Recall for free:

preorder-respecting state monads in F*

Danel Ahman

LFCS, University of Edinburgh

(joint work with Aseem Rastogi and Nikhil Swamy at MSR)

PLInG Meeting
13 October 2016

- An effectful dependently-typed functional language

```
a,b ::= ... | x:a → PURE b wp_p
            | x:a → DIV b wp_d
            | x:a → STATE b wp_s
            | x:a → ST b pre post
```

- An effectful dependently-typed functional language

```
a,b ::= ... | x:a → PURE b wp_p
            | x:a → DIV b wp_d
            | x:a → STATE b wp_s
            | x:a → ST b pre post
```

$\boxed{\textbf{PURE , DIV , STATE - Dijkstra monads}}$

- An effectful dependently-typed functional language

```
a,b  weakest precondition predicate transformers
          | x:a → DIV b wpd
          | x:a → STATE b wps
          | x:a → ST b pre post
```

PURE , DIV , STATE - Dijkstra monads

- An effectful dependently-typed functional language

`a,b` **weakest precondition predicate transformers**

```
| x:a → DIV b wpd
| x:a → STATE b wps
| x:a → ST b pre post
```

PURE , DIV , STATE - Dijkstra monads

- Some resources:
  - www.fstar-lang.org
  - "Dependent Types and Multi-Monadic Effects in F*" [POPL'16]
  - "Dijkstra Monads for Free" [POPL'17]

Outline

- A recurring phenomenon
- Preorder-respecting (Dijkstra) state monads in F*
- Some examples
- A glimpse of the formal metatheory
- What are Dijkstra monads fibrationally?

(if time permits)

A recurring phenomenon

Example 1

Example 1

```
let s   = get () in
let _   = put (s + 1) in
let s'  = get () in
f () ;
let s'' = get () in
g ()
```

Example 1

```
let s   = get () in
let _   = put (s + 1) in
let s'  = get () in
f () ;
let s'' = get () in
g ()
```

```
assert (s' > 0) ;
```

Example 1

```
let s   = get () in
let _   = put (s + 1) in
let s'  = get () in
f () ;
let s'' = get () in
g ()
```

assert (s' > 0) ;

f only increases the state (counter)

Example 1

```
let s   = get () in
let _   = put (s + 1) in
let s'  = get () in
f () ;
let s'' = get () in
g ()
```

assert (s' > 0) ;

f only increases the state (counter)

assert (s'' > 0) ;

Example 1

```
let s   = get () in
let _   = put (s + 1) in
let s'  = get () in
f () ;
let s'' = get () in
g ()
```

assert (s' > 0) ;

f only increases the state (counter)

;

- How to prove the 2nd assert "for free"?
- How to avoid global spec. in the type of f about s' ≤ s''?
- Generalise to other preorders and stable predicates?

Example 2

Example 2

```
val f : ref int → STATE unit (fun p s → True)

let f r =
  let r' = alloc 0 in
  g r r'
```

Example 2

```
val f : ref int → STATE unit (fun p s → True)

let f r =
  let r' = alloc 0 in
  g r'
```

```
assert (r <> r') ;
```

Example 2

```
val f : ref int → STATE unit (fun p s → True)

let f r =
  let r' = alloc 0 in
  g r r'
```

FStar.ST.recall r ;

assert (r <> r') ;

Example 2

```
val f : ref int → STATE unit (fun p s → True)

let f r =
  let r' = alloc 0 in
  g r r'
```

`FStar.ST.recall r ;`

`<> r') ;`

- `FStar.ST.recall` is used pervasively in practice
- Can't implement it - has to be taken as an axiom
- It is intuitively correct - there is no dealloc in F*
- How to make this intuition formal?

Example 3

Example 3

Monotonic references in `FStar.Monotonic.RRef`

```
type m_ref (a:Type) (rel:preorder a)
```

Example 3

Monotonic references in `FStar.Monotonic.RRef`

```
type m_ref (a:Type) (rel:preorder a)
```

Provides operations

- `recall` - works as in `FStar.ST.recall`
- `witness` - witness a predicate holding value of a ref.
- `testify` - a previously witnessed predicate holds for a ref.

Example 3

Monotonic references in `FStar.Monotonic.RRef`

```
type m_ref (a:Type) (rel:preorder a)
```

Provides operations

- `recall` - works as in `FStar.ST.recall`
- `witness` - witness a predicate holding value of a ref.
- `testify` - a previously witnessed predicate holds for a ref.

also has to be taken as an axiom

Example 3

Monotonic references in `FStar.Monotonic.RRef`

```
type m_ref (a:Type) (rel:preorder a)
```

Provides operations

- `recall` - works as in `FStar.ST.recall`
- `witness` - witness a predicate holding value of a ref.
- `testify` - a previously witnessed predicate holds for a ref.

also has to be taken as an axiom

Used pervasively in `mitls-fstar`

- for monotone sequences, -counters and -logs

State monads in

State monads in

State monads in

The state monad in F* has (roughly) the following type

```
STATE :  a:Type
      → wp:((a → state → Type₀) → state → Type₀)
      → Effect
```

State monads in F*

The state monad in F* has (roughly) the following type

```
STATE :  a:Type
      → wp:((a → state → Type0) → state → Type0)
      → Effect
```

WPs of state operations are familiar from Hoare Logic, e.g.

```
val put : x:state
        → STATE unit (fun p s → p () x)
```

State monads in F*

The state monad in F* has (roughly) the following type

$$
\begin{aligned}
&\textbf{STATE} : \texttt{a:Type} \\
&\qquad \to \texttt{wp:((a} \to \texttt{state} \to \texttt{Type}_0\texttt{)} \to \texttt{state} \to \texttt{Type}_0\texttt{)} \\
&\qquad \to \texttt{Effect}
\end{aligned}
$$

Usually a more human-readable syntactic sugar is used

$$
\begin{aligned}
&\textbf{ST} : \texttt{a:Type} \\
&\quad \to \texttt{pre:(state} \to \texttt{Type}_0\texttt{)} \\
&\quad \to \texttt{post:(state} \to \texttt{(a} \to \texttt{state} \to \texttt{Type}_0\texttt{))} \\
&\quad \to \texttt{Effect}
\end{aligned}
$$

Preorder-respecting state monads in

High-level picture

High-level picture

Idea is based on axioms of `FStar.ST.recall` and `mref`

High-level picture

Idea is based on axioms of `FStar.ST.recall` and `mref` and aims to be a replacement for them in long-term

High-level picture

Idea is based on axioms of `FStar.ST.recall` and `mref` and aims to be a replacement for them in long-term

At high-level, we:

High-level picture

Idea is based on axioms of `FStar.ST.recall` and `mref` and aims to be a replacement for them in long-term

At high-level, we:

- index F* state monads by preorders on states

High-level picture

Idea is based on axioms of `FStar.ST.recall` and `mref` and aims to be a replacement for them in long-term

At high-level, we:

- index F* state monads by preorders on states
- ensure that writes respect them (think update monads)

High-level picture

Idea is based on axioms of `FStar.ST.recall` and `mref` and aims to be a replacement for them in long-term

At high-level, we:

- index F* state monads by preorders on states
- ensure that writes respect them (think update monads)
- add an operation for witnessing stable predicates

High-level picture

Idea is based on axioms of `FStar.ST.recall` and `mref` and aims to be a replacement for them in long-term

At high-level, we:

- index F* state monads by preorders on states
- ensure that writes respect them (think update monads)
- add an operation for witnessing stable predicates
- add an operation for recalling stable predicates

High-level picture

Idea is based on axioms of `FStar.ST.recall` and `mref` and aims to be a replacement for them in long-term

At high-level, we:

- index F* state monads by preorders on states
- ensure that writes respect them (think update monads)
- add an operation for witnessing stable predicates
- add an operation for recalling stable predicates
- introduce a ■-modality on stable predicates

High-level picture

Idea is based on axioms of `FStar.ST.recall` and `mref` and aims to be a replacement for them in long-term

At high-level, we:

- index F* state monads by preorders on states
- ensure that writes respect them (think update monads)
- add an operation for witnessing stable predicates
- add an operation for recalling stable predicates
- introduce a ■-modality on stable predicates

"witnessed"

Relations and predicates

Relations and predicates

Relations and preorders

```
let relation a = a → a → Type₀

let preorder a = rel:relation a
  { (forall x     . rel x x) ∧
    (forall x y z . rel x y ∧ rel y z ⇒ rel x z) }
```

Relations and predicates

Relations and preorders

```
let relation a = a → a → Type₀

let preorder a = rel:relation a
  { (forall x     . rel x x) ∧
    (forall x y z . rel x y ∧ rel y z ⇒ rel x z) }
```

Predicates and stability

```
let predicate a      = a → Type₀

let stable_predicate #a rel = p:predicate a
  { forall x y . p x ∧ rel x y ⇒ p y }
```

PSTATE and PST

PSTATE and **PST**

The signature of preorder-respecting state monads

```
PSTATE : rel:preorder state
       → a:Type
       → wp:((a → state → Type0) → state → Type0)
       → Effect
```

PSTATE and **PST**

The signature of preorder-respecting state monads

```
PSTATE : rel:preorder state
       → a:Type
       → wp:((a → state → Type₀) → state → Type₀)
       → Effect
```

We added **PSTATE** into the effect hierarchy of F* via **STATE**

PSTATE and **PST**

The signature of preorder-respecting state monads

```
PSTATE : rel:preorder state
       → a:Type
       → wp:((a → state → Type0) → state → Type0)
       → Effect
```

We added **PSTATE** into the effect hierarchy of F* via **STATE**

Note: Unfortunately, at the moment we can't define

```
sub_effect (forall state rel . Pure ⇝ PSTATE rel)
```

But we can make sub-effecting work for instances of **PSTATE**!

PSTATE and **PST**

The signature of preorder-respecting state monads

```
PSTATE : rel:preorder state
       → a:Type
       → wp:((a → state → Type₀) → state → Type₀)
       → Effect
```

Analogously to **STATE**, we again use syntactic sugar

```
PST : rel:preorder state
    → a:Type
    → pre:(state → Type₀)
    → post:(state → a → state → Type₀)
    → Effect
```

Operations

`get` and `put`

get and put

```
val get : #rel:preorder state
        → PST rel state (fun _ → True)
                        (fun s₀ s s₁ → s₀ = s ∧ s = s₁)
```

get and put

pre and post are exactly as for **STATE** and **ST**

```
val get : #rel:preorder state
       → PST rel state (fun _ → True)
                        (fun s₀ s s₁ → s₀ = s ∧ s = s₁)
```

get and put

pre and post are exactly as for **STATE** and **ST**

```
val get : #rel:preorder state
        → PST rel state (fun _ → True)
                        (fun s₀ s s₁ → s₀ = s ∧ s = s₁)


val put : #rel:preorder state
        → x:state
        → PST rel unit (fun s₀ → rel s₀ x)
                       (fun _ _ s₁ → s₁ = x)
```

get and put

pre and post are exactly as for **STATE** and **ST**

```
val get : #rel:preorder state
        → PST rel state (fun _ → True)
                        (fun s₀ s s₁ → s₀ = s ∧ s = s₁)
```

the change wrt. **STATE** and **ST**

```
val put : #rel:preorder state
        → x:state
        → PST rel unit (fun s₀ → rel s₀ x)
                       (fun _ _ s₁ → s₁ = x)
```

■-modality in

■-modality in

We introduce an uninterpreted function symbol

```
val ■ : #rel:preorder state
      → p:stable_predicate rel
      → Type₀
```

■-modality in

We introduce an uninterpreted function symbol

```
val ■ : #rel:preorder state
      → p:stable_predicate rel
      → Type₀
```

We assume logical axioms, e.g., functoriality:

```
forall p p' . (forall s . p s ⇒ p' s) ⇒ (■p ⇒ ■p')
```

■-modality in

We introduce an uninterpreted function symbol

```
val ■ : #rel:preorder state
      → p:stable_predicate rel
      → Type₀
```

We assume logical axioms, e.g., functoriality:

```
forall p p' . (forall s . p s ⇒ p' s) ⇒ (■p ⇒ ■p')
```

Two readings of `■p` :

p held at some past state of an **PSTATE** computation

p holds at all states reachable from the current with **PSTATE**

witness and recall

witness and recall

```
val witness : #rel:preorder state
            → p:stable_predicate rel
            → PST rel unit (fun s₀ → p s₀)
                           (fun s₀ _ s₁ → s₀ = s₁ ∧ ■p)
```

witness and recall

```
val witness : #rel:preorder state
            → p:stable_predicate rel
            → PST rel unit (fun s₀ → p s₀)
                           (fun s₀ _ s₁ → s₀ = s₁ ∧ ■p)

val recall : #rel:preorder state
           → p:stable_predicate rel
           → PST rel unit (fun _ → ■p)
                          (fun s₀ _ s₁ → s₀ = s₁ ∧ p s₁)
```

Examples

Examples

Examples

- Recalling that allocated references remain allocated
  - using `FStar.Heap.heap`

    (need a source of freshness for `alloc`)
  - ⋆ using our own heap type

    (source of freshness built into the heap)

Examples

- Recalling that allocated references remain allocated
  - using `FStar.Heap.heap`
    (need a source of freshness for `alloc`)
  - ⋆ using our own heap type
    (source of freshness built into the heap)
- Immutable references and other preorders

Examples

- Recalling that allocated references remain allocated
  - using `FStar.Heap.heap`
    (need a source of freshness for `alloc`)
  - ★ using our own heap type
    (source of freshness built into the heap)
- Immutable references and other preorders
- Monotonic references

Examples

- Recalling that allocated references remain allocated
    - using `FStar.Heap.heap`
      (need a source of freshness for `alloc`)
    - ⋆ using our own heap type
      (source of freshness built into the heap)
- Immutable references and other preorders
- Monotonic references
- ⋆ Temporarily ignoring the constraint on `put` via snapshots

Our `heap` and `ref` types

Our heap and ref types

The heap and ref types

```
let heap  = h:(nat * (nat → option (a:Type0 & a)))
               { ... }

let ref a = nat
```

Our heap and ref types

freshness counter

The heap and ref types

```
let heap  = h:(nat * (nat → option (a:Type₀ & a)))
              { ... }

let ref a = nat
```

Our heap and `ref` types

freshness counter

The heap and `ref` types

```
let heap  = h:(nat * (nat → option (a:Type0 & a)))
              { ... }

let ref a = nat
```

We can define `sel` and `upd` and `gen_fresh` operations

Our heap and ref types

The heap and ref types

freshness counter

```
let heap  = h:(nat * (nat → option (a:Type₀ & a)))
                { ... }

let ref a = nat
```

both ops. have (r ∈ h)
refinements on references

We can define sel and upd and gen_fresh operations

Our heap and `ref` types

freshness counter

The `heap` and `ref` types

```
let heap  = h:(nat * (nat → option (a:Type0 & a)))
                { ... }

let ref a = nat
```

both ops. have `(r ∈ h)`
refinements on references

We can define `sel` and `upd` and `gen_fresh` operations

and prove expected properties, e.g.:

```
r <> r'  ⇒  sel (upd h r x) r' = sel h r'
```

Our heap and ref types

freshness counter

The heap and ref types

```
let heap  = h:(nat * (nat → option (a:Type0 & a)))
              { ... }

let ref a = nat
```

both ops. have (r ∈ h)
refinements on references

Goal: use this heap as drop-in replacement for F*'s heap

(but in F*'s heap, sel and upd don't have (r ∈ h) refinements)

- change the type of refs. to (`let ref a = nat * a`)
- make use of the presence LEM in WPs for checking (r ∈ h)

Allocated references example

Allocated references example

The type of refs. and the preorder for **AllocST**

```
let ref a     = r:(Heap.ref a){ ■(fun h → r ∈ h) }

let rel h0 h1 = forall a r . r ∈ h0 ⇒ r ∈ h1

AllocST a pre post = PST rel a pre post
```

Allocated references example

The type of refs. and the preorder for **AllocST**

```
let ref a     = r:(Heap.ref a){ ■(fun h → r ∈ h) }

let rel h0 h1 = forall a r . r ∈ h0 ⇒ r ∈ h1

AllocST a pre post = PST rel a pre post
```

AllocST operations crucially use `witness` and `recall`, e.g.,

```
let read #a (r:ref a) =
  let h = get () in
  recall (fun h → r ∈ h) ;
  sel h r
```

Snapshots

Snapshots

We first define snaphsot-capable state as

```
let s_state state = state * option state
```

Snapshots

We first define snaphsot-capable state as

```
let s_state state = state * option state
```

The snaphsot-capable preorder is indexed by `rel` on `state`

```
let s_rel (rel:preorder state) s₀ s₁ =
  match (snd s₀) (snd s₁) with
```

Snapshots

We first define snaphsot-capable state as

```
let s_state state = state * option state
```

The snaphsot-capable preorder is indexed by `rel` on `state`

```
let s_rel (rel:preorder state) s₀ s₁ =
  match (snd s₀) (snd s₁) with
  | None        None        ⇒ rel (fst s₀) (fst s₁)
```

Snapshots

We first define snaphsot-capable state as

```
let s_state state = state * option state
```

The snaphsot-capable preorder is indexed by `rel` on `state`

```
let s_rel (rel:preorder state) s₀ s₁ =
  match (snd s₀) (snd s₁) with
  | None        None        ⇒ rel (fst s₀) (fst s₁)
  | None        (Some s)    ⇒ rel (fst s₀) s
```

Snapshots

We first define snaphsot-capable state as

```
let s_state state = state * option state
```

The snaphsot-capable preorder is indexed by `rel` on `state`

```
let s_rel (rel:preorder state) s₀ s₁ =
  match (snd s₀) (snd s₁) with
   | None        None        ⇒ rel (fst s₀) (fst s₁)
   | None        (Some s)    ⇒ rel (fst s₀) s
   | (Some s)    None        ⇒ rel s (fst s₁)
```

Snapshots

We first define snaphsot-capable state as

```
let s_state state = state * option state
```

The snaphsot-capable preorder is indexed by `rel` on `state`

```
let s_rel (rel:preorder state) s₀ s₁ =
  match (snd s₀) (snd s₁) with
  | None        None        ⇒ rel (fst s₀) (fst s₁)
  | None        (Some s)    ⇒ rel (fst s₀) s
  | (Some s)    None        ⇒ rel s (fst s₁)
  | (Some s₀') (Some s₁') ⇒ rel s₀' s₁'
```

`read` and `write`

read and write

```
val read : #rel:preorder state
         → SST rel state
              (fun s₀ → True)
              (fun s₀ s s₁ → fst s₀ = s ∧ s = fst s₁ ∧
                             snd s₀ = snd s₁)
let read #rel x = ...
```

read and write

```
val read : #rel:preorder state
         → SST rel state
               (fun s0 → True)
               (fun s0 s s1 → fst s0 = s ∧ s = fst s1 ∧
                             snd s0 = snd s1)
let read #rel x = ...

val write : #rel:preorder state
          → x:state
          → SST rel unit
                (fun s0 → s_rel rel s0 (x,snd s0))
                (fun s0 _ s1 → s1 = (x,snd s0))
let write #rel x = ...
```

witness and recall

witness and recall

```
val witness : #rel:preorder state
            → p:stable_predicate rel
            → SST rel unit (fun s0 → p (fst s0) ∧
                                     snd s0 = None)
                           (fun s0 _ s1 → s0 = s1 ∧ ■p)
let witness #rel p = ...
```

witness and recall

```
val witness : #rel:preorder state
            → p:stable_predicate rel
            → SST rel unit (fun s0 → p (fst s0) ∧
                                     snd s0 = None)
                           (fun s0 _ s1 → s0 = s1 ∧ ■p)
let witness #rel p = ...

val recall : #rel:preorder state
           → p:stable_predicate rel
           → SST rel unit (fun s0 → ■p ∧ snd s0 = None)
                          (fun s0 _ s1 → s0 = s1 ∧
                                         p (fst s1))
let recall #rel p = ...
```

snap and ok

snap and ok

```
val snap : #rel:preorder state
         → SST rel unit
              (fun s₀ → snd s₀ = None)
              (fun s₀ _ s₁ → fst s₀ = fst s₁ ∧
                             snd s₁ = Some (fst s₀))
let snap #rel = ...
```

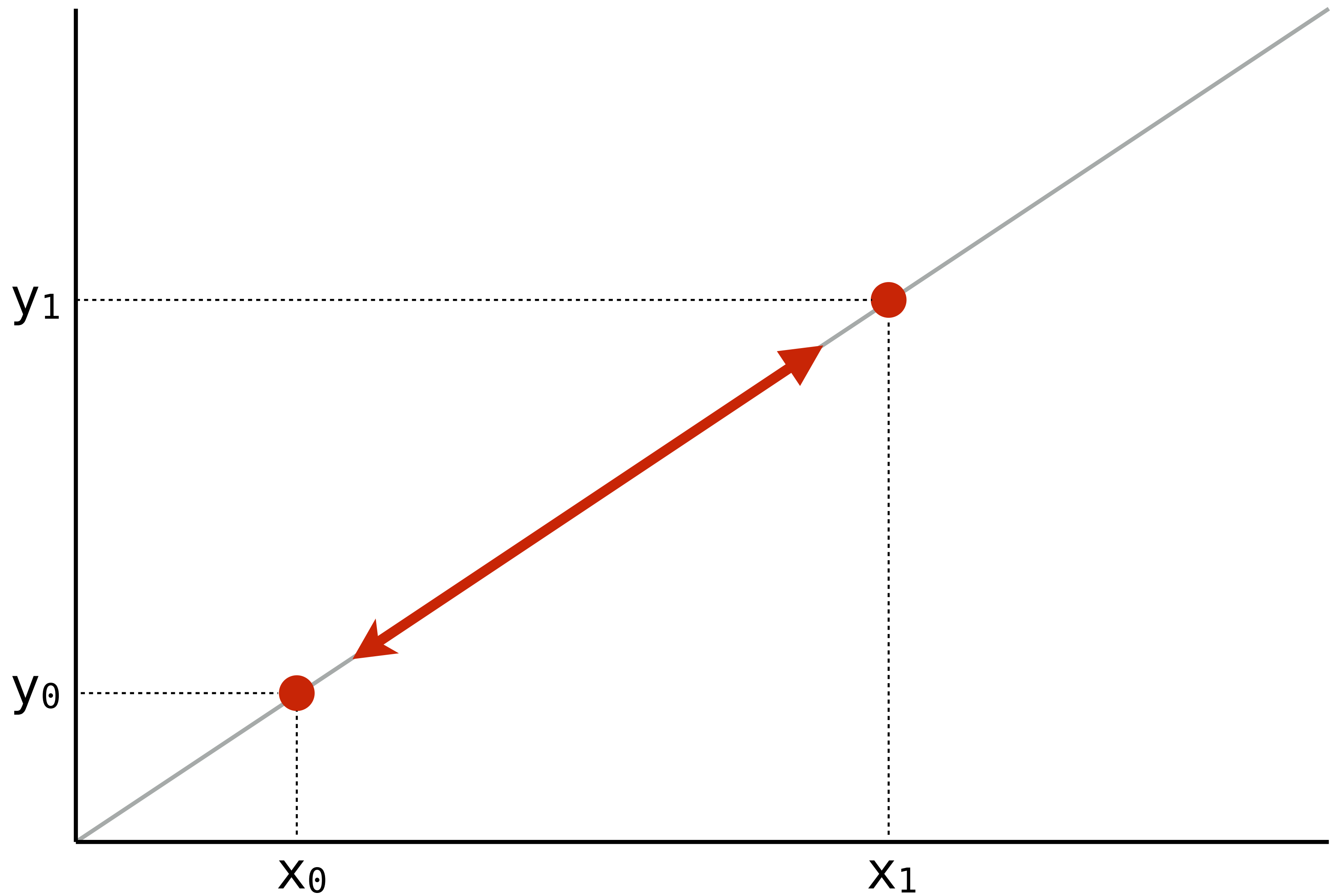
snap and ok

```
val snap : #rel:preorder state
         → SST rel unit
              (fun s₀ → snd s₀ = None)
              (fun s₀ _ s₁ → fst s₀ = fst s₁ ∧
                             snd s₁ = Some (fst s₀))
let snap #rel = ...

val ok : #rel:preorder state
       → SST rel unit
            (fun s₀ → exists s . snd s₀ = Some s ∧
                                 rel s (fst s₀))
            (fun s₀ _ s₁ → fst s₀ = fst s₁ ∧
                           snd s₁ = None)
let ok #rel = ...
```

Example use of SST

Example use of SST



- Implementing a 2D point using two memory locations
- E.g., want to enforce that ● can only move along some line

A glimpse of the formal metatheory

PSTATE formally

PSTATE formally

We work with a small calculus based on EMF* from DM4F

```
t, wp, ::= state | rel | x:t1 → Tot t2 | x:t1 → PSTATE t2 wp | ...
e, φ     | x | fun x:t → e | e1 e2 | (e1,e2) | fst e | ...
         | return e | bind e1 x:t.e2
         | get e | put e | witness e | recall e
```

PSTATE formally

We work with a small calculus based on EMF* from DM4F

```
t, wp, ::= state | rel | x:t1 → Tot t2 | x:t1 → PSTATE t2 wp | ...
e, φ       | x | fun x:t → e | e1 e2 | (e1,e2) | fst e | ...
           | return e | bind e1 x:t.e2
           | get e | put e | witness e | recall e
```

Typing judgements have the form

```
G ⊢ e : Tot t
G ⊢ e : PSTATE t wp
```

PSTATE formally

We work with a small calculus based on EMF* from DM4F

```
t, wp, ::= state | rel | x:t1 → Tot t2 | x:t1 → PSTATE t2 wp | ...
e, φ       | x | fun x:t → e | e1 e2 | (e1,e2) | fst e | ...
           | return e | bind e1 x:t.e2
           | get e | put e | witness e | recall e
```

Typing judgements have the form

```
G ⊢ e : Tot t
G ⊢ e : PSTATE t wp
```

There is also a judgement for logical reasoning in WPs

```
G | Φ ⊨ φ
```

PSTATE formally

We work with a small calculus based on EMF* from DM4F

```
t, wp, ::= state | rel | x:t1 → Tot t2 | x:t1 → PSTATE t2 wp | ...
e, φ       | x | fun x:t → e | e1 e2 | (e1,e2) | fst e | ...
           | return e | bind e1 x:t.e2
           | get e | put e | witness e | recall e
```

Typing judgements have the form

```
G ⊢ e : Tot t
G ⊢ e : PSTATE t wp
```

There is also a judgement for logical reasoning in WPs

```
G | Φ ⊨ φ
```

nat. deduction for classical predicate logic

Operational semantics

Operational semantics

Small-step call-by-value reduction relation

$$(\Phi,s,e) \longrightarrow (\Phi',s',e')$$

where

- Φ is a finite set of (witnessed) stable predicates
- s is a value of type `state`
- e is an expression

Operational semantics

Small-step call-by-value reduction relation

```
(Φ,s,e) ⟶ (Φ',s',e')
```

where

- Φ is a finite set of (witnessed) stable predicates
- s is a value of type `state`
- e is an expression

Examples of reduction rules

```
(Φ,s,put v)       ⟶ (Φ,v,return ())

(Φ,s,witness v) ⟶ (Φ ∪ {v},s,return ())
```

Progress thm. for **`PSTATE`**

Progress thm. for **PSTATE**

$\forall$ `f t wp .`

$\vdash$ `f :` **`PSTATE`** `t wp`

$\Rightarrow$

1. $\exists$ `v . f = return v`

$\vee$

2. $\forall$ `Φ s .` $\exists$ `Φ' s' f' . (Φ,s,f)` $\longrightarrow$ `(Φ',s',f')`

Preservation thm. for **PSTATE**

Preservation thm. for **PSTATE**

$$
\begin{array}{l}
\forall\ \mathtt{f\ t\ wp}\ \Phi\ \mathtt{s}\ \Phi'\ \mathtt{s'\ f'}. \\
\quad \vdash \mathtt{f} : \mathbf{PSTATE}\ \mathtt{t\ wp} \ \wedge\ (\Phi,\mathtt{s})\ \mathtt{wf}\ \wedge \\
\quad (\Phi,\mathtt{s},\mathtt{f}) \longrightarrow (\Phi',\mathtt{s'},\mathtt{f'}) \\
\quad \Rightarrow \\
\quad \forall\ \mathtt{post}\ .\ \blacksquare\Phi \vDash \mathtt{wp\ post\ s} \\
\qquad\qquad \Rightarrow \\
\qquad\qquad \Phi \subseteq \Phi'\ \wedge\ (\Phi',\mathtt{s'})\ \mathtt{wf}\ \wedge \\
\qquad\qquad \blacksquare\Phi \vDash \mathtt{rel\ s\ s'}\ \wedge \\
\qquad\qquad \exists\ \mathtt{wp'}\ .\ \vdash \mathtt{f'} : \mathbf{PSTATE}\ \mathtt{t\ wp'}\ \wedge \\
\qquad\qquad\qquad \blacksquare\Phi' \vDash \mathtt{wp'\ post\ s'}
\end{array}
$$

Preservation thm. for **PSTATE**

$\forall$ f t wp Φ s Φ' s' f'.

⊢ f : **PSTATE** t wp ∧ (Φ,s) wf ∧

(Φ,s,f) ⟶ (Φ',s',f')

⇒

■Φ = ■(**fun** x → φ_1 x ∧ … ∧ φ_n x)

$\forall$ post . ■Φ ⊨ wp post s

⇒

Φ ⊆ Φ' ∧ (Φ',s') wf ∧

■Φ ⊨ rel s s' ∧

∃ wp' . ⊢ f' : **PSTATE** t wp' ∧

■Φ' ⊨ wp' post s'

The proof requires an inversion property (in empty context)

$$\frac{\models \ \blacksquare\varphi \ \Rightarrow \ \blacksquare\psi}{\models \ \texttt{forall}\ x\ .\ \varphi\ x \ \Rightarrow \ \psi\ x}\ (\blacksquare\text{- inv})$$

We justify ($\blacksquare$- inv) via a cut-elimination in sequent calculus

- where we have a single derivation rule for $\blacksquare$

$$\frac{G \vdash \Phi_1 \qquad G \vdash \Phi_2 \qquad G, x \mid \Phi_1, \varphi_1\ x, \ldots, \varphi_n\ x \ \vdash \ \psi_1\ x, \ldots, \psi_m\ x, \Phi_2}{G \mid \Phi_1, \blacksquare\varphi_1, \ldots, \blacksquare\varphi_n \ \vdash \ \blacksquare\psi_1, \ldots, \blacksquare\psi_m, \Phi_2}$$

Future work: model theory of $\blacksquare$

Conclusion

Conclusion

In this talk we covered:

- preorder-respecting state monads in F*
- their formal metatheory
- some of the examples of these monads

Conclusion

In this talk we covered:

- preorder-respecting state monads in F*
- their formal metatheory
- some of the examples of these monads

Ongoing and future work:

- change F*'s libraries to use **`PSTATE`**
- **`PSTATE`** in DM4F setting? (how to reify it safely?)
- model theory of $\blacksquare$
- categorical semantics of Dijkstra monads (rel. monads.)

Dijkstra monad T in CT?

Dijkstra monad T in CT?

Type formation rule for a Dijkstra monad

$$\frac{\Gamma \vdash t : \mathsf{Type} \qquad \Gamma \vdash wp : WP\ A}{\Gamma \vdash T\ t\ wp : \mathsf{Type}}$$

The unit of a Dijkstra monad

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{return}\ e : T\ t\ (WP.\mathtt{return}\ e)}$$

The Kleisli extension of a Dijkstra monad

$$\frac{\Gamma \vdash M : T\ t_1\ wp_1 \qquad \Gamma \vdash t_2 \qquad \Gamma, x{:}t_1 \vdash N : T\ t_2\ wp_2}{\Gamma \vdash \mathtt{bind}\ e_1\ x.e_2 : T\ t_2\ (WP.\mathtt{bind}\ wp_1\ x.wp_2)}$$

Dijkstra monad T in CT?

We'll work in the setting of closed comprehension cats., i.e.,

- $\mathcal{B}$ models contexts
- $\mathcal{V}$ models types in context
- terms in context Γ are modeled as global elements in $\mathcal{V}_{[\![\Gamma]\!]}$

$$
\begin{tikzcd}
\mathcal{B}^{\to} \arrow[ddr, bend right, "\mathrm{cod}"'] & & \mathcal{V} \arrow[ll, "\mathcal{P}"'] \arrow[dd, bend right=60, "p"'] \arrow[dd, bend left=60, "\{-\}"] \\
& & \\
& & \mathcal{B} \arrow[uu, "{\dashv \quad 1 \qquad \dashv}" description]
\end{tikzcd}
$$

- $\mathcal{P}$ is fully faithful

Dijkstra monad T in CT?

For modeling Dijkstra monads, we assume:

- a split fibred monad $WP : p \to p$
- a functor $T : \mathcal{V} \to \mathcal{V}$

s.t. $p \circ T = \{ - \} \circ WP$

T preserves Cartesian morphisms on-the-nose

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{return}\, e : T\ t\ (WP.\mathtt{return}\ e)}$$

$$\begin{tikzcd}
\mathcal{B}^{\to} \arrow[dr, bend right=40, "\mathrm{cod}"'] & & \mathcal{V} \arrow[ll, "\mathcal{P}"'] \arrow[d, bend right=70, "p"'] \arrow[d, bend left=70, "\{-\}"] \\
& & \mathcal{B} \arrow[u, "1"'] \arrow[u, phantom, shift right=6, "\dashv"] \arrow[u, phantom, shift left=6, "\dashv"]
\end{tikzcd}$$

Can we model the unit and Kleisli ext. for T in known terms?

Dijkstra monad T in CT?

For modeling Dijkstra monads, we assume:

- a split fibred monad $\quad WP : p \to p$
- a functor $\;T : \mathcal{V} \to \mathcal{V}$

  s.t. $\quad p \circ T = \{-\} \circ WP$

  T preserves Cartesian morphisms on-the-nose

dependency on WP

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{return}\; e : T\; t\; (WP.\mathtt{return}\; e)}$$

$$\begin{tikzcd}
\mathcal{B}^{\to} \arrow[dr, bend right, "\mathrm{cod}"'] & & \mathcal{V} \arrow[ll, "\mathcal{P}"'] \arrow[d, bend right=60, "p"'] \arrow[d, bend left=60, "\{-\}"] \\
& & \mathcal{B} \arrow[u, "1"]
\end{tikzcd}$$

Can we model the unit and Kleisli ext. for T in known terms?

Dijkstra monad T in CT?

For modeling Dijkstra monads, we assume:

- a split fibred monad $WP : p \to p$
- a functor $T : \mathcal{V} \to \mathcal{V}$

dependency on WP

s.t. $p \circ T = \{-\} \circ WP$

T preserves Cartesian morphisms on-the-nose

closed under substitution

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathtt{return}\ e : T\ t\ (WP.\mathtt{return}\ e)}$$

$$\begin{tikzcd}
\mathcal{B}^{\to} \arrow[dr, bend right, "\text{cod}"'] & & \mathcal{V} \arrow[ll, "\mathcal{P}"'] \arrow[d, bend right=60, "p"'] \arrow[d, bend left=60, "\{-\}"] \\
& & \mathcal{B} \arrow[u, "1"]
\end{tikzcd}$$

Can we model the unit and Kleisli ext. for T in known terms?

Dijkstra monad T in $\mathcal{B}^{\to}$

Dijkstra monad T in $\mathcal{B}^{\rightarrow}$

The unit of a Dijkstra monad

$$\eta_A \quad : \quad \begin{tikzcd} \{A\} \arrow[r] \arrow[d, "\mathsf{id}_{\{A\}}"'] & \{T(A)\} \arrow[d, "\pi_{T(A)}"] \\ \{A\} \arrow[r, "\{WP.\eta_A\}"'] & \{WP(A)\} \end{tikzcd}$$

Dijkstra monad T in $\mathcal{B}^{\rightarrow}$

The unit of a Dijkstra monad

$$\eta_A \quad : \quad \begin{tikzcd} \{A\} \arrow[r] \arrow[d, "\mathsf{id}_{\{A\}}"'] & \{T(A)\} \arrow[d, "\pi_{T(A)}"] \\ \{A\} \arrow[r, "\{WP.\eta_A\}"'] & \{WP(A)\} \end{tikzcd}$$

The Kleisli extension of a Dijkstra monad

$$\left(\begin{tikzcd} \{A\} \arrow[r, "f"] \arrow[d, "\mathsf{id}_{\{A\}}"'] & \{T(B)\} \arrow[d, "\pi_{T(B)}"] \\ \{A\} \arrow[r, "\{g\}"'] & \{WP(B)\} \end{tikzcd} \right)^* \quad : \quad \begin{tikzcd} \{T(A)\} \arrow[r] \arrow[d, "\pi_{T(A)}"'] & \{T(B)\} \arrow[d, "\pi_{T(B)}"] \\ \{WP(A)\} \arrow[r, "\{WP.(-)^*(g)\}"'] & \{WP(B)\} \end{tikzcd}$$

Dijkstra monad T in $\mathcal{B}^{\to}$

The unit of a Dijkstra monad

$$\{A\} \longrightarrow \{T(A)\}$$

This data and the associated laws are precisely those for a relative monad

$$\widehat{T} : \mathcal{V} \longrightarrow \overline{\mathsf{im}}(\{-\}) \downarrow \{-\}$$

$$\widehat{T}(A) \stackrel{\text{def}}{=} \{T(A)\} \xrightarrow{\pi_{T(A)}} \{WP(A)\}$$

on

$$J : \mathcal{V} \longrightarrow \overline{\mathsf{im}}(\{-\}) \downarrow \{-\}$$

$$J(A) \stackrel{\text{def}}{=} \{A\} \xrightarrow{\mathsf{id}_{\{A\}}} \{A\}$$

onad

$$\begin{tikzcd}
\{T(A)\} \arrow[r] \arrow[d, "\pi_{T(A)}"'] & \{T(B)\} \arrow[d, "\pi_{T(B)}"] \\
\{WP(A)\} \arrow[r, "\{WP.(-)^*(g)\}"'] & \{WP(B)\}
\end{tikzcd}$$