Recalling a Witness

Foundations and Applications of Monotonic State

Danel Ahman

Prosecco Team, INRIA Paris

Cătălin Hriţcu 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)
- Meta-theory and correctness results (see the paper)
- First steps towards monadic reification (see the paper)

Outline

- Monotonic state by example
- Key ideas behind our approach
- Accommodating monotonic state in F*
- Examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Meta-theory and correctness results (see the paper)
- First steps towards monadic reification (see the paper)

Monotonic state and program verification

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

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

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

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

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

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

Monotonic state and program verification

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

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

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

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

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

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

Monotonic state and program verification

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

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

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

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

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

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

Monotonic state and program verification

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

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

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

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

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

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

Monotonicity is really useful!

- To come later in this talk
  - **motivating example** and **monotonic counters**
  - both **typed** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)
- For more examples, see the paper
  - temporarily performing **non-stable updates** via snapshots
  - two substantial case studies
    - a **secure file-transfer** case study
    - 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!

- To come later in this talk
  - **motivating example** and **monotonic counters**
  - both **typed** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)
- For more examples, see the paper
  - temporarily performing **non-stable updates** via snapshots
  - two substantial case studies
    - a **secure file-transfer** case study
    - 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!

- To come later in this talk
  - **motivating example** and **monotonic counters**
  - both **typed** (`ref t`) and **untyped references** (`uref`)
  - more flexibility with **monotonic references** (`mref t rel`)
- For more examples, see the paper
  - temporarily performing **non-stable updates** via snapshots
  - two substantial case studies
    - a **secure file-transfer** case study
    - 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*
- Examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Meta-theory and correctness results (see the paper)
- First steps towards monadic reification (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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F*) with
  - a means for turning a `p` into a **state-independent proposition**
  - an operation to **witness** the validity of `p` s in some state s
  - an operation to **recall** the validity of `p` s′ in a 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** $\texttt{rel}$ on states
    - set inclusion, heap inclusion, increasing counter values, ...
  - a stateful program $\mathtt{e}$ is **monotonic** (wrt. $\texttt{rel}$) when
    $$\forall\, \mathtt{s}\, \mathtt{s}'\, \mathtt{e}'.\ (\mathtt{s}, \mathtt{e}) \rightsquigarrow^* (\mathtt{s}', \mathtt{e}') \implies \texttt{rel}\ \mathtt{s}\ \mathtt{s}'$$
  - a stateful predicate $\texttt{p}$ is **stable** (wrt. $\texttt{rel}$) when
    $$\forall\, \mathtt{s}\, \mathtt{s}'.\ \texttt{p}\ \mathtt{s}\ \wedge\ \texttt{rel}\ \mathtt{s}\ \mathtt{s}' \implies \texttt{p}\ \mathtt{s}'$$
- **Our solution:** extend Hoare-style program logics (e.g., F*) with
  - a means for turning a $\texttt{p}$ into a **state-independent proposition**
  - an operation to **witness** the validity of $\texttt{p}\ \mathtt{s}$ in some state $\mathtt{s}$
  - an operation to **recall** the validity of $\texttt{p}\ \mathtt{s}'$ in a future state $\mathtt{s}'$
- 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*
- Examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Meta-theory and correctness results (see the paper)
- First steps towards monadic reification (see the paper)

Recap: Ordinary global state in F*

- F* is an ML-like dependently typed language, aimed at verification

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

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

  where

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

  (formally, this type is derived from a WP calculus for state)

- The global state actions **get** and **put** have the following types

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

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

- **Local state** will be defined in F* using monotonicity

Recap: Ordinary global state in F*

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

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

  where

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

  (formally, this type is derived from a WP calculus for state)

- The global state actions **get** and **put** have the following types

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

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

- **Local state** will be defined in F* using monotonicity

Recap: Ordinary global state in F*

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

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

  where

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

  (formally, this type is derived from a WP calculus for state)

- The global state actions **get** and **put** have the following types

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

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

- **Local state** will be defined in F* using monotonicity

Recap: Ordinary global state in F*

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

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

  where

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

  (formally, this type is derived from a WP calculus for state)

- The global state actions **get** and **put** have the following types

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

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

- **Local state** will be defined in F* using monotonicity

New: Monotonic state in F*

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

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

  where t, pre, and post are typed as in ST

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

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

- To ensure **monotonicity**, the **put** action now gets the type

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

New: Monotonic state in F*

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

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

  where t, pre, and post are typed as in ST

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

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

- To ensure **monotonicity**, the **put** action now gets the type

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

New: Monotonic state in F*

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

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

  where t, pre, and post are typed as in ST

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

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

- To ensure **monotonicity**, the **put** action now gets the type

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

New: Monotonic state in F*

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

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

  where t, pre, and post are typed as in ST

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

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

- To ensure **monotonicity**, the **put** action now gets the type

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

New: Monotonic state in F*

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

  $\texttt{witnessed} : \texttt{pred state} \to \texttt{Type} \qquad (\texttt{pred state} \stackrel{\text{def}}{=} \texttt{state} \to \texttt{Type})$

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

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

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

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

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

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

New: Monotonic state in F*

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

  $\texttt{witnessed} : \texttt{pred state} \rightarrow \texttt{Type} \qquad (\texttt{pred state} \stackrel{\text{def}}{=} \texttt{state} \rightarrow \texttt{Type})$

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

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

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

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

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

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

New: Monotonic state in F*

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

$$\texttt{witnessed} : \texttt{pred state} \to \texttt{Type} \qquad (\texttt{pred state} \stackrel{\text{def}}{=} \texttt{state} \to \texttt{Type})$$

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

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

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

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

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

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

New: Monotonic state in F*

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

$$\texttt{witnessed} : \texttt{pred state} \rightarrow \texttt{Type} \qquad (\texttt{pred state} \stackrel{\text{def}}{=} \texttt{state} \rightarrow \texttt{Type})$$

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

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

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

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

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

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

Outline

- Monotonic state by example
- Key ideas behind our approach
- Accommodating monotonic state in F*
- Examples of monotonic state at work
- More examples of monotonic state at work (see the paper)
- Meta-theory and correctness results (see the paper)
- First steps towards monadic reification (see the paper)

The motivating example revisited

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

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

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

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

  - For any other `w`, wrapping

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

    around the program is handled similarly easily

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

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

The motivating example revisited

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

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

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

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

  - For any other `w`, wrapping

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

    around the program is handled similarly easily

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

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

The motivating example revisited

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

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

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

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

  - For any other w, wrapping

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

    around the program is handled similarly easily

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

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

The motivating example revisited

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

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

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

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

  - For any other `w`, wrapping

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

  around the program is handled similarly easily

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

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

The motivating example revisited

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

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

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

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

  - For any other `w`, wrapping

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

    around the program is handled similarly easily

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

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

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:(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**

  $\texttt{type heap} =$

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

  where

  $\texttt{type cell} =$

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

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

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

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

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

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

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

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

ML-style typed references (local state)

- 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**

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

- Also, we can define the type of **typed references**

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

  where

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

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

ML-style typed references (local state)

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

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

- Also, we can define the type of **typed references**

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

where

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

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

ML-style typed references (local state)

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

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

- Also, we can define the type of **typed references**

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

where

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

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

ML-style typed references (local state)

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

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

- Also, we can define the type of **typed references**

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

  where

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

- 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

Adding untyped and monotonic references

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

    where

    $$| \; \texttt{Used} : \texttt{a:Type} \rightarrow \texttt{v:a} \rightarrow \texttt{t:tag} \rightarrow \texttt{cell}$$

    $$\texttt{type tag} \; = \; \texttt{Typed} : \texttt{tag} \; | \; \texttt{Untyped} : \texttt{tag}$$
  - `uref` actions have correspondingly **weaker types**

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

    where

    $$| \; \texttt{Used} : \texttt{a:Type} \rightarrow \texttt{v:a} \rightarrow \texttt{t:tag a} \rightarrow \texttt{cell}$$

    $$\texttt{type tag a} \; = \; \texttt{Typed} : \texttt{rel:preorder a} \rightarrow \texttt{tag a} \; | \; \texttt{Untyped} : \texttt{tag a}$$
  - Provides **more flexibility** with ref.-wise `witness` and `recall`

Adding untyped and monotonic references

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

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

    where

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

  - `uref` actions have correspondingly **weaker types**

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

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

    where

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

  - Provides **more flexibility** with ref.-wise `witness` and `recall`

Adding untyped and monotonic references

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

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

    where

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

  - `uref` actions have correspondingly **weaker types**

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

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

    where

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

  - Provides **more flexibility** with ref.-wise `witness` and `recall`

Conclusion

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

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

$$(\texttt{witness}\ s.\varphi, \sigma, W) \leadsto (\texttt{return}\ (), \sigma, W \cup \{s.\varphi\})$$

    - and cut elimination for the logic with witnessed
  - first steps towards **monadic reification** for MST, based on

$$\texttt{m t} = \texttt{s}_0\texttt{:state} \to \texttt{t} * \texttt{s}_1\texttt{:state}\{\texttt{rel}\ \texttt{s}_0\ \texttt{s}_1\}$$

Conclusion

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

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

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

    - and cut elimination for the logic with `witnessed`
  - first steps towards **monadic reification** for MST, based on

$$\mathtt{m}\ \mathtt{t} = \mathtt{s_0{:}state} \rightarrow \mathtt{t} * \mathtt{s_1{:}state}\{\mathtt{rel}\ \mathtt{s_0}\ \mathtt{s_1}\}$$

Thank you!

Questions?