

Recalling a Witness

Foundations and Applications of Monotonic State

Danel Ahman

Prosecco Team, 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 approach
- 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 approach
- 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)

Monotonic state and program 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**

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

- 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` **does not remove**, then $\lambda s.v \in s$ is **stable**, and we would like the program logic to give us $v \in \text{get()}$ “for free”

Monotonic state and program verification

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

insert v; complex_procedure(); **assert** ($v \in \text{get}()$)

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

Monotonic state and program verification

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

insert v; complex_procedure(); **assert** ($v \in \text{get}()$)

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

Monotonic state and program verification

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

insert v; complex_procedure(); **assert** ($v \in \text{get}()$)

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

Monotonicity is really useful!

- In this talk
 - **motivating example** and **monotonic counters**
 - both **typed** (`ref t`) and **untyped references** (`uref`)
 - more flexibility with **monotonic references** (`mref t rel`)
- More in the paper
 - temporarily performing **non-monotonic updates** 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
 - **motivating example** and **monotonic counters**
 - both **typed** (`ref t`) and **untyped references** (`uref`)
 - more flexibility with **monotonic references** (`mref t rel`)
- More in the paper
 - temporarily performing **non-monotonic updates** 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
 - **motivating example** and **monotonic counters**
 - both **typed** (`ref t`) and **untyped references** (`uref`)
 - more flexibility with **monotonic references** (`mref t rel`)
- More in the paper
 - temporarily performing **non-monotonic updates** 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 approach
- 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 approach

- 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

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - α means for turning a p into a **state-independent proposition**
 - α means to **witness** the validity of $p \ s$ in some state s
 - α 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 approach

- 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

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with

- α means for turning a p into a **state-independent proposition**
- α means to **witness** the validity of $p \ s$ in some state s
- α 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 approach

- 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

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - α means for turning a p into a **state-independent proposition**
 - α means to **witness** the validity of $p \ s$ in some state s
 - α 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 approach

- 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

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - α means for turning a p into a **state-independent proposition**
 - α means to **witness** the validity of $p s$ in some state s
 - α 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 approach

- 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

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - α means for turning a p into a **state-independent proposition**
 - α means to **witness** the validity of $p \ s$ in some state s
 - α 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 approach

- 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

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - \mathbf{a} means for turning a p into a **state-independent proposition**
 - \mathbf{a} means to **witness** the validity of $p \ s$ in some state s
 - \mathbf{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 approach

- 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

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

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

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

- **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^*

Outline

- Monotonic state by example
- Key ideas behind our approach
- 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**

$$ST_{\text{state}}\ t\ (\text{requires}\ \text{pre})\ (\text{ensures}\ \text{post})$$

where

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

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

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

- The global state **actions** have types

$$\text{get} : \text{unit} \rightarrow ST\ \text{state}\ (\text{requires}\ (\lambda _.\top))\ (\text{ensures}\ (\lambda s_0\ s\ s_1.\ s_0 = s = s_1))$$
$$\text{put} : s:\text{state} \rightarrow ST\ \text{unit}\ (\text{requires}\ (\lambda _.\top))\ (\text{ensures}\ (\lambda _ \ s_1.\ s_1 = s))$$

- **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**

$$ST_{\text{state}} \, t \, (\text{requires } \text{pre}) \, (\text{ensures } \text{post})$$

where

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

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

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

- The global state **actions** have types

$$\text{get} : \text{unit} \rightarrow ST \, \text{state} \, (\text{requires } (\lambda _ . T)) \, (\text{ensures } (\lambda s_0 \, s \, s_1 . s_0 = s = s_1))$$

$$\text{put} : s : \text{state} \rightarrow ST \, \text{unit} \, (\text{requires } (\lambda _ . T)) \, (\text{ensures } (\lambda _ \, s_1 . s_1 = s))$$

- 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**

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

where

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

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

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

- The global state **actions** have types

$$\text{get} : \text{unit} \rightarrow \text{ST } \text{state} \, (\text{requires } (\lambda _ . \top)) \, (\text{ensures } (\lambda s_0 \, s \, s_1 . s_0 = s = s_1))$$

$$\text{put} : s : \text{state} \rightarrow \text{ST } \text{unit} \, (\text{requires } (\lambda _ . \top)) \, (\text{ensures } (\lambda _ _ s_1 . s_1 = s))$$

- 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**

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

where

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

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

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

- The global state **actions** have types

$$\text{get} : \text{unit} \rightarrow \text{ST } \text{state} \, (\text{requires } (\lambda _ . \top)) \, (\text{ensures } (\lambda s_0 \, s \, s_1 . s_0 = s = s_1))$$

$$\text{put} : s : \text{state} \rightarrow \text{ST } \text{unit} \, (\text{requires } (\lambda _ . \top)) \, (\text{ensures } (\lambda _ _ s_1 . s_1 = s))$$

- **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**

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

where pre and post are typed as in ST

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

$\text{get} : \text{unit} \rightarrow \text{MST}\ \text{state}\ (\text{requires}\ (\lambda\ _.\text{T}))$
 $\quad\quad\quad (\text{ensures}\ (\lambda\ s_0\ s\ s_1.\ s_0 = s = s_1))$

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

$\text{put} : \text{s}:\text{state} \rightarrow \text{MST}\ \text{unit}\ (\text{requires}\ (\lambda\ s_0.\ \text{rel}\ s_0\ s))$
 $\quad\quad\quad (\text{ensures}\ (\lambda\ _.\ s_1.\ s_1 = s))$

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

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

New: Monotonic global state in F^*

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

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

where pre and post are typed as in ST

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

$\text{get} : \text{unit} \rightarrow \text{MST state} \ (\text{requires} \ (\lambda _ . \top))$
 $(\text{ensures} \ (\lambda s_0 \ s \ s_1 . s_0 = s = s_1))$

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

$\text{put} : \text{s}:\text{state} \rightarrow \text{MST unit} \ (\text{requires} \ (\lambda s_0 . \text{rel} \ s_0 \ s))$
 $(\text{ensures} \ (\lambda _ \ s_1 . s_1 = s))$

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

$\text{mst} \ t \stackrel{\text{def}}{=} \text{s}_0:\text{state} \rightarrow t * \text{s}_1:\text{state} \{ \text{rel} \ s_0 \ s_1 \}$

New: Monotonic global state in F*

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

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

where pre and post are typed as in ST

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

$$\begin{aligned} \text{get} : \text{unit} \rightarrow \text{MST} \ \text{state} \ (\text{requires} \ (\lambda _ . \top)) \\ (\text{ensures} \ (\lambda \ s_0 \ s \ s_1 . s_0 = s = s_1)) \end{aligned}$$

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

$$\begin{aligned} \text{put} : \text{s} : \text{state} \rightarrow \text{MST} \ \text{unit} \ (\text{requires} \ (\lambda \ s_0 . \text{rel} \ s_0 \ s)) \\ (\text{ensures} \ (\lambda \ _ \ s_1 . s_1 = s)) \end{aligned}$$

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

$$\text{mst} \ t \stackrel{\text{def}}{=} \text{s}_0 : \text{state} \rightarrow t * \text{s}_1 : \text{state} \{ \text{rel} \ s_0 \ s_1 \}$$

New: Monotonic global state in F*

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

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

where pre and post are typed as in ST

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

$$\begin{aligned} \text{get} : \text{unit} \rightarrow \text{MST} \, \text{state} \, (\text{requires} \, (\lambda _ . \top)) \\ (\text{ensures} \, (\lambda s_0 \, s \, s_1 . s_0 = s = s_1)) \end{aligned}$$

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

$$\begin{aligned} \text{put} : s : \text{state} \rightarrow \text{MST} \, \text{unit} \, (\text{requires} \, (\lambda s_0 . \text{rel} \, s_0 \, s)) \\ (\text{ensures} \, (\lambda _ _ s_1 . s_1 = s)) \end{aligned}$$

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

$$\text{mst} \, t \stackrel{\text{def}}{=} s_0 : \text{state} \rightarrow t * s_1 : \text{state} \{ \text{rel} \, s_0 \, s_1 \}$$

New: Monotonic global state in F^*

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

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

where pre and post are typed as in ST

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

$$\begin{aligned} \text{get} : \text{unit} \rightarrow \text{MST} \ \text{state} \ (\text{requires} \ (\lambda _ . \top)) \\ (\text{ensures} \ (\lambda s_0 \ s \ s_1 . s_0 = s = s_1)) \end{aligned}$$

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

$$\begin{aligned} \text{put} : s : \text{state} \rightarrow \text{MST} \ \text{unit} \ (\text{requires} \ (\lambda s_0 . \text{rel} \ s_0 \ s)) \\ (\text{ensures} \ (\lambda _ \ s_1 . s_1 = s)) \end{aligned}$$

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

$$\text{mst} \ t \stackrel{\text{def}}{=} s_0 : \text{state} \rightarrow t * s_1 : \text{state} \{ \text{rel} \ s_0 \ s_1 \}$$

New: Recalling a Witness

- We introduce a **logical capability (modality)**

$\text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type}$

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

$\text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma} \left(\text{requires } (\forall s. p \ s \implies q \ s) \right)$
 $\left(\text{ensures } (\text{witnessed } p \implies \text{witnessed } q) \right)$

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

$\text{witness} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left(\text{requires } (\lambda s_0. p \ s_0 \wedge \text{stable } p) \right)$
 $\left(\text{ensures } (\lambda s_0 _ s_1. s_0 = s_1 \wedge \text{witnessed } p) \right)$

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

$\text{recall} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left(\text{requires } (\lambda _. \text{witnessed } p) \right)$
 $\left(\text{ensures } (\lambda s_0 _ s_1. s_0 = s_1 \wedge p \ s_1) \right)$

New: Recalling a Witness

- We introduce a **logical capability (modality)**

$\text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type}$

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

$\text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma} \left(\text{requires} \left(\forall s. p \ s \implies q \ s \right) \right.$
 $\left. \left(\text{ensures} \left(\text{witnessed } p \implies \text{witnessed } q \right) \right) \right)$

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

$\text{witness} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left(\text{requires} \left(\lambda s_0. p \ s_0 \wedge \text{stable } p \right) \right.$
 $\left. \left(\text{ensures} \left(\lambda s_0 _ s_1. s_0 = s_1 \wedge \text{witnessed } p \right) \right) \right)$

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

$\text{recall} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left(\text{requires} \left(\lambda _. \text{witnessed } p \right) \right.$
 $\left. \left(\text{ensures} \left(\lambda s_0 _ s_1. s_0 = s_1 \wedge p \ s_1 \right) \right) \right)$

New: Recalling a Witness

- We introduce a **logical capability (modality)**

$\text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type}$

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

$\text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma} \left(\text{requires} \left(\forall s. p\ s \implies q\ s \right) \right.$
 $\left. \left(\text{ensures} \left(\text{witnessed}\ p \implies \text{witnessed}\ q \right) \right) \right)$

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

$\text{witness} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left(\text{requires} \left(\lambda s_0. p\ s_0 \wedge \text{stable}\ p \right) \right.$
 $\left. \left(\text{ensures} \left(\lambda s_0 - s_1. s_0 = s_1 \wedge \text{witnessed}\ p \right) \right) \right)$

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

$\text{recall} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left(\text{requires} \left(\lambda _. \text{witnessed}\ p \right) \right.$
 $\left. \left(\text{ensures} \left(\lambda s_0 - s_1. s_0 = s_1 \wedge p\ s_1 \right) \right) \right)$

New: Recalling a Witness

- We introduce a **logical capability (modality)**

$\text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type}$

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

$\text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma} \left(\text{requires} \left(\forall s. p \ s \implies q \ s \right) \right.$
 $\left. \left(\text{ensures} \left(\text{witnessed } p \implies \text{witnessed } q \right) \right) \right)$

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

$\text{witness} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left(\text{requires} \left(\lambda s_0. p \ s_0 \wedge \text{stable } p \right) \right.$
 $\left. \left(\text{ensures} \left(\lambda s_0 - s_1. s_0 = s_1 \wedge \text{witnessed } p \right) \right) \right)$

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

$\text{recall} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left(\text{requires} \left(\lambda _. \text{witnessed } p \right) \right.$
 $\left. \left(\text{ensures} \left(\lambda s_0 - s_1. s_0 = s_1 \wedge p \ s_1 \right) \right) \right)$

Outline

- Monotonic state by example
- Key ideas behind our approach
- 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**

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

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

```
insert v; witness ( $\lambda s. v \in s$ ); c_p(); recall ( $\lambda s. v \in s$ ); assert (v ∈ get())
```

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

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

around the program is handled similarly easily

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

```
create 0; incr(); witness ( $\lambda c. c > 0$ ); c_p(); recall ( $\lambda c. c > 0$ )
```

The motivating example revisited

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

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

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

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

- For **any other w**, wrapping

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

around the program is handled similarly easily

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

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

The motivating example revisited

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

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

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

```
insert v; witness ( $\lambda s. v \in s$ ); c_p(); recall ( $\lambda s. v \in s$ ); assert (v ∈ get())
```

- For any other w , wrapping

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

around the program is handled similarly easily

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

```
create 0; incr(); witness ( $\lambda c. c > 0$ ); c_p(); recall ( $\lambda c. c > 0$ )
```

The motivating example revisited

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

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

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

```
insert v; witness ( $\lambda s. v \in s$ ); c_p(); recall ( $\lambda s. v \in s$ ); assert (v ∈ get())
```

- For **any other** w, wrapping

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

around the program is handled similarly easily

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

```
create 0; incr(); witness ( $\lambda c. c > 0$ ); c_p(); recall ( $\lambda c. c > 0$ )
```

The motivating example revisited

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

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

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

```
insert v; witness ( $\lambda s. v \in s$ ); c_p(); recall ( $\lambda s. v \in s$ ); assert (v ∈ get())
```

- For **any other** w, wrapping

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

around the program is handled similarly easily

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

```
create 0; incr(); witness ( $\lambda c. c > 0$ ); c_p(); recall ( $\lambda c. c > 0$ )
```

ML-style typed references (local state)

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

```
type heap =
```

```
| H : h : (N → cell) → ctr : N { ∀ n. ctr ≤ n ⇒ h n = Unused } → heap
```

where

```
type cell =
```

```
| Unused : cell
```

```
| Used : a : Type → v : a → cell
```

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

```
let heap_inclusion (H h0 _) (H h1 _) = ∀ id. match h0 id, h1 id with
```

```
| Used a _, Used b _ → a = b
```

```
| Unused, Used _ _ → ⊥
```

```
| Unused, Unused → ⊤
```

```
| Used _ _, Unused → ⊥
```

ML-style typed references (local state)

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

```
type heap =
```

```
| H : h:( $\mathbb{N} \rightarrow \text{cell}$ )  $\rightarrow$  ctr: $\mathbb{N}\{\forall n. \text{ctr} \leq n \implies h\ n = \text{Unused}\}$   $\rightarrow$  heap
```

where

```
type cell =
```

```
| Unused : cell
```

```
| Used : a:Type  $\rightarrow$  v:a  $\rightarrow$  cell
```

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

```
let heap_inclusion (H h0 _) (H h1 _) =  $\forall \text{id}. \text{match } h_0\ \text{id}, h_1\ \text{id} \text{ with}$ 
```

```
| Used a _, Used b _  $\rightarrow$  a = b
```

```
| Unused, Used _ _  $\rightarrow$   $\top$ 
```

```
| Unused, Unused  $\rightarrow$   $\top$ 
```

```
| Used _ _, Unused  $\rightarrow$   $\perp$ 
```


ML-style typed references (local state)

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

`type heap =`

`| H : h : (ℕ → cell) → ctr : ℕ {∀ n. ctr ≤ n ⇒ h n = Unused} → heap`

where

`type cell =`

`| Unused : cell`

`| Used : a : Type → v : a → cell`

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

`let heap_inclusion (H h0 _) (H h1 _) = ∀ id. match h0 id, h1 id with`

`| Used a _, Used b _ → a = b`

`| Unused, Used _ _ → ⊥`

`| Unused, Unused → ⊤`

`| Used _ _, Unused → ⊥`

ML-style typed references (local state)

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

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

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

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

where

```
let contains (H h _) id a =  
  match h id with  
  | Used b _  $\rightarrow$  a = b  
  | Unused  $\rightarrow \perp$ 
```

- Observe that contains is **stable** wrt. heap_inclusion

ML-style typed references (local state)

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

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

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

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

where

```
let contains (H h _) id a =  
  match h id with  
  | Used b _  $\rightarrow$  a = b  
  | Unused  $\rightarrow \perp$ 
```

- Observe that contains is **stable** wrt. heap_inclusion

ML-style typed references (local state)

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

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

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

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

where

```
let contains (H h _) id a =  
  match h id with  
  | Used b _  $\rightarrow$  a = b  
  | Unused  $\rightarrow \perp$ 
```

- Observe that contains is **stable** wrt. heap_inclusion

ML-style typed references (local state)

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

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

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

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

where

```
let contains (H h _) id a =  
  match h id with  
  | Used b _  $\rightarrow$  a = b  
  | Unused  $\rightarrow \perp$ 
```

- Observe that 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** $(a:\text{Type}) (v:a) : \text{LST } t \dots = \dots$
 - **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** $(r:\text{ref } a) : \text{LST } t \dots = \dots$
 - **recall** that the given ref. is in the heap
 - **get** the current heap
 - **select** the given reference from the heap
 - **let write** $(r:\text{ref } a) (v:a) : \text{LST } \text{unit } \dots = \dots$
 - **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`)

- Used heap cells are extended with **tags**

`| Used : a:Type → v:a → t:tag → cell`

where

`type tag = Typed : tag | Untyped : tag`

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

- Monotonic references (`mref a rel`)

- Used heap cells are extended with **typed tags**

`| Used : a:Type → v:a → t:tag a → cell`

where

`type tag a = Typed : rel:preorder a → tag a | Untyped : tag a`

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

Adding untyped and monotonic references

- **Untyped references** (`uref`)

- Used heap cells are extended with **tags**

| `Used : a:Type → v:a → t:tag → cell`

where

`type tag = Typed : tag | Untyped : tag`

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

- **Monotonic references** (`mref a rel`)

- Used heap cells are extended with **typed tags**

| `Used : a:Type → v:a → t:tag a → cell`

where

`type tag a = Typed : rel:preorder a → tag a | Untyped : tag a`

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

Adding untyped and monotonic references

- **Untyped references** (`uref`)

- Used heap cells are extended with **tags**

| `Used : a:Type → v:a → t:tag → cell`

where

`type tag = Typed : tag | Untyped : tag`

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

- **Monotonic references** (`mref a rel`)

- Used heap cells are extended with **typed tags**

| `Used : a:Type → v:a → t:tag a → cell`

where

`type tag a = Typed : rel:preorder a → tag a | Untyped : 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
$$(\text{witness } x.\varphi, s, W) \rightsquigarrow (\text{return } (), s, W \cup \{x.\varphi\})$$
 - and cut elimination for the witnessed-logic
 - revealing the representation of MST via **monadic reification**
 - useful for extrinsic reasoning, e.g., for relational properties

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
$$(\text{witness } x.\varphi, s, W) \rightsquigarrow (\text{return } (), s, W \cup \{x.\varphi\})$$
 - and cut elimination for the witnessed-logic
 - revealing the representation of **MST** via **monadic reification**
 - useful for extrinsic reasoning, e.g., for relational properties

Thank you!

Questions?