Recalling a Witness
Foundations and Applications of Monotonic State

Danel Ahman @ INRIA Paris

Cătălin Hriţcu and Kenji Maillard @ INRIA Paris
Cédric Fournet, Aseem Rastogi, and Nikhil Swamy @ MSR

POPL 2018
January 12, 2018
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 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**

  \[
  \text{insert } v; \text{ complex\_procedure}(); \text{ assert } (v \in \text{get}())
  \]

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

  \[
  \{\lambda s. v \in s\} \text{ complex\_procedure}() \{\lambda s. v \in s\}
  \]

- 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 \(c_p\) **never removes**, 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 in program verification

• Consider a program operating on set-valued state

\[
\text{insert } v; \text{ complex\_procedure}(); \text{ assert } (v \in \text{get}())
\]

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

\[
\{\lambda s. v \in s\} \text{ complex\_procedure}() \{\lambda s. v \in s\}
\]

• 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 \(c_p\) never removes, 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 in program verification

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

  \[
  \text{insert } v; \text{ complex\_procedure}(); \text{ assert } (v \in \text{get}())
  \]

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

  \[
  \{\lambda s. v \in s\} \text{ complex\_procedure}() \{\lambda s. v \in s\}
  \]

- 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 \(c_p\) never removes, 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 in program verification

- Consider a program operating on \textbf{set-valued state}

  \[ \text{insert } v; \text{ complex\_procedure()}; \text{ assert } (v \in \text{get()}) \]

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

  \[ \{ \lambda s. v \in s \} \text{ complex\_procedure()} \{ \lambda s. v \in s \} \]

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

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

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

- Consider ML-style typed references $r: \text{ref } a$
  - $r$ 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 \( r : \text{ref } a \)
  - \( r \) 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 (\texttt{ref t}) and untyped references (\texttt{uref})
  - more flexibility with monotonic references (\texttt{mref t rel})

- More in the paper
  - temporarily violating monotonicity via snapshots
  - two substantial case studies in \texttt{F*}
    - a secure file-transfer application
    - Ariadne state continuity protocol \cite{strackx2016}
  - pointers to other works in \texttt{F*} relying on monotonicity for
    - sophisticated region-based memory models \cite{fstar-lang.org}
    - crypto and TLS verification \cite{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 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 \((\text{ref} \ t)\) and untyped references \((\text{uref})\)
  • more flexibility with monotonic references \((\text{mref} \ t \ \text{rel})\)

• More in the paper
  • 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 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 $\text{rel}$ on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program $e$ is monotonic (wrt. $\text{rel}$) when
    $$\forall s e' s'. (e, s) \leadsto^* (e', s') \implies \text{rel} \; s \; s'$$
  - a stateful predicate $p$ is stable (wrt. $\text{rel}$) when
    $$\forall s s'. p \; s \land \text{rel} \; s \; s' \implies p \; s'$$

- 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 focus on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, …
  - a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    \[
    \forall s \; e' \; s'. \; (e, s) \leadsto^* (e', s') \implies \text{rel} \; s \; s'
    \]
  - a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
    \[
    \forall s \; s'. \; p \; s \; \land \; \text{rel} \; s \; s' \implies p \; s'
    \]
- Our solution: extend Hoare-style program logics (e.g., \( \text{F*} \)) 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 \textit{ad hoc} uses in \( \text{F*} \)
Key ideas behind our general framework

- We focus on monotonic programs and stable predicates
  - per verification task, we choose a preorder $\text{rel}$ on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program $e$ is monotonic (wrt. $\text{rel}$) when
    \[
    \forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s s'
    \]
  - a stateful predicate $p$ is stable (wrt. $\text{rel}$) when
    \[
    \forall s s'. p s \land \text{rel } s s' \implies p s'
    \]
- 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 focus on **monotonic programs** and **stable predicates**
  - per verification task, we choose a preorder \( \text{rel} \) on states
    * set inclusion, heap inclusion, increasing counter values, . . .
  - a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    \[
    \forall s e' s'. (e, s) \rightsquigarrow^* (e', s') \implies \text{rel } s s'
    \]

- a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
  \[
  \forall s s'. p s \land \text{rel } s s' \implies p s'
  \]

- 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 focus on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** $\text{rel}$ on states
    - set inclusion, heap inclusion, increasing counter values, …
  - a stateful program $e$ is **monotonic** (wrt. $\text{rel}$) when
    \[
    \forall s e' s'. (e, s) \xrightarrow{\ast} (e', s') \implies \text{rel } s s'
    \]
  - a stateful predicate $p$ is **stable** (wrt. $\text{rel}$) when
    \[
    \forall s s'. p s \land \text{rel } s s' \implies p s'
    \]

- 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 focus on **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, …
  - a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    \[
    \forall s e' s'. (e, s) \sim^* (e', s') \implies \text{rel} s s'
    \]
  - a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
    \[
    \forall s s'. p s \land \text{rel} s s' \implies p s'
    \]

- **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 focus on **monotonic programs** and stable predicates
  - per verification task, we choose a preorder \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, ... 
  - a stateful program \( e \) is **monotonic** (wrt. \( \text{rel} \)) when
    \[
    \forall s e' s'. (e, s) \sim^* (e', s') \implies \text{rel} s s'
    \]
  - a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
    \[
    \forall s s'. p s \land \text{rel} s s' \implies p s'
    \]

- **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*
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`:

  \[
  \text{ST}_{\text{state}} \ t \ (\text{requires } \text{pre}) \ (\text{ensures } \text{post})
  \]

  where

  \[
  \text{pre} : \text{state} \rightarrow \text{Type} \quad \text{post} : \text{state} \rightarrow t \rightarrow \text{state} \rightarrow \text{Type}
  \]

- ST is an abstract pre-postcondition refinement of

  \[
  \text{st} \ t \ \overset{\text{def}}{=} \ \text{state} \rightarrow t \ast \text{state}
  \]

- The global state actions have types

  \[
  \begin{align*}
  \text{get} : \text{unit} \rightarrow \text{ST state} & \ (\text{requires } (\lambda_.\top)) \ (\text{ensures } (\lambda s_0 s s_1.s_0 = s = s_1)) \\
  \text{put} : s: \text{state} \rightarrow \text{ST unit} & \ (\text{requires } (\lambda_.\top)) \ (\text{ensures } (\lambda_\_ s_1.s_1 = s))
  \end{align*}
  \]

- 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

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

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

$$\text{st} \ t \ \overset{\text{def}}{=} \text{state} \rightarrow t \ast \text{state}$$

- The global state **actions** have types

  - `get : unit \rightarrow \text{ST state} \ (\text{requires } (\lambda \_ . \top)) \ (\text{ensures } (\lambda s_0 s s_1 . s_0 = s = s_1))`
  - `put : s : \text{state} \rightarrow \text{ST 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
  \[
  \text{pre} : \text{state} \rightarrow \text{Type} \quad \text{post} : \text{state} \rightarrow \ t \rightarrow \text{state} \rightarrow \text{Type}
  \]

- **ST** is an abstract pre-postcondition refinement of
  \[
  \text{st} \ t \ \overset{\text{def}}{=} \text{state} \rightarrow \ t \ast \text{state}
  \]

- The global state **actions** have types
  \[
  \text{get} : \text{unit} \rightarrow \text{ST} \ \text{state} \ (\text{requires } (\lambda \_ . \top)) \ (\text{ensures } (\lambda s_0 \ s s_1 . s_0 = s = s_1))
  \]
  \[
  \text{put} : \text{s:state} \rightarrow \text{ST} \ \text{unit} \ (\text{requires } (\lambda \_ . \top)) \ (\text{ensures } (\lambda \_ \_ s_1 . s_1 = s))
  \]

- 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**

  \[
  \text{ST}_{\text{state}} \ t \ (\text{requires pre}) \ (\text{ensures post})
  \]

  where

  \[
  \begin{align*}
  \text{pre} : & \ \text{state} \to \text{Type} \\
  \text{post} : & \ \text{state} \to t \to \text{state} \to \text{Type}
  \end{align*}
  \]

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

  \[
  \text{st t} \ \overset{\text{def}}{=} \ \text{state} \to t \ast \text{state}
  \]

- The global state **actions** have types

  \[
  \begin{align*}
  \text{get} : & \ \text{unit} \to \text{ST} \ \text{state} \ (\text{requires } (\lambda _. \top)) \ (\text{ensures } (\lambda s_0 s s_1. s_0 = s = s_1)) \\
  \text{put} : & \ \text{s:state} \to \text{ST} \ \text{unit} \ (\text{requires } (\lambda _. \top)) \ (\text{ensures } (\lambda _- s_1. s_1 = s))
  \end{align*}
  \]

- **Refs.** and **local state** will be defined in F* using **monotonicity**
New: Monotonic global state in F*

- We capture monotonic state with a new computational type

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

where pre and post are typed as in ST

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

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

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

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

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

\[ \text{mst} \ t \ \overset{\text{def}}{=} s_0 : \text{state} \rightarrow t * s_1 : \text{state} \{\text{rel} \ s_0 \ s_1\} \]
New: Monotonic global state in F*

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

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

  where \( \text{pre} \) and \( \text{post} \) are typed as in \( \text{ST} \)

- The `get` action is typed as in \( \text{ST} \)

  \[
  \text{get} : \text{unit} \rightarrow \text{MST \ state} \ (\text{requires \ } (\lambda \_ . \top)) \\
  \hspace{1cm} (\text{ensures \ } (\lambda s_0 s_1 . s_0 = s = s_1))
  \]

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

  \[
  \text{put} : s : \text{state} \rightarrow \text{MST \ unit} \ (\text{requires \ } (\lambda s_0 . \text{rel} \ s_0 \ s)) \\
  \hspace{1cm} (\text{ensures \ } (\lambda _-_ s_1 . s_1 = s))
  \]

- So intuitively, \( \text{MST} \) is an **abstract** pre-postcondition refinement of

  \[
  \text{mst} t \overset{\text{def}}{=} s_0 : \text{state} \rightarrow t * s_1 : \text{state} \{\text{rel} \ s_0 \ s_1\}
  \]
New: Monotonic global state in F*

- We capture monotonic state with a new *computational type*
  \[ \text{MST}_{\text{state, rel}} \ t \ (\text{requires pre}) \ (\text{ensures post}) \]
  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
  \[ \text{put} : s : \text{state} \rightarrow \text{MST unit} \ (\text{requires } (\lambda s_0 . \text{rel } s_0 s)) \]
  \[ (\text{ensures } (\lambda _\_ s_1 . s_1 = s)) \]

- So intuitively, MST is an **abstract** pre-postcondition refinement of
  \[ \text{mst} \ t \overset{\text{def}}{=} s_0 : \text{state} \rightarrow t * s_1 : \text{state}\{\text{rel } s_0 s_1}\]
New: Monotonic global state in F*

- We capture monotonic state with a new **computational 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

\[
\text{get} : \text{unit} \rightarrow \text{MST} \ \text{state} \ (\text{requires} \ (\lambda \_. \top))
\]

\[
(\text{ensures} \ (\lambda s_0 s s_1 . s_0 = s = s_1))
\]

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

\[
\text{put} : s:\text{state} \rightarrow \text{MST} \ \text{unit} \ (\text{requires} \ (\lambda s_0 . \text{rel} s_0 s))
\]

\[
(\text{ensures} \ (\lambda \_. s_1 . s_1 = s))
\]

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

\[
\text{mst} \ t \ \overset{\text{def}}{=} s_0:\text{state} \rightarrow t * s_1:\text{state}\{\text{rel} s_0 s_1}\]

New: Monotonic global state in F*

- We capture monotonic state with a new computational 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

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

- To ensure monotonicity, the put action gets a precondition

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

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

\[ \text{mst} \ t \triangleq s_0 : \text{state} \rightarrow t * s_1 : \text{state}\{\text{rel } s_0 s_1\} \]
New: Recalling a Witness

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

  \[ \text{witnessed} : (\text{state} \to \text{Type}) \to \text{Type} \]

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

  \[ \text{wk} : p, q : (\text{state} \to \text{Type}) \to \text{Lemma} \]

  \[ (\text{requires} (\forall s. p s \implies q s)) \]

  \[ (\text{ensures} (\text{witnessed} p \implies \text{witnessed} q)) \]

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

  \[ \text{witness} : p : (\text{state} \to \text{Type}) \to \text{MST unit} \]

  \[ (\text{requires} (\lambda s_0. p s_0 \land \text{stable} p)) \]

  \[ (\text{ensures} (\lambda s_0 s_1. s_0 = s_1 \land \text{witnessed} p)) \]

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

  \[ \text{recall} : p : (\text{state} \to \text{Type}) \to \text{MST unit} \]

  \[ (\text{requires} (\lambda _. \text{witnessed} p)) \]

  \[ (\text{ensures} (\lambda s_0 s_1. s_0 = s_1 \land p s_1)) \]
New: Recalling a Witness

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

  \[ \text{witnessed} : \text{(state} \rightarrow \text{Type)} \rightarrow \text{Type} \]

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

  \[ \text{wk} : p, q : \text{(state} \rightarrow \text{Type)} \rightarrow \text{Lemma} \text{ (requires (} \forall s. p s \implies q s) } \]

  \[ \text{(ensures (} \text{witnessed } p \implies \text{witnessed } q) } \]

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

  \[ \text{witness} : p : \text{(state} \rightarrow \text{Type)} \rightarrow \text{MST unit (requires (} \lambda s_0. p s_0 \land \text{stable } p) } \]

  \[ \text{(ensures (} \lambda s_0 s_1. s_0 = s_1 \land \text{witnessed } p) } \]

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

  \[ \text{recall} : p : \text{(state} \rightarrow \text{Type)} \rightarrow \text{MST unit (requires (} \lambda _. \text{witnessed } p) } \]

  \[ \text{(ensures (} \lambda s_0 s_1. s_0 = s_1 \land p s_1) } \]
New: Recalling a Witness

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

\[ \text{witnessed} : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Type} \]

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

\[ \text{wk} : p, q : (\text{state} \rightarrow \text{Type}) \rightarrow \text{Lemma} \left( \begin{array}{l}
\text{(requires) } \forall s. p s \implies q s \\
\text{(ensures) } \text{witnessed} p \implies \text{witnessed} q
\end{array} \right) \]

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

\[ \text{witness} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left( \begin{array}{l}
\text{(requires) } \lambda s_0. p s_0 \land \text{stable} p \\
\text{(ensures) } \lambda s_0 s_1. s_0 = s_1 \land \text{witnessed} p
\end{array} \right) \]

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

\[ \text{recall} : p : (\text{state} \rightarrow \text{Type}) \rightarrow \text{MST unit} \left( \begin{array}{l}
\text{(requires) } \lambda_. \text{witnessed} p \\
\text{(ensures) } \lambda s_0 s_1. s_0 = s_1 \land p s_1
\end{array} \right) \]
New: Recalling a Witness

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

  \[
  \text{witnessed} : (\text{state} \to \text{Type}) \to \text{Type}
  \]

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

  \[
  \text{wk} : p, q : (\text{state} \to \text{Type}) \to \text{Lemma} \quad (\text{requires} \ (\forall s. p \ s \implies q \ s))
  \]

  \[
  (\text{ensures} \ (\text{witnessed} \ p \implies \text{witnessed} \ q))
  \]

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

  \[
  \text{witness} : p : (\text{state} \to \text{Type}) \to \text{MST unit} \quad (\text{requires} \ (\lambda s_0. p \ s_0 \land \text{stable} \ p))
  \]

  \[
  (\text{ensures} \ (\lambda s_0 s_1. s_0 = s_1 \land \text{witnessed} \ p))
  \]

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

  \[
  \text{recall} : p : (\text{state} \to \text{Type}) \to \text{MST unit} \quad (\text{requires} \ (\lambda . \text{witnessed} \ p))
  \]

  \[
  (\text{ensures} \ (\lambda s_0 s_1. s_0 = s_1 \land p \ s_1))
  \]
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**

  \[
  \text{insert } v; \ \text{complex\_procedure}(); \ \text{assert} \ (v \in \text{get}())
  \]

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

    \[
    \text{insert } v; \ \text{witness} \ (\lambda s. v \in s); \ \text{c\_p}(); \ \text{recall} \ (\lambda s. v \in s); \ \text{assert} \ (v \in \text{get}())
    \]

  - For any other $w$, wrapping

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

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

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

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

  \[
  \text{create } 0; \ \text{incr}(); \ \text{witness} \ (\lambda c. c > 0); \ \text{c\_p}(); \ \text{recall} \ (\lambda c. c > 0)
  \]
The motivating example revisited

- Recall the program operating on the \textbf{set-valued state}:
  \begin{verbatim}
  insert v; complex_procedure(); \textbf{assert} (v \in \text{get()})
  \end{verbatim}

- We pick \textbf{set inclusion} $\subseteq$ as our preorder \texttt{rel} on states.

- We prove the assertion by inserting a witness and recall:
  \begin{verbatim}
  insert v; \textbf{witness} ($\lambda s. v \in s$); cp(); \textbf{recall} ($\lambda s. v \in s$); \textbf{assert} (v \in \text{get()})
  \end{verbatim}

- For any other $w$, wrapping:
  \begin{verbatim}
  insert w; [ ]; \textbf{assert} (w \in \text{get()})
  \end{verbatim}

  around the program is handled \textit{similarly easily} by
  \begin{verbatim}
  insert w; \textbf{witness} ($\lambda s. w \in s$); [ ]; \textbf{recall} ($\lambda s. w \in s$); \textbf{assert} (w \in \text{get()})
  \end{verbatim}

- \textbf{Monotonic counters} are analogous, by picking $\mathbb{N}$ and \texttt{\leq}, e.g.,
  \begin{verbatim}
  create 0; incr(); \textbf{witness} ($\lambda c. c > 0$); cp(); \textbf{recall} ($\lambda c. c > 0$)
  \end{verbatim}
The motivating example revisited

- Recall the program operating on the **set-valued state**
  
  \[
  \text{insert } v; \text{ complex\_procedure}(); \text{ assert } (v \in \text{get}())
  \]

- We pick **set inclusion** \( \subseteq \) as our preorder \( \text{rel on states} \)

- We **prove the assertion** by inserting a witness and recall

\[
\text{insert } v; \text{ witness } (\lambda s. v \in s); \text{ c\_p}(); \text{ recall } (\lambda s. v \in s); \text{ assert } (v \in \text{get}())
\]

- For any other \( w \), wrapping

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

around the program is handled similarly easily by

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

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

\[
\text{create } 0; \text{ incr}(); \text{ witness } (\lambda c. c > 0); \text{ c\_p}(); \text{ recall } (\lambda c. c > 0)
\]
The motivating example revisited

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

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

• We pick **set inclusion** \( \subseteq \) as our preorder \( \preceq \) on states

• We **prove the assertion** by inserting a witness and recall

```plaintext
insert v; witness (\lambda s. v ∈ s); c.p(); recall (\lambda s. v ∈ s); assert (v ∈ get())
```

• For **any other** \( w \), wrapping

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

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

```plaintext
insert w; witness (\lambda s. w ∈ s); [ ]; recall (\lambda s. w ∈ s); assert (w ∈ get())
```

• **Monotonic counters** are analogous, by picking \( \mathbb{N} \) and \( \leq \), e.g.,

```plaintext
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**
  \[
  \text{insert } v; \text{ complex\_procedure}(); \text{ assert } (v \in \text{get}());
  \]

- We pick **set inclusion** $\subseteq$ as our preorder rel on states

- We **prove the assertion** by inserting a witness and recall
  \[
  \text{insert } v; \text{ witness } (\lambda s. v \in s); \text{ c\_p}(); \text{ recall } (\lambda s. v \in s); \text{ assert } (v \in \text{get}());
  \]

- For **any other** $w$, wrapping
  \[
  \text{insert } w; [ ]; \text{ assert } (w \in \text{get}());
  \]
  around the program is handled **similarly easily** by
  \[
  \text{insert } w; \text{ witness } (\lambda s. w \in s); [ ]; \text{ recall } (\lambda s. w \in s); \text{ assert } (w \in \text{get}());
  \]

- **Monotonic counters** are analogous, by picking $\mathbb{N}$ and $\leq$, e.g.,
  \[
  \text{create } 0; \text{ incr}(); \text{ witness } (\lambda c. c > 0); \text{ c\_p}(); \text{ recall } (\lambda c. c > 0)
  \]
ML-style typed references (local state)

• First, we define a type of **heaps** as a finite map
  
  ```
  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 a **preorder** on heaps (**heap inclusion**)
  
  ```
  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** as a finite map

```
  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 a **preorder** on heaps (**heap inclusion**):

```
  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** as a finite map

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

where

```haskell
type cell =
     | Unused : cell
     | Used : a : Type → v : a → cell
```

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

```haskell
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*

\[
\text{MLST } t \text{ pre post } \overset{\text{def}}{=} \text{MST}_{\text{heap,heap_inclusion}} t \text{ pre post}
\]

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

\[
\text{abstract type ref } a = \text{id:N}\{\text{witnessed } (\lambda h. \text{contains } h \text{ id } a)\}
\]

where

\[
\text{let contains } (H \ h \ _) \text{ id } a =
\]

\[
\text{match } h \text{ id with}
\]

\[
| \text{Used } b \_ \rightarrow a = b
\]

\[
| \text{Unused } \rightarrow \bot
\]

- Important: contains is *stable* wrt. heap_inclusion
ML-style typed references (local state)

• As a result, we can define new local state effect

\[
\text{MLST} \ t \ \text{pre} \ \text{post} \ \overset{\text{def}}{=} \ \text{MST}_{\text{heap},\text{heap\_inclusion}} \ t \ \text{pre} \ \text{post}
\]

• Next, we define the type of references using monotonicity

\[
\text{abstract type } \text{ref} \ a = \text{id} : \text{N}\{\text{witnessed} (\lambda h. \text{contains} h \text{ id} a)\}
\]

where

\[
\text{let contains } (H \ h \ _) \text{ id} a =
\]

match h \text{ id} with

| Used \ b \ _ \ \rightarrow \ a = b
| Unused \ \rightarrow \ \bot

• Important: \text{contains is stable wrt. heap\_inclusion}
ML-style typed references (local state)

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

\[
\text{MLST } t \text{ pre post } \overset{\text{def}}{=} \text{MST}_{\text{heap,heap_inclusion}} t \text{ pre post}
\]

- Next, we define the type of references using monotonicity

\[
\text{abstract type } \text{ref a } = \text{id:}\mathbb{N}\{\text{witnessed } (\lambda h. \text{contains } h \text{id a})\}
\]

where

\[
\text{let contains (H h _)} \text{id a } =
\begin{align*}
&\text{match h id with} \\
&| \text{Used b _ } \rightarrow a = b \\
&| \text{Unused } \rightarrow \bot
\end{align*}
\]

- Important: contains is stable wrt. heap_inclusion
ML-style typed references (local state)

• As a result, we can define new local state effect

\[ \text{MLST} \ t \ pre \ post \defeq \text{MST}_{\text{heap,heap\_inclusion}} \ t \ pre \ post \]

• Next, we define the type of references using monotonicity

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

where

let contains (H h _) id a =

match h id with

| Used b _ \rightarrow a = b
| Unused \rightarrow \bot

• 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) (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) (v:a): MLST (ref a) \ldots = \ldots
    
    - 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 \ldots = \ldots
    
    - 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 \ldots = \ldots
    
    - 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
    ```
    | 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) with strong updates
  - 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) with strong updates
  - Used heap cells are extended with **tags**
    - Used : a:Type → v:a → t:tag → cell
      - 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
      - type tag a = Typed: rel:preorder a → tag a | Untyped:tag a
  - mrefs provide **more flexibility** with ref.-wise monotonicity
Conclusion

• Monotonicity
  • can be distilled into a **simple** and **general** framework
  • is **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
  • 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 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
  - 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!

Interested in doing an F* internship?

Get in touch with the F* team!

www.fstar-lang.org
Appendix: witnessed as a modality

- Part of ongoing work into improving mon. reification for MST

- state-indexed Kripke-semantics

\[ \llbracket \text{witnessed } p \rrbracket (s) \overset{\text{def}}{=} \forall s'. \text{rel } s s' \implies \llbracket p s' \rrbracket (s) \]

- Allows us to validate additional properties, such as

\[ p \iff \text{witnessed } (\text{fun } _\rightarrow p) \]

\[ \text{witnessed } p \iff \text{witnessed } (\text{fun } _\rightarrow \text{witnessed } p) \]

\[ \text{witnessed } p \land \text{witnessed } q \iff \text{witnessed } (\text{fun } s \rightarrow p s \land q s) \]

\[ \ldots \]