Leveraging monotonic state in F*

Danel Ahman @ INRIA Paris

joint work with

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

EUTypes Working Meeting
January 23, 2018
Global state + monotonicity is really useful!

Its essence can be captured very neatly!
Outline

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

• Monotonic state by example

• Key ideas behind our general framework

• Accommodating monotonic state in F*

• Some examples of monotonic state at work

• More examples of monotonic state at work (see POPL’18 paper)

• First steps in mon. reification and reflection (see POPL’18 paper)

• Meta-theory and correctness results (see POPL’18 paper)
Monotonicity in verification

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

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

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

  ```
  {λs. v ∈ s} complex_procedure() {λs. v ∈ s}
  ```

- Likely that we have to **carry λs. v ∈ s through** the proof of c_p
  - **Does not guarantee** that λs. v ∈ s holds at every point in c_p
  - **Sensitive** to proving that c_p maintains λs. w ∈ s for some other w

- However, if c_p **never removes**, then λs. v ∈ s is **stable**, and we would like the program logic to give us v ∈ get() “for free”
Monotonicity in verification

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

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

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

  ```
  {λs.v ∈ s} complex_procedure() {λs.v ∈ s}
  ```

- Likely that we have to **carry λs.v ∈ s through** the proof of `c_p`

- **Does not** guarantee that `λs.v ∈ s` holds at every point in `c_p`

- **Sensitive** to proving that `c_p` maintains `λs.w ∈ s` for some other `w`

- However, if `c_p` **never removes**, then `λs.v ∈ s` is **stable**, and we would like the program logic to give us `v ∈ get()” for free”`
Monotonicity in verification

- Consider a program operating on **set-valued state**
  
  ```
  insert v; complex_procedure(); assert (v ∈ get())
  ```

- To prove the assertion (say, in a Floyd-Hoare style logic), we could prove that the code maintains a **stateful invariant**
  
  ```
  \{ \lambda s. v ∈ s \} complex_procedure() \{ \lambda s. v ∈ s \}
  ```

- Likely that we have to **carry \( \lambda s. v ∈ s \)** through the proof of \( c_p \)
  - Does not guarantee that \( \lambda s. v ∈ s \) holds at every point in \( c_p \)
  - Sensitive to proving that \( c_p \) maintains \( \lambda s. w ∈ s \) for some other \( w \)

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

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

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

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

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

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

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

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

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

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

- Consider ML-style typed references \( r : \text{ref } a \)
  - \( r \) 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})

- See our POPL 2018 paper for more
  - temporarily \textit{violating monotonicity} via snapshots
  - two substantial case studies in F*
    - a secure file-transfer application
    - Ariadne \texttt{state continuity} protocol \cite{StrackxPiessens16}
  - pointers to other works in F* relying on monotonicity for
    - sophisticated \texttt{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 (\texttt{ref t}) and untyped references (\texttt{uref})
  - more flexibility with monotonic references (\texttt{mref t rel})

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

- In this talk
  - our motivating example and monotonic counters
  - typed references \((\text{ref} \ t)\) and untyped references \((\text{uref})\)
  - more flexibility with monotonic references \((\text{mref} \ t \ \text{rel})\)

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

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

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

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

- Provides a unifying account of the existing ad hoc uses in F*
Key ideas behind our general framework

- We make use of **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** $\text{rel}$ 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 make use of **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
    \[
    \forall s e' s'. (e, s) \leadsto^* (e', s') \Rightarrow \text{rel} \; s \; s'
    \]
  - a stateful predicate \( p \) is **stable** (wrt. \( \text{rel} \)) when
    \[
    \forall s s'. p \; s \land \text{rel} \; s \; s' \Rightarrow p \; s'
    \]
- **Our solution**: extend Hoare-style program logics (e.g., F*) with
  - a means to **witness** the validity of \( p \; s \) in some state \( s \)
  - a means for turning a \( p \) into a **state-independent proposition**
  - a means to **recall** the validity of \( p \; s' \) in any future state \( s' \)
- Provides a **unifying account** of the existing ad hoc uses in F*
Key ideas behind our general framework

• We make use of **monotonic programs** and **stable predicates**
  • per verification task, we **choose a preorder** $\text{rel}$ 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 make use of 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 make use of **monotonic programs** and **stable predicates**
  - per verification task, we **choose a preorder** \( \text{rel} \) on states
    - set inclusion, heap inclusion, increasing counter values, \ldots
  - 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 make use of **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
    \[
    \forall s \ e' s'. \ (e, s) \xrightarrow{*} (e', s') \implies \text{rel } s s'
    \]
  - 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 *ad hoc* uses in $\text{F*}$
Outline

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

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

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

where

\[
\text{pre} : \text{state} \rightarrow \text{Type}_0 \quad \text{post} : \text{state} \rightarrow t \rightarrow \text{state} \rightarrow \text{Type}_0
\]

- ST is an abstract pre-postcondition refinement of

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

- The global state `actions` have types

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

- Refs. and local state are defined in F* using monotonicity
Recap: Ordinary global state in F*

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

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

where

\[
\begin{align*}
\text{pre} &: \text{state} \to \text{Type}_0 & \text{post} &: \text{state} \to t \to \text{state} \to \text{Type}_0
\end{align*}
\]

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

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

- The global state **actions** have types

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

- Refs. and local state are defined in F* using **monotonicity**
Recap: Ordinary global state in F*

- F* supports Hoare-style reasoning about state via the **comp. type**
  \[ ST_{\text{state}} \ t \ (\text{requires } \text{pre}) \ (\text{ensures } \text{post}) \]

  where

  \[
  \text{pre} : \text{state} \rightarrow \text{Type}_0 \quad \text{post} : \text{state} \rightarrow \text{t} \rightarrow \text{state} \rightarrow \text{Type}_0
  \]

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

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

- Refs. and local state are defined in F* using monotonicity
Recap: Ordinary global state in F*

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

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

where

\[
\text{pre} : \text{state} \rightarrow \text{Type}_0 \quad \text{post} : \text{state} \rightarrow t \rightarrow \text{state} \rightarrow \text{Type}_0
\]

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

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

- The global state **actions** have types

\[
\begin{align*}
g\text{et} &: \text{unit} \rightarrow \text{ST} \ \text{state} \ (\text{requires} \ (\lambda \_.\top)) \ (\text{ensures} \ (\lambda s_0 s s_1. s_0 = s = s_1)) \\
p\text{ut} &: s:\text{state} \rightarrow \text{ST} \ \text{unit} \ (\text{requires} \ (\lambda \_.\top)) \ (\text{ensures} \ (\lambda _-_s_1. s_1 = s))
\end{align*}
\]

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

- We capture monotonic state with a new computational type

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

- 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

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

- We capture monotonic state with a new computational type

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

- The get action is typed as in ST

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

- To ensure monotonicity, the put action gets a precondition

\[
\begin{align*}
\text{put : s:state} & \to \text{MST unit} \ (\text{requires } (\lambda s_0 . \text{rel } s_0 s)) \\
& \quad \quad \quad \ (\text{ensures } (\lambda . s_1. s_1 = s))
\end{align*}
\]

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

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

- We capture monotonic state with a new computational type

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

- The \textbf{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 \textbf{monotonicity}, the \textbf{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, \textbf{MST} is an \textbf{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

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

• The get action is typed as in ST

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

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

• To ensure monotonicity, the put action gets a precondition

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

$$\text{(ensures} \ (\lambda _- s_1 . s_1 = s)\text{)}$$

• 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

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

- The get action is typed as in ST

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

- To ensure monotonicity, the put action gets a precondition

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

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

\[
\text{mst} t \overset{\text{def}}{=} s_0:\text{state} \to t \ast s_1:\text{state}\{\text{rel } s_0 s_1\}
\]
New: Recalling a Witness

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

\[ \text{witnessed} : (\text{state} \to \text{Type}_0) \to \text{Type}_0 \]

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

\[ \text{wk} : p, q : (\text{state} \to \text{Type}_0) \to \text{Lemma} \left( \text{requires} \left( \forall s. p\ s \implies q\ s \right) \right) \]

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

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

\[ \left[ \text{witnessed}\ p \right](s) \stackrel{\text{def}}{=} \forall s'. \text{rel}\ s\ s' \implies \left[ p\ s' \right](s) \]

- As usual, for natural deduction, need **world-indexed sequents**

- Oh, wait a minute ...
New: Recalling a Witness

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

\[
\text{witnessed} : (\text{state} \rightarrow \text{Type}_0) \rightarrow \text{Type}_0
\]

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

\[
\text{wk} : p, q : (\text{state} \rightarrow \text{Type}_0) \rightarrow \text{Lemma (requires \( \forall s. p s \implies q s \))}
\]
\[
(\text{ensures \( \text{witnessed p} \implies \text{witnessed q} \))}
\]

• Intuitively, a lot like the **necessity modality** □

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

• As usual, for natural deduction, need **world-indexed sequents**

• Oh, wait a minute ...
New: Recalling a Witness

• We extend F* with a logical capability

\[ \text{witnessed} : (\text{state} \to \text{Type}_0) \to \text{Type}_0 \]

together with a \textbf{weakening principle (functoriality)}

\[ \text{wk} : p,q:(\text{state} \to \text{Type}_0) \to \text{Lemma} (\text{requires} (\forall s. p\ s \Rightarrow q\ s)) \]
\[ (\text{ensures} (\text{witnessed} p \Rightarrow \text{witnessed} q)) \]

• Intuitively, a lot like the \textbf{necessity modality} \( \Box \)

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

• As usual, for natural deduction, need \textbf{world-indexed sequents}

• Oh, wait a minute …
New: Recalling a Witness

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

\[
\text{witnessed} : (\text{state} \to \text{Type}_0) \to \text{Type}_0
\]

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

\[
wk : p, q : (\text{state} \to \text{Type}_0) \to \text{Lemma}\ (\text{requires} \ (\forall s. p\ s \implies q\ s))
\]

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

- Intuitively, a lot like the **necessity modality** □

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

- As usual, for natural deduction, need **world-indexed sequents**

- Oh, wait a minute . . .
New: Recalling a Witness

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

\[
\text{witnessed} : (\text{state} \to \text{Type}_0) \to \text{Type}_0
\]

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

\[
\text{wk} : p, q : (\text{state} \to \text{Type}_0) \to \text{Lemma} (\text{requires} (\forall s. p \ s \Rightarrow q \ s))
\]
\[
(\text{ensures} (\text{witnessed} p \Rightarrow \text{witnessed} q))
\]

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

\[
\left[\text{witnessed} \ p\right](s) \overset{\text{def}}{=} \forall s'. \text{rel} \ s \ s' \Rightarrow \left[\ p \ s'\right](s)
\]

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

• ... Hoare-style logics are essentially world/state-indexed, so

• we include a stateful introduction rule for witnessed

\[
\text{witness} : \ p : (\text{state} \to \text{Type}_0) \\
\to \text{MST unit (requires } (\lambda s_0. p \ 'stable\_from' s_0)) \\
(\text{ensures } (\lambda s_0 s_1. s_0 = s_1 \land \text{witnessed } p))
\]

• and a stateful elimination rule for witnessed

\[
\text{recall} : \ p : (\text{state} \to \text{Type}_0) \\
\to \text{MST unit (requires } (\lambda _. \text{witnessed } p)) \\
(\text{ensures } (\lambda s_0 s_1. s_0 = s_1 \land p \ 'stable\_from' s_1))
\]
New: Recalling a Witness

- Hoare-style logics are essentially \textbf{world/state-indexed}, so

- we include a \textbf{stateful introduction rule} for witnessed

\[
\text{witness} : \ p : (\text{state} \rightarrow \text{Type}_0) \\
\quad \rightarrow \ \text{MST unit (requires } (\lambda s_0 . p \ '\text{stable from} \ ' s_0)) \\
\quad \quad \quad \text{(ensures } (\lambda s_0 \_ s_1 . s_0 = s_1 \land \text{witnessed } p))
\]

- and a \textbf{stateful elimination rule} for witnessed

\[
\text{recall} : \ p : (\text{state} \rightarrow \text{Type}_0) \\
\quad \rightarrow \ \text{MST unit (requires } (\lambda \_ . \text{witnessed } p)) \\
\quad \quad \quad \text{(ensures } (\lambda s_0 \_ s_1 . s_0 = s_1 \land p \ '\text{stable from} \ ' s_1))
\]
New: Recalling a Witness

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

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

  \[
  \text{witness} : \ p : (\text{state} \to \text{Type}_0) \\
  \quad \to \ \text{MST unit} \ (\text{requires} \ (\lambda s_0. p \ 'stable\_from' \ s_0)) \\
  \quad \quad \ (\text{ensures} \ (\lambda s_0 s_1. s_0 = s_1 \land \text{witnessed} \ p))
  \]

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

  \[
  \text{recall} : \ p : (\text{state} \to \text{Type}_0) \\
  \quad \to \ \text{MST unit} \ (\text{requires} \ (\lambda _ . \text{witnessed} \ p)) \\
  \quad \quad \ (\text{ensures} \ (\lambda s_0 s_1. s_0 = s_1 \land p \ 'stable\_from' \ s_1))
  \]
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 POPL’18 paper)
- First steps in mon. reification and reflection (see POPL’18 paper)
- Meta-theory and correctness results (see POPL’18 paper)
The motivating example revisited

- Recall the program operating on the set-valued state

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

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

- We prove the assertion by inserting a witness and recall

```plaintext
insert v; witness (\(\lambda s. v \in s\)); c.p(); recall (\(\lambda s. v \in s\)); 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 (\(\lambda s. w \in s\)); [ ]; recall (\(\lambda s. w \in s\)); 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 (\(\lambda c. c > 0\))
```
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 \( \operatorname{rel} \) 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

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

- 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.,

\[
\text{create } 0; \text{ incr}(); \text{ witness } (\lambda c. c > 0); \text{ c\_p}(); \text{ recall } (\lambda c. c > 0)
\]
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 (\( \lambda s. v ∈ s \)); c.p(); recall (\( \lambda s. v ∈ s \)); assert (v ∈ get())
  ```

- For **any other** \( w \), wrapping

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

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

  ```
  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.,

  ```
  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.,

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

  type cell =
    | Unused : cell
    | Used : a:Type₀ → 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 _ _ → T
    | Unused, Unused → T
    | Used _, Unused → ⊥
```
ML-style typed references (local state)

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

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

  where
  type cell =
  | Unused : cell
  | Used : a : Type₀ → v : a → cell
  ```

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

  ```ml
  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

  type cell =
  | Unused : cell
  | Used : a : Type₀ → 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

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

• 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 _ → 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**

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

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

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

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

- 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

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

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

| \text{Used } b\ _ \rightarrow a = b  \\
| \text{Unused} \rightarrow \bot

- Important: contains is stable wrt. heap_inclusion
Finally, we define MLST’s actions using MST’s actions

- let alloc (a: Type_0) (v:a): MLST (ref a) ... = ...
  - get the current heap
  - create a fresh ref., and add it to the heap
  - put the updated heap back
  - witness that the created ref. is in the heap

- let read (r:ref a): MLST t ... = ...
  - recall that the given ref. is in the heap
  - get the current heap
  - select the given reference from the heap

- let write (r:ref a) (v:a): MLST unit ... = ...
  - recall that the given ref. is in the heap
  - get the current heap
  - update the heap with the given value at the given ref.
  - put the updated heap back
Finally, we define \texttt{MLST}'s \textbf{actions} using \texttt{MST}'s actions

\begin{itemize}
  \item \texttt{let alloc (a:Type_0) (v:a): MLST (ref a) \ldots = \ldots}
    \begin{itemize}
    \item \texttt{get} the current heap
    \item \texttt{create} a fresh ref., and \texttt{add} it to the heap
    \item \texttt{put} the updated heap back
    \item \texttt{witness} that the created ref. is in the heap
    \end{itemize}
  
  \item \texttt{let read (r:ref a): MLST t \ldots = \ldots}
    \begin{itemize}
    \item \texttt{recall} that the given ref. is in the heap
    \item \texttt{get} the current heap
    \item \texttt{select} the given reference from the heap
    \end{itemize}
  
  \item \texttt{let write (r:ref a) (v:a): MLST unit \ldots = \ldots}
    \begin{itemize}
    \item \texttt{recall} that the given ref. is in the heap
    \item \texttt{get} the current heap
    \item \texttt{update} the heap with the given value at the given ref.
    \item \texttt{put} the updated heap back
    \end{itemize}
\end{itemize}
Adding untyped and monotonic references

- **Untyped references** \((\texttt{uref})\) with strong updates
  - Used heap cells are extended with **tags**
    
    \[
    \text{Used} : a : \text{Type}_0 \rightarrow v : a \rightarrow t : \text{tag} \rightarrow \text{cell}
    \]

    where

    \[
    \text{type tag} = \text{Typed : tag} \mid \text{Untyped : tag}
    \]

  - actions corresponding to \texttt{urefs} have **weaker types** than for \texttt{refs}

- **Monotonic references** \((\texttt{mref} \ a \ \text{rel})\)
  - Used heap cells are extended with **typed tags**
    
    \[
    \text{Used} : a : \text{Type}_0 \rightarrow v : a \rightarrow t : \text{tag} \ a \rightarrow \text{cell}
    \]

    where

    \[
    \text{type tag} a = \text{Typed : rel : preorder} \ a \rightarrow \text{tag} \ a \mid \text{Untyped : tag} \ a
    \]

  - \texttt{mrefs} provide **more flexibility** with ref.-wise monotonicity

- Further, all three can be extended with **manually managed refs.**
Adding untyped and monotonic references

- **Untyped references** (uref) with strong updates
  - Used heap cells are extended with **tags**
    
    \[
    \text{Used : } a: \text{Type}_0 \rightarrow v:a \rightarrow t: \text{tag} \rightarrow \text{cell}
    \]
    
    where
    
    \[
    \text{type tag } = \text{Typed : tag } | \text{ Untyped : tag}
    \]
  
  - actions corresponding to urefs have **weaker types** than for refs

- **Monotonic references** (mref a rel)
  
  - Used heap cells are extended with **typed tags**
    
    \[
    \text{Used : } a: \text{Type}_0 \rightarrow v:a \rightarrow t: \text{tag} a \rightarrow \text{cell}
    \]
    
    where
    
    \[
    \text{type tag a } = \text{Typed : rel : preorder a } \rightarrow \text{tag a } | \text{ Untyped : tag a}
    \]

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

- Further, all three can be extended with **manually managed refs.**
Adding untyped and monotonic references

- **Untyped references** (*uref*) with strong updates
  - Used heap cells are extended with *tags*
    
    \[
    \text{Used : } a : \text{Type}_0 \rightarrow v : a \rightarrow t : \text{tag} \rightarrow \text{cell}
    \]
    
    where
    \[
    \text{type tag } = \text{Typed : tag} \mid \text{Untyped : tag}
    \]
  - Actions corresponding to *urefs* have **weaker types** than for *refs*

- **Monotonic references** (*mref a rel*)
  - Used heap cells are extended with *typed tags*
    
    \[
    \text{Used : } a : \text{Type}_0 \rightarrow v : a \rightarrow t : \text{tag} \rightarrow \text{cell}
    \]
    
    where
    \[
    \text{type tag a } = \text{Typed : rel : preorder a } \rightarrow \text{tag a} \mid \text{Untyped : tag a}
    \]
  - *mrefs* provide **more flexibility** with ref.-wise monotonicity

- Further, all three can be extended with manually managed refs.
Adding untyped and monotonic references

- **Untyped references** (uref) with strong updates
  - Used heap cells are extended with tags
    
    \[
    \begin{array}{c}
    \text{Used} : \text{a:Type}_0 \rightarrow \text{v:a} \rightarrow \text{t:tag} \rightarrow \text{cell}
    \\
    \text{where}
    \\
    \text{type tag} = \text{Typed : tag} \mid \text{Untyped : tag}
    \end{array}
    \]
  
  - actions corresponding to urefs have **weaker types** than for refs

- **Monotonic references** (mref a rel)
  
  - Used heap cells are extended with **typed tags**
    
    \[
    \begin{array}{c}
    \text{Used} : \text{a:Type}_0 \rightarrow \text{v:a} \rightarrow \text{t:tag} \rightarrow \text{cell}
    \\
    \text{where}
    \\
    \text{type tag a} = \text{Typed : rel:preorder a} \rightarrow \text{tag a} \mid \text{Untyped : tag a}
    \end{array}
    \]

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

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

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

- See our POPL 2018 paper for
  - further **examples** and **case studies**
  - meta-theory and **total correctness** for MST
    - based on an instrumented operational semantics
      \[(\text{witness } x.\varphi, s, W) \leadsto (\text{return } (), s, W \cup \{x.\varphi\})\]
    - 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 our POPL 2018 paper for
  - further **examples** and **case studies**
  - **meta-theory** and **total correctness** for **MST**
    - based on an instrumented operational semantics
      \[(witness \ x.\varphi, s, W) \leadsto (\text{return}(), s, W \cup \{x.\varphi\})\]
    - 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