

Recalling a Witness

Foundations and Applications of Monotonic State

Danel Ahman

Prosecco Team at Inria Paris

joint work with

Cătălin Hrițcu and Kenji Maillard @ Inria Paris

Cédric Fournet, Aseem Rastogi, and Nikhil Swamy @ MSR

HOPE 2017

September 3, 2017

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F^*
- Examples of monotonic state at work
- A glimpse of the meta-theory

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F^*
- Examples of monotonic state at work
- A glimpse of the meta-theory

Monotonic state and program verification

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

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

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

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

- likely that we have to **carry** $\lambda s. v \in s$ **through** the proof of `c_p`
 - sensitive** to proving that `c_p` maintains $\lambda s. w \in s$ for some other `w`
 - does not guarantee** that $\lambda s. v \in s$ holds at every point in `c_p`
- However, if `c_p` **only inserts**, then $\lambda s. v \in s$ is **stable**, and we would like the program logic to give us $v \in \text{get()}$ “for free”

Monotonic state and program verification

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

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

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

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

- likely that we have to carry $\lambda s. v \in s$ through the proof of c_p
 - sensitive to proving that c_p maintains $\lambda s. w \in s$ for some other w
 - does not guarantee that $\lambda s. v \in s$ holds at every point in c_p
- However, if c_p **only inserts**, then $\lambda s. v \in s$ is **stable**, and we would like the program logic to give us $v \in \text{get}()$ “for free”

Monotonic state and program verification

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

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

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

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

- likely that we have to **carry** $\lambda s. v \in s$ **through** the proof of c_p
 - sensitive** to proving that c_p maintains $\lambda s. w \in s$ for some other w
 - does not guarantee** that $\lambda s. v \in s$ holds at every point in c_p
- However, if c_p **only inserts**, then $\lambda s. v \in s$ is **stable**, and we would like the program logic to give us $v \in \text{get}()$ “for free”

Monotonic state and program verification

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

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

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

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

- likely that we have to **carry** $\lambda s. v \in s$ **through** the proof of c_p
 - sensitive** to proving that c_p maintains $\lambda s. w \in s$ for some other w
 - does not guarantee** that $\lambda s. v \in s$ holds at every point in c_p
- However, if c_p **only inserts**, then $\lambda s. v \in s$ is **stable**, and we would like the program logic to give us $v \in \text{get}()$ “**for free**”

Monotonicity is really useful!

- To come later in this talk
 - reasoning about **monotonic counters**
 - implementing **typed** (`ref t`) and **untyped references** (`uref`)
 - more flexibility with **monotonic references** (`mref t rel`)
- For more examples, see

Recalling a Witness:
Foundations and Applications of Monotonic State
(arXiv:1707.02466)

which includes

- a secure **file-transfer** application
- pointers to works using monotonicity in **crypto** and **TLS** **verif.**
- Ariadne **state continuity** protocol [Strackx, Piessens 2016]

Monotonicity is really useful!

- To come later in this talk
 - reasoning about **monotonic counters**
 - implementing **typed** (`ref t`) and **untyped references** (`uref`)
 - more flexibility with **monotonic references** (`mref t rel`)
- For more examples, see

Recalling a Witness:
Foundations and Applications of Monotonic State
(arXiv:1707.02466)

which includes

- a secure **file-transfer** application
- pointers to works using monotonicity in **crypto** and **TLS** **verif.**
- Ariadne **state continuity** protocol [Strackx, Piessens 2016]

Monotonicity is really useful!

- To come later in this talk
 - reasoning about **monotonic counters**
 - implementing **typed** (`ref t`) and **untyped references** (`uref`)
 - more flexibility with **monotonic references** (`mref t rel`)
- For more examples, see

Recalling a Witness:
Foundations and Applications of Monotonic State
([arXiv:1707.02466](https://arxiv.org/abs/1707.02466))

which includes

- a secure **file-transfer** application
- pointers to works using monotonicity in **crypto** and **TLS verif.**
- Ariadne **state continuity** protocol [Strackx, Piessens 2016]

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F^*
- Examples of monotonic state at work
- A glimpse of the meta-theory

Overview of our solution

- We focus on **monotonic** programs and **stable** predicates
 - per verification task, we choose a **preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counters, ...

- a program e is **monotonic** (wrt. **rel**) when

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - means for turning a p into a **state-independent proposition**
 - operation to **witness** the validity of $p \ s$ in some state s
 - operation to **recall** the validity of $p \ s'$ in a future state s'
- A **unifying account** of the ad hoc uses of monotonicity in F^*

Overview of our solution

- We focus on **monotonic** programs and **stable** predicates
 - per verification task, we choose a **preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counters, ...
 - a program e is **monotonic** (wrt. **rel**) when
$$(s, e) \rightsquigarrow^* (s', e') \implies \text{rel } s \ s'$$
 - a predicate p on states is **stable** (wrt. **rel**) when
$$\forall s \ s'. \ p \ s \ \wedge \ \text{rel } s \ s' \implies p \ s'$$
- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - means for turning a p into a **state-independent proposition**
 - operation to **witness** the validity of $p \ s$ in some state s
 - operation to **recall** the validity of $p \ s'$ in a future state s'
- A **unifying account** of the ad hoc uses of monotonicity in F^*

Overview of our solution

- We focus on **monotonic** programs and **stable** predicates
 - per verification task, we choose a **preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counters, ...

- a program e is **monotonic** (wrt. **rel**) when

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - means for turning a p into a **state-independent proposition**
 - operation to **witness** the validity of $p \ s$ in some state s
 - operation to **recall** the validity of $p \ s'$ in a future state s'
- A **unifying account** of the ad hoc uses of monotonicity in F^*

Overview of our solution

- We focus on **monotonic** programs and **stable** predicates
 - per verification task, we choose a **preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counters, ...

- a program e is **monotonic** (wrt. **rel**) when

$$(s, e) \rightsquigarrow^* (s', e') \implies \mathbf{rel} \ s \ s'$$

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - means for turning a p into a **state-independent proposition**
 - operation to **witness** the validity of $p \ s$ in some state s
 - operation to **recall** the validity of $p \ s'$ in a future state s'
- A **unifying account** of the ad hoc uses of monotonicity in F^*

Overview of our solution

- We focus on **monotonic** programs and **stable** predicates
 - per verification task, we choose a **preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counters, ...

- a program e is **monotonic** (wrt. **rel**) when

$$(s, e) \rightsquigarrow^* (s', e') \implies \mathbf{rel} \ s \ s'$$

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - means for turning a p into a **state-independent proposition**
 - operation to **witness** the validity of $p \ s$ in some state s
 - operation to **recall** the validity of $p \ s'$ in a future state s'
- A **unifying account** of the ad hoc uses of monotonicity in F^*

Overview of our solution

- We focus on **monotonic** programs and **stable** predicates
 - per verification task, we choose a **preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counters, ...

- a program e is **monotonic** (wrt. **rel**) when

$$(s, e) \rightsquigarrow^* (s', e') \implies \mathbf{rel} \ s \ s'$$

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

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

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

- A unifying account of the ad hoc uses of monotonicity in F^*

Overview of our solution

- We focus on **monotonic** programs and **stable** predicates
 - per verification task, we choose a **preorder** **rel** on states
 - set inclusion, heap inclusion, increasing counters, ...

- a program e is **monotonic** (wrt. **rel**) when

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

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

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

- **Our solution:** extend Hoare-style program logics (e.g., F^*) with
 - means for turning a p into a **state-independent proposition**
 - operation to **witness** the validity of $p \ s$ in some state s
 - operation to **recall** the validity of $p \ s'$ in a future state s'
- A **unifying account** of the ad hoc uses of monotonicity in F^*

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F^*
- Examples of monotonic state at work
- A glimpse of the meta-theory

Reasoning about ordinary state in F*

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

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

where

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

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

- The **get** and **put** actions are typed as follows

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

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

Reasoning about ordinary state in F*

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

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

where

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

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

- The **get** and **put** actions are typed as follows

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

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

Reasoning about ordinary state in F*

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

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

where

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

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

- The **get** and **put** actions are typed as follows

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

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

Reasoning about monotonic state in F^*

Reasoning about monotonic state in F*

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

`MST rel t (requires pre) (ensures post)`

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

- The `get` action is typed as in `ST`
- To ensure **monotonicity**, the `put` action is typed as follows

```
put : s:state → MST unit (requires (λ s0.rel s0 s))  
                               (ensures (λ _ s1. s1 = s))
```

- thus `MST` is a bit like an **update monad** [A., Uustalu'14]

Reasoning about monotonic state in F*

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

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

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

- The **get** action is typed as in ST
- To ensure **monotonicity**, the **put** action is typed as follows

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

- thus MST is a bit like an **update monad** [A., Uustalu'14]

Reasoning about monotonic state in F*

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

`MST rel t (requires pre) (ensures post)`

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

- The **get** action is typed as in `ST`
- To ensure **monotonicity**, the **put** action is typed as follows

`put : s:state → MST unit (requires (λ s0. red s0 s))`
`(ensures (λ _ s1. s1 = s))`

- thus `MST` is a bit like an **update monad** [A., Uustalu'14]

Reasoning about monotonic state in F^*

- We introduce a **logical capability**

$\text{witnessed} : \text{pred state} \rightarrow \text{Type}$

together with a **weakening** principle

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

- We introduce an operation for **witnessing** stable predicates

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

- We introduce an operation for **recalling** validity of predicates

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

Reasoning about monotonic state in F^*

- We introduce a **logical capability**

$\text{witnessed} : \text{pred state} \rightarrow \text{Type}$

together with a **weakening** principle

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

- We introduce an operation for **witnessing** stable predicates

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

- We introduce an operation for **recalling** validity of predicates

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

Reasoning about monotonic state in F^*

- We introduce a **logical capability**

$\text{witnessed} : \text{pred state} \rightarrow \text{Type}$

together with a **weakening** principle

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

- We introduce an operation for **witnessing** stable predicates

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

- We introduce an operation for **recalling** validity of predicates

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

Reasoning about monotonic state in F^*

- We introduce a **logical capability**

$\text{witnessed} : \text{pred state} \rightarrow \text{Type}$

together with a **weakening** principle

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

- We introduce an operation for **witnessing** stable predicates

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

- We introduce an operation for **recalling** validity of predicates

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

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F^*
- Examples of monotonic state at work
- A glimpse of the meta-theory

The motivating example revisited

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

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

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

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

- For any other w, wrapping

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

around the program is handled similarly easily

- Monotonic counters** are analogous, with \mathbb{N} and \leq

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


The motivating example revisited

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

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

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

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

- For any other w, wrapping

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

around the program is handled similarly easily

- Monotonic counters** are analogous, with \mathbb{N} and \leq

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

The motivating example revisited

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

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

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

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

- For any other w, wrapping

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

around the program is handled similarly easily

- Monotonic counters** are analogous, with \mathbb{N} and \leq

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

The motivating example revisited

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

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

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

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

- For any other w, wrapping

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

around the program is handled similarly easily

- Monotonic counters are analogous, with \mathbb{N} and \leq

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

The motivating example revisited

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

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

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

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

- For any other w, wrapping

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

around the program is handled similarly easily

- Monotonic counters** are analogous, with \mathbb{N} and \leq

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

References: both typed and untyped

- We define **local state** using global state + monotonicity
- We define **heaps** as maps

```
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 → t : tag → cell
```

```
type tag = Typed : tag | Untyped : live : bool → tag
```

- The **preorder** on heaps is given by

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

```
| Used a _ Typed, Used b _ Typed → a = b
```

```
| Used _ _ (Untyped l0), Used _ _ (Untyped l1) → ¬(l0) ⇒ ¬(l1)
```

```
| _, _ → ⊥
```

References: both typed and untyped

- We define **local state** using global state + monotonicity
- We define **heaps** as maps

```
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 → t : tag → cell
```

```
type tag = Typed : tag | Untyped : live : bool → tag
```

- The **preorder** on heaps is given by

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

```
| Used a _ Typed, Used b _ Typed → a = b
```

```
| Used _ _ (Untyped l0), Used _ _ (Untyped l1) → ¬(l0) ⇒ ¬(l1)
```

```
| _, _ → ⊥
```

References: both typed and untyped

- We define **local state** using global state + monotonicity
- We define **heaps** as maps

type heap =

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

where

type cell = Unused : cell | Used : a : Type → v : a → t : tag → cell

type tag = Typed : tag | Untyped : live : bool → tag

- The preorder on heaps is given by

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

| Used a _ Typed, Used b _ Typed → a = b

| Used _ _ (Untyped l₀), Used _ _ (Untyped l₁) → $\neg(l_0) \implies \neg(l_1)$

| _, _ → ⊥

References: both typed and untyped

- We define **local state** using global state + monotonicity
- We define **heaps** as maps

type heap =

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

where

type cell = Unused : cell | Used : a : Type → v : a → t : tag → cell

type tag = Typed : tag | Untyped : live : bool → tag

- The **preorder** on heaps is given by

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

| Used a - Typed, Used b - Typed → a = b

| Used - - (Untyped l₀), Used - - (Untyped l₁) → $\neg(l_0) \implies \neg(l_1)$

| -, - → ⊥

References: both typed and untyped ctd.

- We define **local state** as global state + monotonicity
- We define **heaps** as maps

type heap =

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

where

type cell = Unused : cell | Used : a:Type \rightarrow v:a \rightarrow t:tag \rightarrow cell

type tag = Typed : tag | Untyped : live:bool \rightarrow tag

- Typed references are defined as

abstract type ref t = id: $\mathbb{N}\{\text{witnessed } (\lambda h. \text{has_used_typed } \text{id } t\ h)\}$

- Untyped references are defined as

abstract type uref = id: $\mathbb{N}\{\text{witnessed } (\lambda h. \text{has_used_untyped_live } \text{id } h)\}$

References: both typed and untyped ctd.

- We define **local state** as global state + monotonicity
- We define **heaps** as maps

type heap =

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

where

type cell = Unused : cell | Used : a:Type \rightarrow v:a \rightarrow t:tag \rightarrow cell

type tag = Typed : tag | Untyped : live:bool \rightarrow tag

- **Typed references** are defined as

abstract type ref t = id: $\mathbb{N}\{\text{witnessed } (\lambda h. \text{has_used_typed id t h})\}$

- Untyped references are defined as

abstract type uref = id: $\mathbb{N}\{\text{witnessed } (\lambda h. \text{has_used_untyped_live id h})\}$

References: both typed and untyped ctd.

- We define **local state** as global state + monotonicity
- We define **heaps** as maps

`type` heap =

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

where

`type` cell = Unused : cell | Used : `a`:Type \rightarrow `v`:a \rightarrow `t`:tag \rightarrow cell

`type` tag = Typed : tag | Untyped : `live`:bool \rightarrow tag

- **Typed references** are defined as

`abstract type` ref t = id: $\mathbb{N}\{\text{witnessed } (\lambda h. \text{has_used_typed id t h})\}$

- **Untyped references** are defined as

`abstract type` uref = id: $\mathbb{N}\{\text{witnessed } (\lambda h. \text{has_used_untyped_live id h})\}$

References: **typed** and **untyped** ctd.

- The state actions for **typed references** use **witness** and **recall**
 - `let alloc t (v:t) : MST (ref t) ... = ...`
 - **get** the current heap (using global state `get`)
 - **create** a fresh ref., and **add** it to the heap
 - **put** the updated heap back (using global state `put`)
 - **witness** that the created ref. is in the heap
 - `let read t (r:ref t) : MST t ... = ...`
 - **recall** that the given ref. is in the heap
 - **get** the current heap (using global state `get`)
 - **select** the given reference from the heap
 - `let write t (r:ref t) (v:t) : MST unit ... = ...`
 - **recall** that the given ref. is in the heap
 - **get** the current heap (using global state `get`)
 - **update** the heap with the given value at the given ref.
 - **put** the updated heap back (using global state `put`)
- The actions for **untyped references** involve liveness preconditions

References: typed and untyped ctd.

- The state actions for **typed references** use **witness** and **recall**
 - `let alloc t (v:t) : MST (ref t) ... = ...`
 - **get** the current heap (using global state get)
 - **create** a fresh ref., and **add** it to the heap
 - **put** the updated heap back (using global state put)
 - **witness** that the created ref. is in the heap
 - `let read t (r:ref t) : MST t ... = ...`
 - **recall** that the given ref. is in the heap
 - **get** the current heap (using global state get)
 - **select** the given reference from the heap
 - `let write t (r:ref t) (v:t) : MST unit ... = ...`
 - **recall** that the given ref. is in the heap
 - **get** the current heap (using global state get)
 - **update** the heap with the given value at the given ref.
 - **put** the updated heap back (using global state put)
- The actions for **untyped references** involve liveness preconditions

References: typed and untyped ctd.

- The state actions for **typed references** use **witness** and **recall**
 - `let alloc t (v:t) : MST (ref t) ... = ...`
 - **get** the current heap (using global state get)
 - **create** a fresh ref., and **add** it to the heap
 - **put** the updated heap back (using global state put)
 - **witness** that the created ref. is in the heap
 - `let read t (r:ref t) : MST t ... = ...`
 - **recall** that the given ref. is in the heap
 - **get** the current heap (using global state get)
 - **select** the given reference from the heap
 - `let write t (r:ref t) (v:t) : MST unit ... = ...`
 - **recall** that the given ref. is in the heap
 - **get** the current heap (using global state get)
 - **update** the heap with the given value at the given ref.
 - **put** the updated heap back (using global state put)
- The actions for **untyped references** involve liveness preconditions

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

type tag a = Typed : **rel**:preorder a → tag a | Untyped : live:bool → tag a

- The **global preorder** is a point-wise lifting of the individual ones

```
let rel (H h0 _) (H h1 _) = ∀ id. match h0 id, h1 id with  
| Used a0 v0 (Typed rel0),  
  Used a1 v1 (Typed rel1) → a0 = a1 ∧ rel0 = rel1 ∧ rel0 v0 v1  
| ...
```

- **Monotonic references** are then given as

```
abstract type mref t rel = id:N{witnessed (λ h. has_mref id t rel h)}
```

- State actions

- The **write** action is constrained by **rel** of the given mref.
- The **witness** and **recall** actions are given reference-wise

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

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

- The **global preorder** is a point-wise lifting of the individual ones

```
let rel (H h0 _) (H h1 _) = ∀ id. match h0 id, h1 id with
| Used a0 v0 (Typed rel0),
  Used a1 v1 (Typed rel1) → a0 = a1 ∧ rel0 = rel1 ∧ rel0 v0 v1
| ...
```

- **Monotonic references** are then given as

```
abstract type mref t rel = id:N{witnessed (λ h. has_mref id t rel h)}
```

- State actions

- The **write** action is constrained by **rel** of the given mref.
- The **witness** and **recall** actions are given reference-wise

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

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

- The **global preorder** is a point-wise lifting of the individual ones

```
let rel (H h0 -) (H h1 -) = ∀ id. match h0 id, h1 id with  
| Used a0 v0 (Typed rel0),  
  Used a1 v1 (Typed rel1) → a0 = a1 ∧ rel0 = rel1 ∧ rel0 v0 v1  
| ...
```

- Monotonic references are then given as

```
abstract type mref t rel = id:N{witnessed (λ h. has_mref id t rel h)}
```

- State actions

- The **write** action is constrained by **rel** of the given mref.
- The **witness** and **recall** actions are given reference-wise

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

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

- The **global preorder** is a point-wise lifting of the individual ones

```
let rel (H h0 -) (H h1 -) = ∀ id. match h0 id, h1 id with
| Used a0 v0 (Typed rel0),
  Used a1 v1 (Typed rel1) → a0 = a1 ∧ rel0 = rel1 ∧ rel0 v0 v1
| ...
```

- **Monotonic references** are then given as

```
abstract type mref t rel = id:ℕ {witnessed (λ h. has_mref id t rel h)}
```

- State actions

- The **write** action is constrained by **rel** of the given mref.
- The **witness** and **recall** actions are given reference-wise

Monotonic references: more flexibility

- The heap now associates a **local preorder** with each reference

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

- The **global preorder** is a point-wise lifting of the individual ones

```
let rel (H h0 -) (H h1 -) = ∀ id. match h0 id, h1 id with
| Used a0 v0 (Typed rel0),
  Used a1 v1 (Typed rel1) → a0 = a1 ∧ rel0 = rel1 ∧ rel0 v0 v1
| ...
```

- **Monotonic references** are then given as

```
abstract type mref t rel = id:ℕ{witnessed (λ h. has_mref id t rel h)}
```

- State actions

- The **write** action is constrained by **rel** of the given mref.
- The **witness** and **recall** actions are given reference-wise

Outline

- Monotonic state and program verification by example
- Key ideas behind our interface for monotonic state
- Adding monotonic state to F^*
- Examples of monotonic state at work
- A glimpse of the meta-theory

A glimpse of the meta-theory

- We formalize **MST** in a small dependently typed CBV calculus

$t ::= \text{state} \mid x:t_1 \rightarrow \mathbf{Tot} \ t_2 \mid x:t_1 \rightarrow \mathbf{MST} \ t_2 \ (s.\varphi_{\text{pre}}) \ (s.y.s'.\varphi_{\text{post}}) \mid \dots$

$e ::= \text{get} \mid \text{put } v \mid \text{witness } s.\varphi \mid \text{recall } s.\varphi \mid \dots$

$\varphi ::= \text{rel } v_1 \ v_2 \mid \text{witnessed } s.\varphi \mid \dots$

- Consistency and props. of the logic via seq. calc. and cut-adm.
- Operational semantics on configurations (e, σ, W)

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

$(\text{recall } s.\varphi, \sigma, W) \rightsquigarrow (\text{return } (), \sigma, W)$

- Total correctness via progress, preservation, and SN

$$\begin{array}{l} \vdash e : \mathbf{MST} \ t \ (s.\varphi_{\text{pre}}) \ (s.x.s'.\varphi_{\text{post}}) \\ \text{witnessed } W \vdash \varphi_{\text{pre}}[\sigma/s] \end{array} \quad \Longrightarrow \quad \begin{array}{l} (e, \sigma, W) \rightsquigarrow^* (\text{return } v, \sigma', W') \quad \vdash v : t \\ W \subseteq W' \quad \text{witnessed } W' \vdash \text{rel } \sigma \ \sigma' \\ \text{witnessed } W' \vdash \varphi_{\text{post}}[\sigma/s, v/x, \sigma'/s'] \end{array}$$

A glimpse of the meta-theory

- We formalize **MST** in a small dependently typed CBV calculus

$$t ::= \text{state} \mid x:t_1 \rightarrow \mathbf{Tot} \ t_2 \mid x:t_1 \rightarrow \mathbf{MST} \ t_2 \ (s.\varphi_{\text{pre}}) \ (s.y.s'.\varphi_{\text{post}}) \mid \dots$$
$$e ::= \text{get} \mid \text{put } v \mid \text{witness } s.\varphi \mid \text{recall } s.\varphi \mid \dots$$
$$\varphi ::= \text{rel } v_1 \ v_2 \mid \text{witnessed } s.\varphi \mid \dots$$

- **Consistency** and **props. of the logic** via seq. calc. and cut-adm.

- **Operational semantics** on configurations (e, σ, W)

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

- **Total correctness** via progress, preservation, and SN

$$\begin{array}{l} \vdash e : \mathbf{MST} \ t \ (s.\varphi_{\text{pre}}) \ (s.x.s'.\varphi_{\text{post}}) \\ \text{witnessed } W \vdash \varphi_{\text{pre}}[\sigma/s] \end{array} \quad \Longrightarrow \quad \begin{array}{l} (e, \sigma, W) \rightsquigarrow^* (\text{return } v, \sigma', W') \quad \vdash v : t \\ W \subseteq W' \quad \text{witnessed } W' \vdash \text{rel } \sigma \ \sigma' \\ \text{witnessed } W' \vdash \varphi_{\text{post}}[\sigma/s, v/x, \sigma'/s'] \end{array}$$

A glimpse of the meta-theory

- We formalize **MST** in a small dependently typed CBV calculus

$$\begin{aligned} t &::= \text{state} \mid x:t_1 \rightarrow \mathbf{Tot} \ t_2 \mid x:t_1 \rightarrow \mathbf{MST} \ t_2 \ (s.\varphi_{\text{pre}}) \ (s.y.s'.\varphi_{\text{post}}) \mid \dots \\ e &::= \text{get} \mid \text{put } v \mid \text{witness } s.\varphi \mid \text{recall } s.\varphi \mid \dots \\ \varphi &::= \text{rel } v_1 \ v_2 \mid \text{witnessed } s.\varphi \mid \dots \end{aligned}$$

- **Consistency** and **props. of the logic** via seq. calc. and cut-adm.
- **Operational semantics** on configurations (e, σ, W)

$$\begin{aligned} (\text{witness } s.\varphi, \sigma, W) &\rightsquigarrow (\text{return } (), \sigma, W \cup \{s.\varphi\}) \\ (\text{recall } s.\varphi, \sigma, W) &\rightsquigarrow (\text{return } (), \sigma, W) \end{aligned}$$

- **Total correctness** via progress, preservation, and SN

$$\begin{aligned} \vdash e : \mathbf{MST} \ t \ (s.\varphi_{\text{pre}}) \ (s.x.s'.\varphi_{\text{post}}) &\quad (e, \sigma, W) \rightsquigarrow^* (\text{return } v, \sigma', W') \quad \vdash v : t \\ \text{witnessed } W \vdash \varphi_{\text{pre}}[\sigma/s] &\implies W \subseteq W' \quad \text{witnessed } W' \vdash \text{rel } \sigma \ \sigma' \\ &\quad \text{witnessed } W' \vdash \varphi_{\text{post}}[\sigma/s, v/x, \sigma'/s'] \end{aligned}$$

A glimpse of the meta-theory

- We formalize **MST** in a small dependently typed CBV calculus

$$\begin{aligned}
 t &::= \text{state} \mid x:t_1 \rightarrow \mathbf{Tot} \ t_2 \mid x:t_1 \rightarrow \mathbf{MST} \ t_2 \ (s.\varphi_{\text{pre}}) \ (s.y.s'.\varphi_{\text{post}}) \mid \dots \\
 e &::= \text{get} \mid \text{put } v \mid \text{witness } s.\varphi \mid \text{recall } s.\varphi \mid \dots \\
 \varphi &::= \text{rel } v_1 \ v_2 \mid \text{witnessed } s.\varphi \mid \dots
 \end{aligned}$$

- Consistency** and **props. of the logic** via seq. calc. and cut-adm.
- Operational semantics** on configurations (e, σ, W)

$$\begin{aligned}
 (\text{witness } s.\varphi, \sigma, W) &\rightsquigarrow (\text{return } (), \sigma, W \cup \{s.\varphi\}) \\
 (\text{recall } s.\varphi, \sigma, W) &\rightsquigarrow (\text{return } (), \sigma, W)
 \end{aligned}$$

- Total correctness** via progress, preservation, and SN

$$\begin{aligned}
 \vdash e : \mathbf{MST} \ t \ (s.\varphi_{\text{pre}}) \ (s.x.s'.\varphi_{\text{post}}) &\quad (e, \sigma, W) \rightsquigarrow^* (\text{return } v, \sigma', W') \quad \vdash v : t \\
 \text{witnessed } W \vdash \varphi_{\text{pre}}[\sigma/s] &\implies W \subseteq W' \quad \text{witnessed } W' \vdash \text{rel } \sigma \ \sigma' \\
 &\quad \text{witnessed } W' \vdash \varphi_{\text{post}}[\sigma/s, v/x, \sigma'/s']
 \end{aligned}$$

Conclusion

- In conclusion
 - making use of monotonicity is quite useful in verification
 - using monotonicity can be distilled into a simple interface
 - useful for both programming (refs.) and verification (crypto,TLS)
- Not in this talk (see the draft paper on arXiv)
 - temporarily **escaping the preorder** via snapshots
 - **revealing the representation** via selective monadic reification
- Future work
 - extending F^* with indexed effects
 - combining preorders (e.g., ala graded monads)
 - modal aspects of witnessed p
 - connections with other works, e.g., Iris and [Pilkiewicz,Pottier'11]

Conclusion

- In conclusion
 - making use of monotonicity is quite useful in verification
 - using monotonicity can be distilled into a simple interface
 - useful for both programming (refs.) and verification (crypto,TLS)
- Not in this talk (see the draft paper on arXiv)
 - temporarily **escaping the preorder** via snapshots
 - **revealing the representation** via selective monadic reification
- Future work
 - extending F^* with indexed effects
 - combining preorders (e.g., ala graded monads)
 - modal aspects of witnessed p
 - connections with other works, e.g., Iris and [Pilkiewicz,Pottier'11]

Conclusion

- In conclusion
 - making use of monotonicity is quite useful in verification
 - using monotonicity can be distilled into a simple interface
 - useful for both programming (refs.) and verification (crypto,TLS)
- Not in this talk (see the draft paper on arXiv)
 - temporarily **escaping the preorder** via snapshots
 - **revealing the representation** via selective monadic reification
- Future work
 - extending F^* with indexed effects
 - combining preorders (e.g., ala graded monads)
 - modal aspects of witnessed p
 - connections with other works, e.g., Iris and [Pilkiewicz,Pottier'11]

Thank you!

Questions?

Recalling a Witness:
Foundations and Applications of Monotonic State
(arXiv:1707.02466)