TYPES 2022

Partial Dijkstra monads for all

Théo **Winterhalter** Cezar-Constantin **Andrici** Cătălin **Hriţcu**

Kenji **Maillard** Guido **Martínez** Exequiel **Rivas**

How to define effects in F*?

How to use effects in F*?

```
using  state = nat
```

```
let decr () : unit
= let x = get () in
  put (x - 1)
```

How to use effects in F*?

```
using  state = nat
```

```
let decr () : St unit
= let x = get () in
  put (x - 1)
```

How to use effects in F*?

using state = nat

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```

How to use effects in F*?

using state = nat

```
let decr () :
  St unit
     (requires λ s₀. 1 ≤ s₀)
     (ensures λ s₀ s₁. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```

How to use effects in F*?

using state = nat

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```

```
get () : St nat (requires λ s₀. ⊤)
                (ensures λ s₀ s₁ r. s₁ = s₀ = r)
```

```
put (x : nat) : St unit (requires λ s₀. ⊤)
                        (ensures λ s₀ s₁ r. s₁ = x)
```

```
bind (get ()) (λ x,
  put (x - 1)
)
```

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁
= let x = get () in
  put (x - 1)
```

```
get () : St nat (requires λ s₀. ⊤)
                (ensures λ s₀ s₁ r. s₁ = s₀ = r)
```

```
put (x : nat) : St unit (requires λ s₀. ⊤)
                        (ensures λ s₀ s₁ r. s₁ = x)
```

```
bind (get ()) (λ x,
  put (x - 1)
)
```

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁
= let x = get () in
  put (x - 1)
```

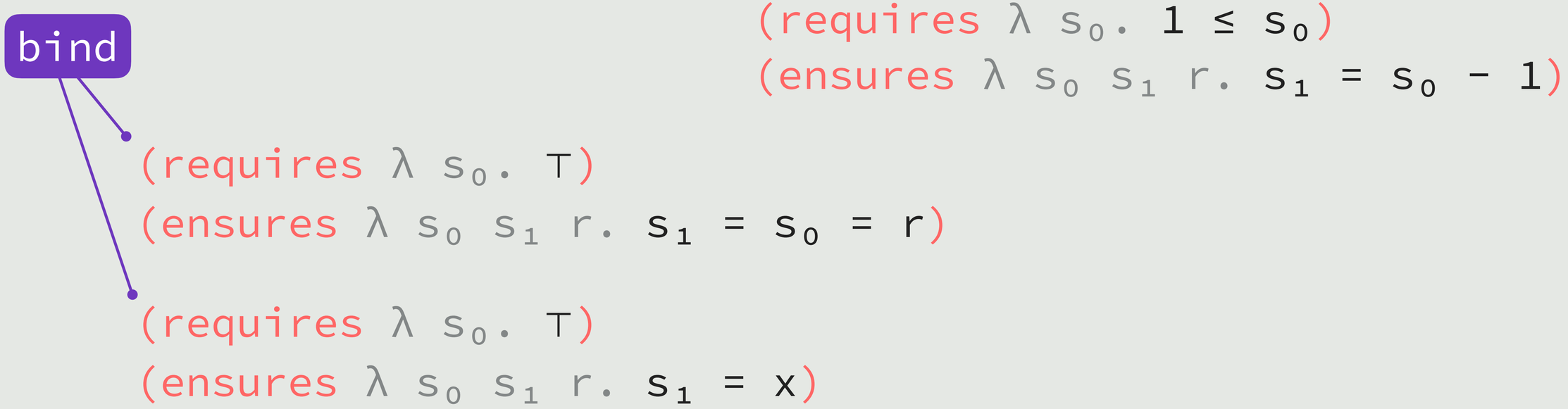
```
get () : St nat (requires λ s₀. ⊤)
                (ensures λ s₀ s₁ r. s₁ = s₀ = r)
```

```
put (x : nat) : St unit (requires λ s₀. ⊤)
                        (ensures λ s₀ s₁ r. s₁ = x)
```

Computation level

```
bind (get ()) (λ x,
   put (x - 1)
)
```

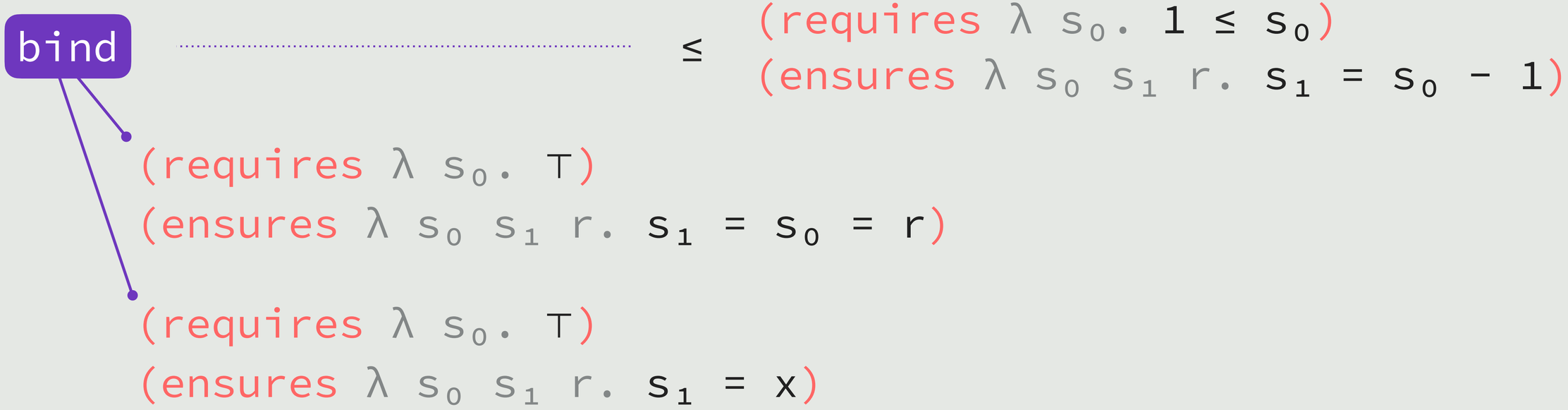
Specification level



Computation level

```
bind (get ()) (λ x,
   put (x - 1)
)
```

Specification level



Dijkstra monads

```
D (A : Type) (w : W A) : Type
```

ordered specification *monad*

Dijkstra monads

```
D (A : Type) (w : W A) : Type
```

ordered specification *monad*

```
returnᴰ (x : A) : D A (returnᵂ x)
```

Dijkstra monads

```
D (A : Type) (w : W A) : Type
```

ordered specification *monad*

```
returnᴰ (x : A) : D A (returnᵂ x)

bindᴰ (c : D A w) (f : (x : A) → D B (wf x)) : D B (bindᵂ w wf)
```

Dijkstra monads

```
D (A : Type) (w : W A) : Type
```

ordered specification *monad*

```
returnᴰ (x : A) : D A (returnᵂ x)

bindᴰ (c : D A w) (f : (x : A) → D B (wf x)) : D B (bindᵂ w wf)

subcompᴰ (c : D A w) : Pure (D A w')
                            (requires w ≤ w')
                            (ensures λ r, r = c)
```

Dijkstra monads *for all* (Maillard *et al.* ICFP 2019)


```
D (A : Type) (w : W A) = { c : M A | θ c ≤ w }
```

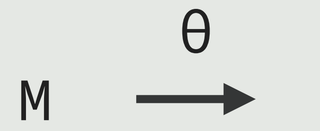
e.g.

computation monad St **specification monad**

```
M A = S → A × S

returnᴹ x = λ s₀. (x,s₀)

bindᴹ c f = λ s₀.
let (x,s₁) = c s₀ in f x s₁
```

```
W A = post A → pre

returnᵂ x = λ P s₀. P x s₀

bindᵂ w wf =
λ P. w (λ s₁ x. wf x P s₁)
```

Dijkstra monads *for all* (Maillard *et al.* ICFP 2019)




```
D (A : Type) (w : W A) = { c : M A | θ c ≤ w }
```

e.g.

computation monad `St` **specification monad**

```
M A = S → A × S

returnᴹ x = λ s₀. (x,s₀)

bindᴹ c f = λ s₀.
let (x,s₁) = c s₀ in f x s₁
```

```
W A = post A → pre

returnᵂ x = λ P s₀. P x s₀

bindᵂ w wf =
λ P. w (λ s₁ x. wf x P s₁)
```

```
θ c = λ P s₀. let (x,s₁) = c s₀ in P x s₁
```

using state = nat

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```

```
using state = nat = { x : int | 0 ≤ x }
```

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```

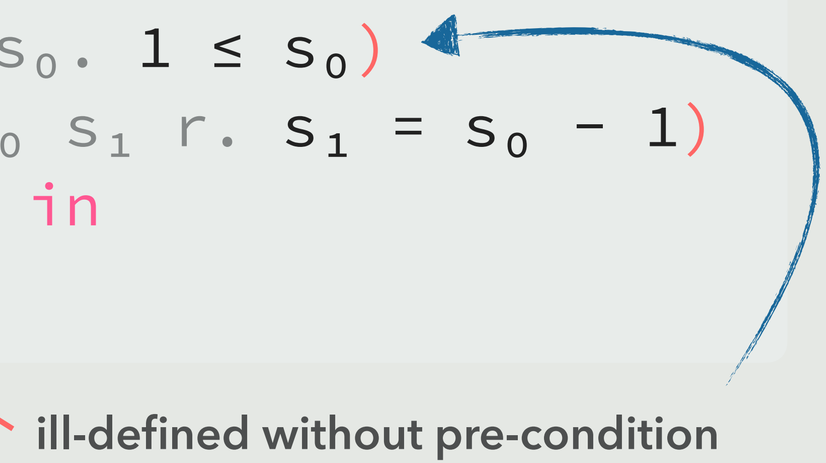
using state = nat = { x : int | 0 ≤ x }

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```

ill-defined without pre-condition

```
using state = nat = { x : int | 0 ≤ x }
```

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```



ill-defined without pre-condition

```
let head (l : list A) :
  Pure A (requires l ≠ []) (ensures λ r. ⊤)
= match l with
  | x :: l' → x
```

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```

```
let head (l : list A) :
  Pure A (requires l ≠ []) (ensures λ r. ⊤)
= match l with
  | x :: l' → x
```

```
let x = get () in
let y = get () in
assert (x = y)
```

```
let decr () :
  St unit
    (requires λ s₀. 1 ≤ s₀)
    (ensures λ s₀ s₁ r. s₁ = s₀ - 1)
= let x = get () in
  put (x - 1)
```

```
let head (l : list A) :
  Pure A (requires l ≠ []) (ensures λ r. ⊤)
= match l with
  | x :: l' → x
```

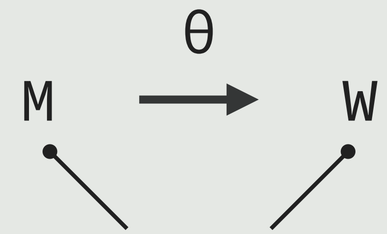
```
let x = get () in
let y = get () in
assert (x = y)
```

need the pre-condition *deeply*

Dijkstra monads *for all*


```
D (A : Type) (w : W A) = { c : M A | θ c ≤ w }
```

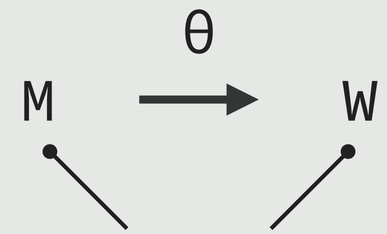
Partial Dijkstra monads *for all*



monads with require operator

```
D (A : Type) (w : W A) = { c : M A | θ c ≤ w }
```

Partial Dijkstra monads *for all*

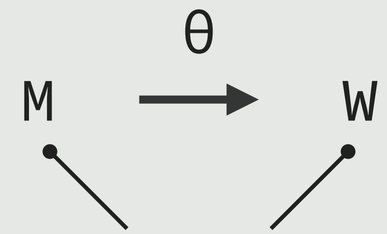


monads with require operator

```
requireᴹ (P : ℙ) : M P
```

```
D (A : Type) (w : W A) = { c : M A | θ c ≤ w }
```

Partial Dijkstra monads *for all*

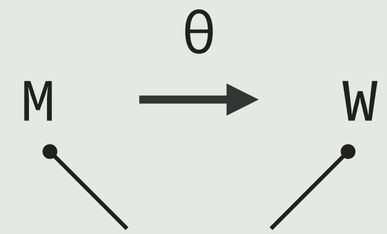


monads with require operator

```
requireᴹ (P : ℙ) : M P

requireᴰ (P : ℙ) : D P (requireᵂ P)
```

```
D (A : Type) (w : W A) = { c : M A | θ c ≤ w }
```

Partial Dijkstra monads *for all*



monads with require operator

```
requireᴹ (P : ℙ) : M P

requireᴰ (P : ℙ) : D P (requireᵂ P)

θ (requireᴹ P) = requireᵂ P
```

```
D (A : Type) (w : W A) = { c : M A | θ c ≤ w }
```

State partial Dijkstra monad

```
M A = S → G (A × S)
```

State partial Dijkstra monad

```
M A = S → G (A × S)
```

where `G B = Σ (pre : ℙ). pre → B`

State partial Dijkstra monad

$$M\ A = S \to G\ (A \times S)$$

where $G\ B = \Sigma\ (pre : \mathbb{P}).\ pre \to B$

given by (pre, f) with $f : pre \to B$

State partial Dijkstra monad

$$M\ A = S \to G\ (A \times S)$$

where $G\ B = \Sigma\ (pre : \mathbb{P}).\ pre \to B$

given by (pre, f) with $f : pre \to B$

$$\theta\ c = \lambda\ P\ s_0,$$

State partial Dijkstra monad

```
M A = S → G (A × S)
```

where `G B = Σ (pre : ℙ). pre → B`

given by `(pre, f)` with `f : pre → B`

```
θ c =
  λ P s₀,
    let (pre, f) := c s₀ in
```

State partial Dijkstra monad

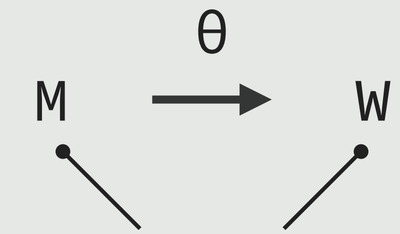
$$M\ A = S \to G\ (A \times S)$$

where $G\ B = \Sigma\ (pre : \mathbb{P}).\ pre \to B$

given by (pre, f) with $f : pre \to B$

```
θ c =
  λ P s₀,
    let (pre, f) := c s₀ in
    ∃ (h : pre). let (x, s₁) := f h in P x s₁
```

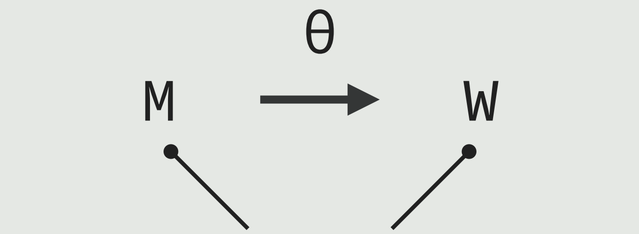
Partial Dijkstra monads *for all*



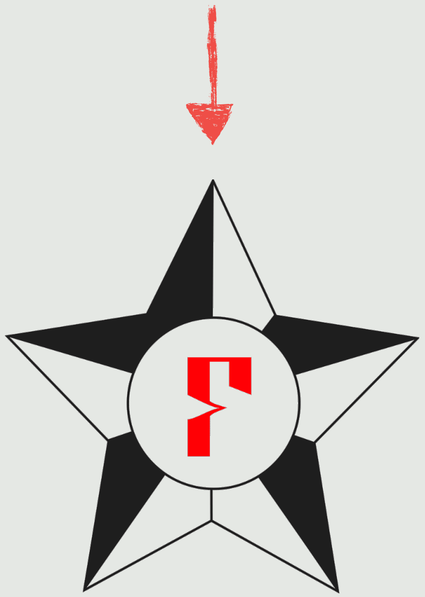
monads with require operator

partial effect in F*

Partial Dijkstra monads *for all*

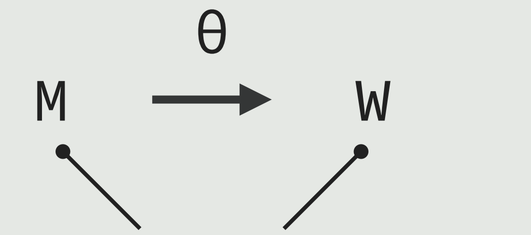


monads with require operator



partial effect in F*

Partial Dijkstra monads *for all*



monads with require operator

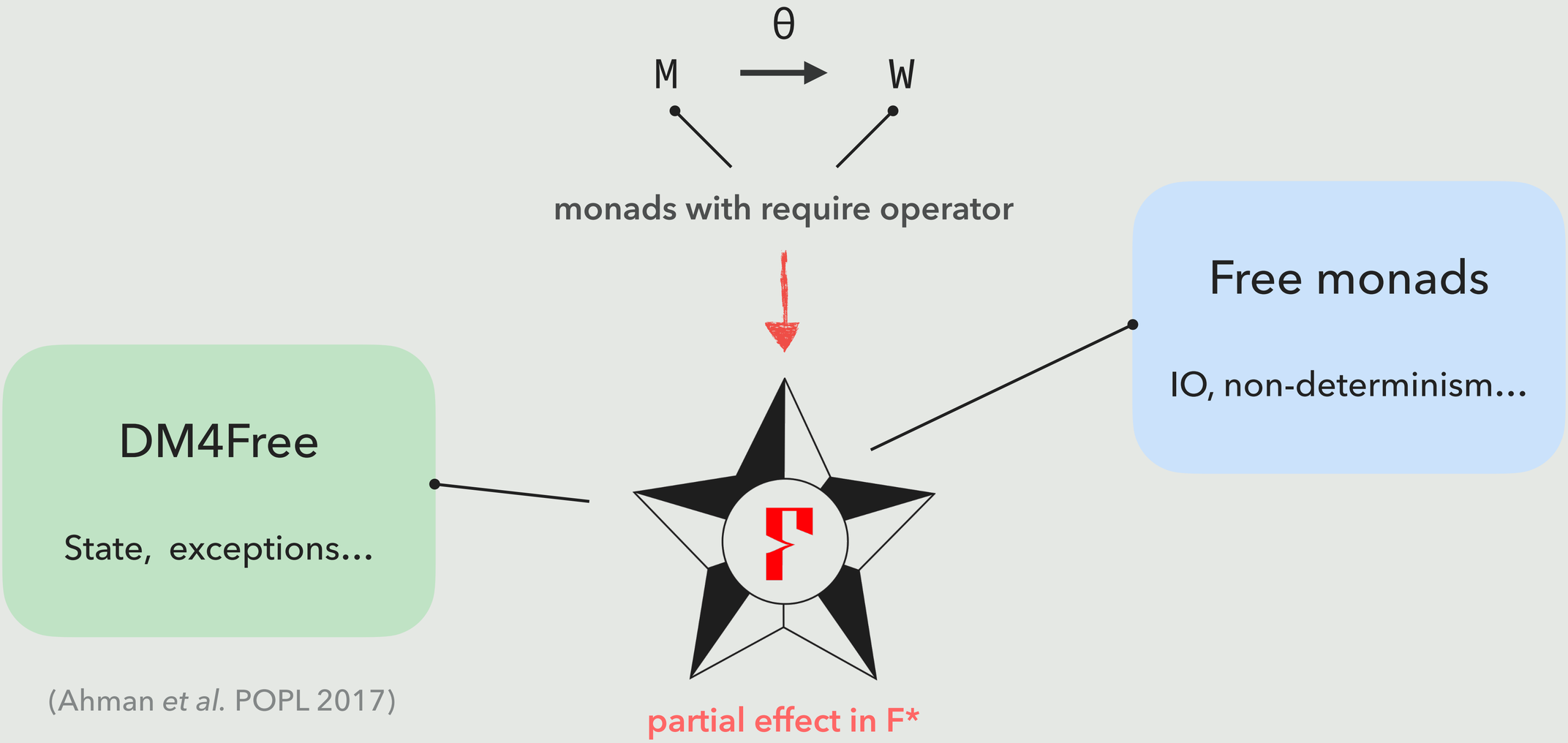
DM4Free

State, exceptions…

(Ahman *et al.* POPL 2017)

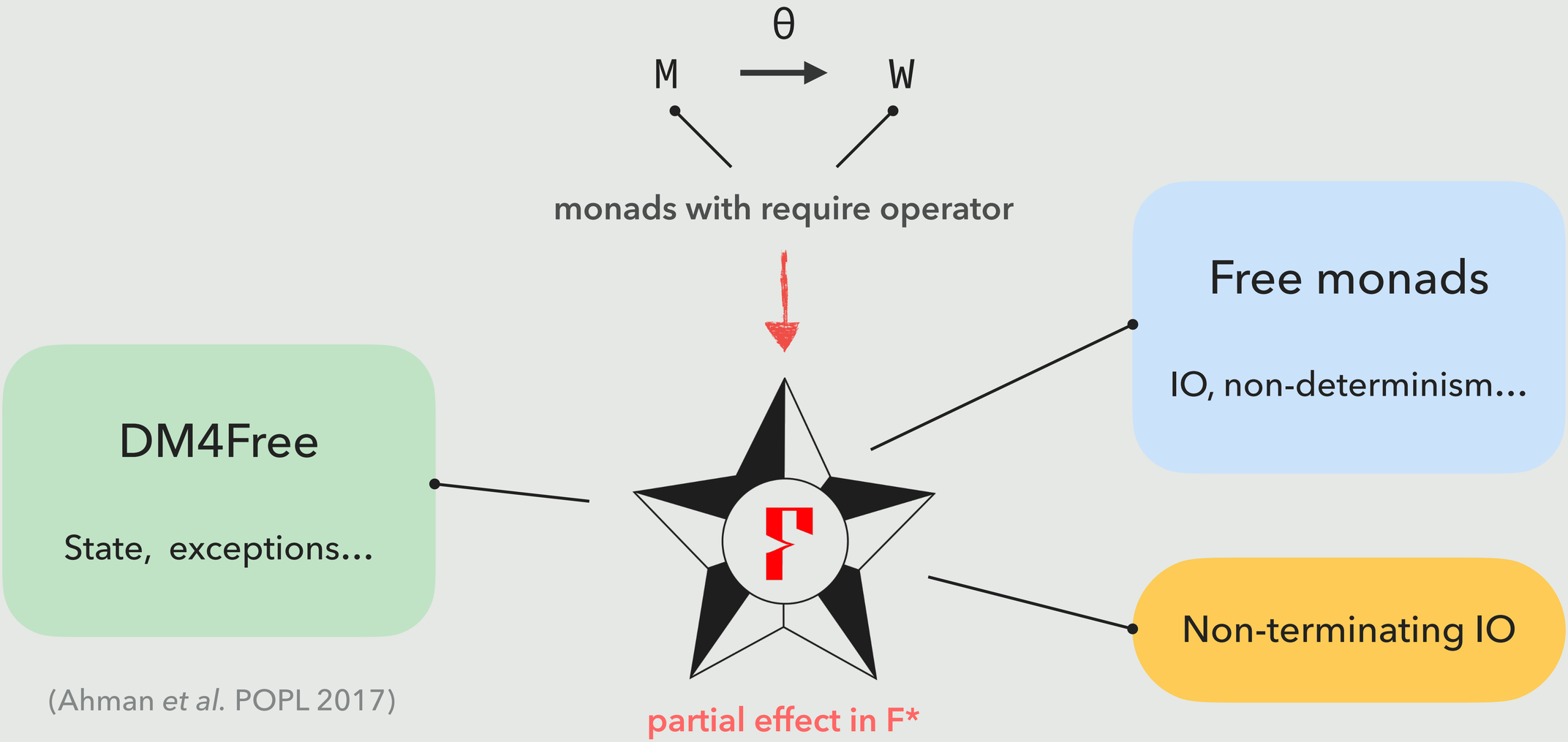
partial effect in F*

Partial Dijkstra monads *for all*



(Ahman *et al.* POPL 2017)

Partial Dijkstra monads *for all*



(Ahman *et al.* POPL 2017)