# A glance at F⋆'s effect system
## ... And Dijkstra Monads for All

**Guido Martínez** — CIFASIS-CONICET Rosario

## Intro: Dijkstra Monads and F⋆ effects

*Dijkstra Monads*: monads indexed by *specifications* in the form of *weakest preconditions (WPs)*. The WPs are in the *continuation* monad to prop:

```
type purewp a = (a → prop) → prop

let returnwp (x:α) : purewp α= λp → p x
let bindwp (wp : purepw α) (f : α→ purewp β) : purewp β=
      λp → wp (λ x → f x p)
```

```
val ret5 : unit → PURE int (λ p → p 5)
let ret5 () = 5

val incr : x:int → PURE int (λ p → ∀y. y > x ⟹ p y)
let incr x = x + 1
```

The Dijkstra monad allows to compute composite specifications

```
let incr2 x = let y = incr x in incr y
```

F⋆ infers:

```
val incr2 : x:int → PURE int (bindwp (incr_wp x) incr_wp)
val incr2 : x:int → PURE int
                        (λ p → ∀y. y > x ⟹ ∀z. z > y ⟹ p z)
```

## A toy version of F⋆'s primitive state

```
type stwp a = (a → state → prop) → (state → prop)

let returnwp (x:α) : stwp α= λp s → p x s
let bindwp (wp : stwp α) (f : α→ stwp β) : stwp β=
      λp s → wp (λ x s' → f x p s') s

val get : unit → ST state (λ p s → p s s)
val set : s:state → ST unit (λ p s → p () s)
```

**How to obtain** these monads and specifications for other effects? Previously, either fully by hand or, for a rather limited and amicable subset, automatically from a translation (DM4Free).

Dijkstra monads can be derived from an **effect observation** between a computational monad and a specification monad

## Forging new effects via DM4All

Take a computational monad,

```
type st a = state → a ∗ state
let return (x:α) : st α= λs → (x, s)
let bind (m : st α) (f : α→ st β) : st β= λs → let (y, s') = m s
                                               in f y s'
```

and a specification monad (such as stwp above), and an *effect observation* between them (a monad morphism).

```
val θ: st α→ stwp α
let θ(m) = λp s → let (y, s') = m s in p y s'
```

The effect observation computes specifications for actions. Then we can construct a Dijkstra Monad and a new F⋆ effect

```
new_effect ST { comp = (st, return, bind)
              ; spec = (stwp, returnwp, bindwp)
              ; observ = (θ) }

let get () : ST state (λ p s → p s s) = reflect (λ s → (s, s))
let set (s:state) : ST unit (λ p _ → p () s) = reflect (λ _ → ((), s))

let switch (s:state) = let s0 = get () in set s; s0
val switch : s:state → ST state (λ p s0 → p s0 s)
```

Choice of both monads and observation ***fully up to the user!***

## New & cooler effects are possible!

### Nondeterminism

Two different possible interpretations (demonic vs angelic):

```
type nd a = list a
type ndwp a = purewp a

let θ_dem (m : nd α) : ndwp α= λp → ∀x. elem x m ⟹ p x
let θ_ang (m : nd α) : ndwp α= λp → ∃x. elem x m ∧ p x

new_effect NDD { ... observ = (θ_dem) }
```

Computing pythagorean triples nondeterministically, with proof.

```
val choose : x:α → y:α → NDD α(λ p → p x ∧ p y)
let choose x y = reflect [x;y]

val fail : unit → NDD α(λ p → True)
let fail () = reflect []

let guard (b:bool) : NDD unit (λ p → b ⟹ p ()) =
    if b then () else fail ()

let rec pick_from (l : list α) : NDD α(λ p → ∀x. elem x l ⟹ p x) =
    match l with
    | [] → fail ()
    | x::xs → if choose true false then x else pick_from xs

let pyths () : NDD (int & int & int)
                        (λ p → ∀x y z. x∗x + y∗y == z∗z ⟹ p (x,y,z)) =
    let l = [1..10] in
    let x = pick_from l in let y = pick_from l in let z = pick_from l in
    guard (x∗x + y∗y = z∗z);
    (x, y, z)
```

### Input-output

Several different specification monads are possible (see paper!)

```
type io a = | Read : (input → io a) → io a
            | Write : output → io a → io a
            | Return : a → io a

let iowp_forget a = (a → prop) → prop
let iowp_wlocal a = (a → list output → prop) → prop

type event = | In : input → event | Out : output → event

let iowp_rwhist a = (a → list event → prop)
                          → list event → prop
```