Recalling a Witness

Foundations and Applications of Monotonic State

Danel Ahman @ INRIA Paris

Cătălin Hrițcu and Kenji Maillard @ INRIA Paris

Cédric Fournet, Aseem Rastogi, and Nikhil Swamy @ MSR

POPL 2018

January 12, 2018

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 the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the 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 the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

Monotonicity in program verification

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

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

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

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

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

- However, if c_p **does not remove**, then $\lambda\, \texttt{s} \,.\, \texttt{v} \in \texttt{s}$ is **stable**, and we would like the program logic to give us $\texttt{v} \in \texttt{get()}$ "**for free**"

Monotonicity in program verification

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

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

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

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

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

- However, if c_p **does not remove**, then $\lambda\, \mathtt{s}\,.\,\mathtt{v} \in \mathtt{s}$ is **stable**, and we would like the program logic to give us $\mathtt{v} \in \texttt{get}()$ "**for free**"

Monotonicity in program verification

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

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

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

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

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

- However, if c_p **does not remove**, then $\lambda\, \texttt{s} \,.\, \texttt{v} \in \texttt{s}$ is **stable**, and we would like the program logic to give us $\texttt{v} \in \texttt{get()}$ "**for free**"

Monotonicity in program verification

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

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

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

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

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

- However, if c_p **does not remove**, 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

- **Programs** also rely 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

- **Programs** also rely 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

- **Programs** also rely 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

- **Programs** also rely 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`)
- More in the paper
  - temporarily **violating monotonicity** via snapshots
  - two substantial case studies
    - 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`)
- More in the paper
  - temporarily **violating monotonicity** via snapshots
  - two substantial case studies
    - 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`)

- More in the paper
  - temporarily **violating monotonicity** via snapshots
  - two substantial case studies
    - 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 the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

Key ideas behind our general framework

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

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F*) with
  - a means for turning a p into a **state-independent proposition**
  - a means to **witness** the validity of p s in some state s
  - 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 focus on **monotonic programs** and **stable predicates**
    - per verification task, we **choose a preorder** `rel` on states
        - set inclusion, heap inclusion, increasing counter values, ...
    - a stateful program e is **monotonic** (wrt. `rel`) when

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

    - 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 for turning a p into a **state-independent proposition**
    - a means to **witness** the validity of p s in some state s
    - 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 focus on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** `rel` on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program e is **monotonic** (wrt. `rel`) when

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F*) with
  - a means for turning a p into a **state-independent proposition**
  - a means to **witness** the validity of p s in some state s
  - 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 focus on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** `rel` on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program e is **monotonic** (wrt. `rel`) when

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

  - 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 for turning a p into a **state-independent proposition**
  - a means to **witness** the validity of p s in some state s
  - 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 focus on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** `rel` on states
    - set inclusion, heap inclusion, increasing counter values, . . .
  - a stateful program e is **monotonic** (wrt. `rel`) when

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

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

Key ideas behind our general framework

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

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

  - 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 for turning a `p` into a **state-independent proposition**
  - a means to **witness** the validity of `p` s in some state s
  - 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 the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

Recap: Ordinary global state in F*

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

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

  where

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

- ST is an **abstract** pre-postcondition refinement of

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

- 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** will be defined in F* using monotonicity

Recap: Ordinary global state in F*

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

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

  where

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

- ST is an **abstract** pre-postcondition refinement of

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

- The global state **actions** have types

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

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

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

Recap: Ordinary global state in F*

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

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

  where

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

- ST is an **abstract** pre-postcondition refinement of

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

- The global state **actions** have types

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

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

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

Recap: Ordinary global state in F*

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

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

  where

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

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

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

- The global state **actions** have types

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

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

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

New: Monotonic global state in F*

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

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

  where pre and post are typed as in ST

- 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

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

New: Monotonic global state in F*

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

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

  where pre and post are typed as in ST

- 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

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

New: Monotonic global state in F*

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

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

  where pre and post are typed as in ST

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

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

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

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

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

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

New: Monotonic global state in F*

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

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

  where pre and post are typed as in $\mathtt{ST}$

- 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

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

New: Monotonic global state in F*

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

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

  where pre and post are typed as in ST

- 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

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

New: Recalling a Witness

- We introduce a **logical capability** (a **modality** in ongoing work)

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

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

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

- We add a **stateful introduction rule** for `witnessed`

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

- We add a **stateful elimination rule** for `witnessed`

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

New: Recalling a Witness

- We introduce a **logical capability** (a **modality** in ongoing work)

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

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

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

- We add a **stateful introduction rule** for witnessed

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

- We add a **stateful elimination rule** for witnessed

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

New: Recalling a Witness

- We introduce a **logical capability** (a **modality** in ongoing work)

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

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

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

- We add a **stateful introduction rule** for witnessed

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

- We add a **stateful elimination rule** for witnessed

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

New: Recalling a Witness

- We introduce a **logical capability** (a **modality** in ongoing work)

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

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

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

- We add a **stateful introduction rule** for witnessed

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

- We add a **stateful elimination rule** for witnessed

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

Outline

- Monotonic state by example
- Key ideas behind our general framework
- Accommodating monotonic state in F*
- Some examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Monadic reification and reflection (see the paper)
- Meta-theory and correctness results (see the paper)

The motivating example revisited

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

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

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

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

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

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

    around the program is handled **similarly easily**

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

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

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

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**

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

- **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**

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

- **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**

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

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

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

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

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

    around the program is handled **similarly easily**

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

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

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

The motivating example revisited

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

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

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

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

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

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

    around the program is handled **similarly easily**

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

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

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

ML-style typed references (local state)

- First, we define a type of **heaps**

$\texttt{type heap} =$

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

where

$\texttt{type cell} =$

$\quad | \; \texttt{Unused} : \texttt{cell}$

$\quad | \; \texttt{Used} : \texttt{a}{:}\texttt{Type} \to \texttt{v}{:}\texttt{a} \to \texttt{cell}$

- Next, we define the **heap inclusion** preorder

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

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

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

$\quad | \; \texttt{Unused} \,,\, \texttt{Unused} \; \to \; \top$

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

ML-style typed references (local state)

- First, we define a type of **heaps**

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

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

  where

  $\mathtt{type}\ \mathtt{cell} =$

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

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

- Next, we define the **heap inclusion** preorder

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

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

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

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

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

ML-style typed references (local state)

- First, we define a type of **heaps**

  $\texttt{type heap} =$

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

  where

  $\texttt{type cell} =$

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

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

- Next, we define the **heap inclusion** preorder

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

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

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

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

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

ML-style typed references (local state)

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

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

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

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

where

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

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

ML-style typed references (local state)

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

$$\mathtt{LST}\ \mathtt{t}\ \mathtt{pre}\ \mathtt{post} \stackrel{\mathsf{def}}{=} \mathtt{MST}_{\mathtt{heap},\mathtt{heap\_inclusion}}\ \mathtt{t}\ \mathtt{pre}\ \mathtt{post}$$

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

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

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**

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

- 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**

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

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

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

  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 `LST`'s **actions** using `MST`'s actions
  - `let alloc` (a:Type) (v:a) : `LST` (`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`) : `LST` `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) : `LST` `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 `LST`'s **actions** using `MST`'s actions
  - `let alloc (a:Type) (v:a) : LST (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) : LST 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) : LST unit ... = ...`
    - **recall** that the given ref. is in the heap
    - **get** the current heap
    - **update** the heap with the given value at the given ref.
    - **put** the updated heap back

Adding untyped and monotonic references

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

    where

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

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

  - `uref`s can be extended to also support **deallocation**

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

    where

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

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

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

Adding untyped and monotonic references

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

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

    where

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

  - `urefs` can be extended to also support **deallocation**

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

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

    where

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

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

Adding untyped and monotonic references

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

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

    where

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

  - `urefs` can be extended to also support **deallocation**

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

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

    where

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

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

Conclusion

- In conclusion
  - making use of monotonicity is very **useful** in verification
  - using monotonicity can be distilled into a **simple** interface
  - useful for **programming** (refs.) and **verification** (Prj. Everest)

- See the paper for
  - further **examples** and **case studies**
  - **meta-theory** and **correctness results** for `MST`
    - based on an instrumented operational semantics

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

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

Conclusion

- In conclusion
  - making use of monotonicity is very **useful** in verification
  - using monotonicity can be distilled into a **simple** interface
  - useful for **programming** (refs.) and **verification** (Prj. Everest)

- See the paper for
  - further **examples** and **case studies**
  - **meta-theory** and **correctness results** for `MST`
    - based on an instrumented operational semantics

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

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

Thank you!

Questions?