt{state} \to \texttt{Type}_0) \to \texttt{Type}_0$$

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

$$\texttt{wk} : \texttt{p,q}{:}(\texttt{state} \to \texttt{Type}_0) \to \texttt{Lemma}\ (\texttt{requires}\ (\forall\, \texttt{s}\,.\,\texttt{p s} \implies \texttt{q s}))\ (\texttt{ensures}\ (\texttt{witnessed p} \implies \texttt{witnessed q}))$$

- Intuitively, a lot like the **necessity modality** $\Box$

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

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

New: Recalling a Witness

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

$$\mathtt{witnessed} : (\mathtt{state} \to \mathtt{Type}_0) \to \mathtt{Type}_0$$

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

$$\mathtt{wk} : \mathtt{p},\mathtt{q}{:}(\mathtt{state} \to \mathtt{Type}_0) \to \mathtt{Lemma}\ (\mathtt{requires}\ (\forall\, \mathtt{s}\,.\, \mathtt{p}\ \mathtt{s} \implies \mathtt{q}\ \mathtt{s}))$$
$$(\mathtt{ensures}\ (\mathtt{witnessed}\ \mathtt{p} \implies \mathtt{witnessed}\ \mathtt{q}))$$

- Intuitively, a lot like the **necessity modality** $\Box$

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

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

New: Recalling a Witness

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

$$\texttt{witnessed} : (\texttt{state} \to \texttt{Type}_0) \to \texttt{Type}_0$$

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

$$\begin{aligned}\texttt{wk} : \texttt{p},\texttt{q}{:}(\texttt{state} \to \texttt{Type}_0) \to \texttt{Lemma}\ &(\texttt{requires}\ (\forall\, \texttt{s}\,.\, \texttt{p s} \implies \texttt{q s}))\\ &(\texttt{ensures}\ (\texttt{witnessed p} \implies \texttt{witnessed q}))\end{aligned}$$

- Intuitively, a lot like the **necessity modality** $\Box$

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

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

New: Recalling a Witness

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

$$\texttt{witnessed} : (\texttt{state} \rightarrow \texttt{Type}_0) \rightarrow \texttt{Type}_0$$

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

$$\begin{aligned}\texttt{wk} : \texttt{p,q}{:}(\texttt{state} \rightarrow \texttt{Type}_0) \rightarrow \texttt{Lemma}\ &(\texttt{requires}\ (\forall\, \texttt{s}\,.\,\texttt{p s} \implies \texttt{q s}))\\ &(\texttt{ensures}\ (\texttt{witnessed p} \implies \texttt{witnessed q}))\end{aligned}$$

- Intuitively, a lot like the **necessity modality** $\Box$

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

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

New: Recalling a Witness

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

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

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

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

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

New: Recalling a Witness

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

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

$$
\begin{aligned}
\mathtt{witness} \;:\;\; & p{:}(\mathtt{state} \to \mathtt{Type}_0) \\
& \to \mathtt{MST}\ \mathtt{unit}\ (\mathtt{requires}\ (\lambda\, s_0 \,.\, p\ \text{`}\mathtt{stable\_from}\text{`}\ s_0)) \\
& \qquad\qquad\qquad (\mathtt{ensures}\ (\lambda\, s_0\, \_\, s_1 \,.\, s_0 = s_1 \wedge \mathtt{witnessed}\ p))
\end{aligned}
$$

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

$$
\begin{aligned}
\mathtt{recall} \;:\;\; & p{:}(\mathtt{state} \to \mathtt{Type}_0) \\
& \to \mathtt{MST}\ \mathtt{unit}\ (\mathtt{requires}\ (\lambda\, \_ \,.\, \mathtt{witnessed}\ p)) \\
& \qquad\qquad\qquad (\mathtt{ensures}\ (\lambda\, s_0\, \_\, s_1 \,.\, s_0 = s_1 \wedge p\ \text{`}\mathtt{stable\_from}\text{`}\ s_1))
\end{aligned}
$$

New: Recalling a Witness

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

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

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

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

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

Outline

- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see POPL'18 paper)
- First steps in mon. reification and reflection (see POPL'18 paper)
- Meta-theory and correctness results (see POPL'18 paper)

The motivating example revisited

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

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

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

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

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

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

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

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

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

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

The motivating example revisited

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

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

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

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

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

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

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

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

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

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

The motivating example revisited

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

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

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

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

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

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

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

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

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

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

The motivating example revisited

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

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

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

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

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

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

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

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

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

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

The motivating example revisited

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

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

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

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

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

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

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

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

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

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

ML-style typed references (local state)

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

  $\mathtt{type\ heap} =$

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

  where

  $\mathtt{type\ cell} =$

  $\quad | \ \mathtt{Unused} : \mathtt{cell}$

  $\quad | \ \mathtt{Used} : \mathtt{a}{:}\mathtt{Type}_0 \to \mathtt{v}{:}\mathtt{a} \to \mathtt{cell}$

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

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

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

  $\quad | \ \mathtt{Unused}\,, \mathtt{Used}\ \_\ \_\ \to\ \top$

  $\quad | \ \mathtt{Unused}\,, \mathtt{Unused}\ \to\ \top$

  $\quad | \ \mathtt{Used}\ \_\ \_\,, \mathtt{Unused}\ \to\ \bot$

ML-style typed references (local state)

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

$$\texttt{type heap} =$$

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

where

$$\texttt{type cell} =$$

$$\mid \texttt{Unused} : \texttt{cell}$$

$$\mid \texttt{Used} : \texttt{a}{:}\mathrm{Type}_0 \to \texttt{v}{:}\texttt{a} \to \texttt{cell}$$

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

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

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

$$\mid \texttt{Unused}\,,\, \texttt{Used}\ \_\ \_ \;\to\; \top$$

$$\mid \texttt{Unused}\,,\, \texttt{Unused} \;\to\; \top$$

$$\mid \texttt{Used}\ \_\ \_\,,\, \texttt{Unused} \;\to\; \bot$$

ML-style typed references (local state)

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

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

  where

  $$\texttt{type cell} =$$
  $$\mid \mathtt{Unused} : \mathtt{cell}$$
  $$\mid \mathtt{Used} : \mathtt{a}{:}\mathrm{Type}_0 \to \mathtt{v}{:}\mathtt{a} \to \mathtt{cell}$$

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

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

ML-style typed references (local state)

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

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

- Next, we define the type of **references** using monotonicity

$$\texttt{abstract type ref a} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed}\ (\lambda\,\texttt{h}\,.\,\texttt{contains h id a})\}$$

  where

```
let contains (H h _) id a =
  match h id with
    | Used b _  →  a = b
    | Unused    →  ⊥
```

- Important: `contains` is **stable** wrt. `heap_inclusion`

ML-style typed references (local state)

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

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

- Next, we define the type of **references** using monotonicity

  $\texttt{abstract type ref a} = \texttt{id}{:}\mathbb{N}\{\texttt{witnessed}\ (\lambda\,\texttt{h}\,.\,\texttt{contains h id a})\}$

  where

```
let contains (H h _) id a =
  match h id with
    | Used b _  →  a = b
    | Unused  →  ⊥
```

- Important: contains is **stable** wrt. heap_inclusion

ML-style typed references (local state)

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

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

- Next, we define the type of **references** using monotonicity

```
abstract type ref a = id:ℕ{witnessed (λ h . contains h id a)}
```

where

```
let contains (H h _) id a =
  match h id with
    | Used b _ → a = b
    | Unused → ⊥
```

- Important: `contains` is **stable** wrt. `heap_inclusion`

ML-style typed references (local state)

- Finally, we define `MLST`'s **actions** using `MST`'s actions
  - `let alloc (a:Type`$_0$`) (v:a) : MLST (ref a) ... = ...`
    - **get** the current heap
    - **create** a fresh ref., and **add** it to the heap
    - **put** the updated heap back
    - **witness** that the created ref. is in the heap
  - `let read (r:ref a) : MLST t ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap
    - **select** the given reference from the heap
  - `let write (r:ref a) (v:a) : MLST unit ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap
    - **update** the heap with the given value at the given ref.
    - **put** the updated heap back

ML-style typed references (local state)

- Finally, we define `MLST`'s **actions** using `MST`'s actions
  - `let alloc` (a:Type$_0$) (v:a) : `MLST (ref a) ... = ...`
    - **get** the current heap
    - **create** a fresh ref., and **add** it to the heap
    - **put** the updated heap back
    - **witness** that the created ref. is in the heap
  - `let read (r:ref a)` : `MLST t ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap
    - **select** the given reference from the heap
  - `let write (r:ref a) (v:a)` : `MLST unit ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap
    - **update** the heap with the given value at the given ref.
    - **put** the updated heap back

Adding untyped and monotonic references

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

    where $\mid \texttt{Used} : \mathrm{a}{:}\mathtt{Type}_0 \rightarrow \mathrm{v}{:}\mathrm{a} \rightarrow \mathrm{t}{:}\mathtt{tag} \rightarrow \mathtt{cell}$

    $\texttt{type tag} = \texttt{Typed} : \texttt{tag} \mid \texttt{Untyped} : \texttt{tag}$
  - actions corresponding to `uref`s have **weaker types** than for `ref`s

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

    where $\mid \texttt{Used} : \mathrm{a}{:}\mathtt{Type}_0 \rightarrow \mathrm{v}{:}\mathrm{a} \rightarrow \mathrm{t}{:}\mathtt{tag}\ \mathrm{a} \rightarrow \mathtt{cell}$

    $\texttt{type tag a} = \texttt{Typed} : \mathtt{rel}{:}\mathtt{preorder}\ \mathrm{a} \rightarrow \mathtt{tag}\ \mathrm{a} \mid \texttt{Untyped} : \mathtt{tag}\ \mathrm{a}$
  - `mref`s provide **more flexibility** with ref.-wise monotonicity

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

Adding untyped and monotonic references

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

    $$\mid\ \texttt{Used} : \mathsf{a{:}Type_0} \to \mathsf{v{:}a} \to \texttt{t:tag} \to \texttt{cell}$$

    where

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

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

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

    $$\mid\ \texttt{Used} : \mathsf{a{:}Type_0} \to \mathsf{v{:}a} \to \texttt{t:tag a} \to \texttt{cell}$$

    where

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

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

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

Adding untyped and monotonic references

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

    $$| \ \mathtt{Used} : \mathtt{a}{:}\mathtt{Type}_0 \to \mathtt{v}{:}\mathtt{a} \to \mathtt{t}{:}\mathtt{tag} \to \mathtt{cell}$$

    where

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

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

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

    $$| \ \mathtt{Used} : \mathtt{a}{:}\mathtt{Type}_0 \to \mathtt{v}{:}\mathtt{a} \to \mathtt{t}{:}\mathtt{tag}\ \mathtt{a} \to \mathtt{cell}$$

    where

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

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

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

Adding untyped and monotonic references

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

    $$\mid\ \mathsf{Used} : \mathtt{a{:}Type_0} \to \mathtt{v{:}a} \to \mathtt{t{:}tag} \to \mathtt{cell}$$

    where

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

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

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

    $$\mid\ \mathsf{Used} : \mathtt{a{:}Type_0} \to \mathtt{v{:}a} \to \mathtt{t{:}tag\ a} \to \mathtt{cell}$$

    where

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

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

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

Conclusion

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

- See our POPL 2018 paper for
  - further **examples** and **case studies**
  - **meta-theory** and **total correctness** for `MST`
    - based on an instrumented operational semantics

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

    - and cut elimination for the `witnessed`-logic
  - first steps towards **monadic reification** for `MST`
    - useful for extrinsic reasoning, e.g., for relational properties
    - but have to be careful when breaking abstraction

Conclusion

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

- See our POPL 2018 paper for
  - further **examples** and **case studies**
  - **meta-theory** and **total correctness** for `MST`
    - based on an instrumented operational semantics

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

    - and cut elimination for the `witnessed`-logic
  - first steps towards **monadic reification** for `MST`
    - useful for extrinsic reasoning, e.g., for relational properties
    - but have to be careful when breaking abstraction