HOPE 2022

Verifying non-terminating programs with IO in F*

Cezar-Constantin **Andrici**, Théo **Winterhalter**,

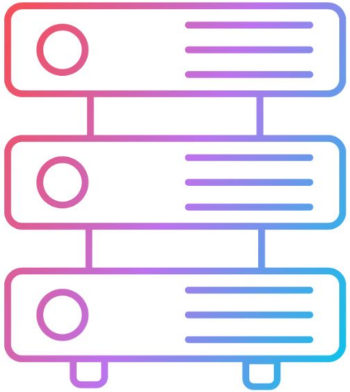
Cătălin **Hriţcu**, Exequiel **Rivas**

11 September 2022

Goal:

practical way to verify *functional correctness* for higher-order non-terminating **I**nput-**O**utput programs practical goal:

Verify a simple Web Server



Non-terminating, non-trivial IO trace properties

What to expect

1. We use F★, but the ideas are general;
2. Using monads to do verification:
   - of terminating programs;
   - of non-terminating programs;
3. We reason about non-terminating runs by using infinite traces.
4. To verify our Web Server, we mix verification of terminating and non-terminating programs;

Why the proof-oriented programming language F★?

(Swamy *et al.* POPL 2016)

F★'s Advantages:

1. Write, specify and verify the program in the same language;
2. User-defined effects with specifications:
   - one effect for termination and one for possible non-termination;
   - hides the binds and returns;
3. Built-in support for verification of higher-order;
4. SMT based-automation.

How to verify terminating programs

Program example: Echo

```
let echo (fd:file_descr) =
  let msg = read fd in
  write fd msg
```

About traces

```
let echo (fd:file_descr) =
  let msg = read fd in
  write fd msg
```

Trace = sequence of IO events that occur during a specific run of the program

[ERead fd_1 "Hello!"; EWrite (fd_1,"Hello!")]

About trace properties

```
let echo (fd:file_descr) =
  let msg = read fd in
  write fd msg
```

Trace = sequence of IO events that occur during a specific run of the program

[ERead fd_1 “Hello!”; EWrite (fd_1,“Hello!”)]

Example of trace properties:

∀ t. t terminates

About trace properties

```
let echo (fd:file_descr) =
  let msg = read fd in
  write fd msg
```

Trace = sequence of IO events that occur during a specific run of the program

$$[\texttt{ERead } fd_1 \text{ “Hello!”}; \texttt{ EWrite } (fd_1, \text{“Hello!”})]$$

Example of trace properties:

$\forall$ t. t terminates

$\forall$ t. $\exists$ msg. t = [ERead fd msg; EWrite (fd,msg)]

About trace properties

```
let echo (fd:file_descr) =
  let msg = read fd in
  write fd msg
```

```
∀ t. ∃ msg. t = [ERead fd msg; EWrite (fd,msg)]
```

Specification of Echo

```
let echo (fd:file_descr) :
  IO unit
   (requires λ h → is_open fd h)
   (ensures  λ h r t → ∃ msg. t = [ERead fd msg; EWrite fd msg]) =
   let msg = read fd in
   write fd msg
```

Echo - Effect

```
let echo (fd:file_descr) :
 IO unit
  (requires λ h → is_open fd h)
  (ensures  λ h r t → ∃ msg. t = [ERead fd msg; EWrite fd msg]) =
  let msg = read fd in
  write fd msg
```

Echo - pre-condition

```
let echo (fd:file_descr) :
  IO unit
   (requires λ h → is_open fd h)
   (ensures  λ h r t → ∃ msg. t = [ERead fd msg; EWrite fd msg]) =
   let msg = read fd in
   write fd msg
```

Echo - pre-condition

```
let echo (fd:file_descr) :
  IO unit
   (requires λ h → is_open fd h)
   (ensures  λ h r t → ∃ msg. t = [ERead fd msg; EWrite fd msg]) =
   let msg = read fd in
   write fd msg
```

Echo - post-condition

```
let echo (fd:file_descr) :
  IO unit
   (requires λ h → is_open fd h)
   (ensures  λ h r t → ∃ msg. t = [ERead fd msg; EWrite fd msg]) =
   let msg = read fd in
   write fd msg
```

Echo - post-condition

```
let echo (fd:file_descr) :
  IO unit
   (requires λ h → is_open fd h)
   (ensures  λ h r t → ∃ msg. t = [ERead fd msg; EWrite fd msg]) =
   let msg = read fd in
   write fd msg
```

Verifying Echo

```
let echo (fd:file_descr) :
  IO unit
   (requires λ h → is_open fd h)
   (ensures  λ h r t → ∃ msg. t = [ERead fd msg; EWrite fd msg]) =
   let msg = read fd in
   write fd msg
```

F★ can prove this automatically.

How *effects* work in F★

Dijkstra monads (Swamy *et al.* PLDI 2013)

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

Dijkstra monads (Swamy *et al.* PLDI 2013)

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

our specification monad for IO

```
pre    = event★ → prop
post a = event★ → a → prop

W a = post a → pre
```

`event★` - type of finite traces

predicate transformer that maps post-conditions to pre-conditions

Dijkstra monads (Swamy *et al.* PLDI 2013)

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

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

Dijkstra monads (Swamy *et al.* PLDI 2013)

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

returnᴰ (x : a) : D a (returnᵂ x)

bindᴰ (w : W a) (wf : a → W b) ... : D b (bindᵂ w wf)
```

Dijkstra monads (Swamy *et al.* PLDI 2013)

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

returnᴰ (x : a) : D a (returnᵂ x)

bindᴰ (w : W a) (wf : a → W b) ... : D b (bindᵂ w wf)

actᴰ ... : D a (actᵂ ...)
```

Dijkstra monads

(Swamy *et al.* PLDI 2013)

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

return^D (x : a) : D a (return^W x)

bind^D (w : W a) (wf : a → W b) ... : D b (bind^W w wf)

act^D ... : D a (act^W ...)
```

```
val read : (fd:file_descr) → IO string
 (requires (λ history          → is_open fd history))
 (ensures  (λ history msg lt → lt = [ERead fd msg]))
```

Back to our example

```
let echo (fd:file_descr) :
  IO unit
   (requires λ h → is_open fd h)
   (ensures  λ h r t → ∃ msg. t = [ERead fd msg; EWrite fd msg]) =
   let msg = read fd in
   write fd msg
```

$\text{bind}^{W}\ (\text{read}^{W}\ \text{fd})\ (\text{write}^{W}\ \text{fd}) \leq \text{wp}$

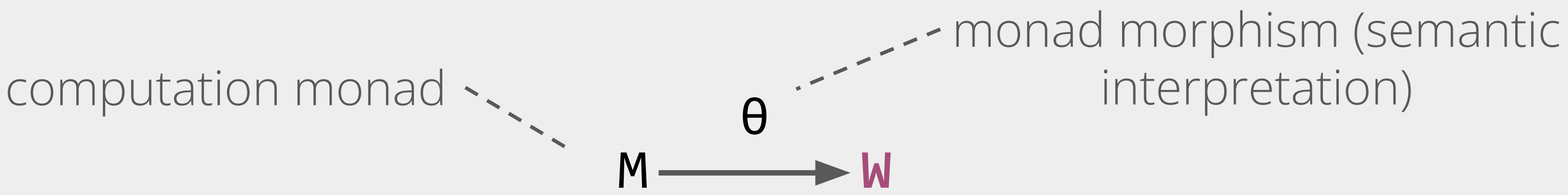
Back to our example

```
let echo (fd:file_descr) :
  IO unit
   (requires λ h → is_open fd h)
   (ensures  λ h r t → ∃ msg. t = [ERead fd msg; EWrite fd msg]) =
   let msg = read fd in
   write fd msg
```

$$\text{bind}^{W}\ (\text{read}^{W}\ \text{fd})\ (\text{write}^{W}\ \text{fd}) \leq \text{wp}$$

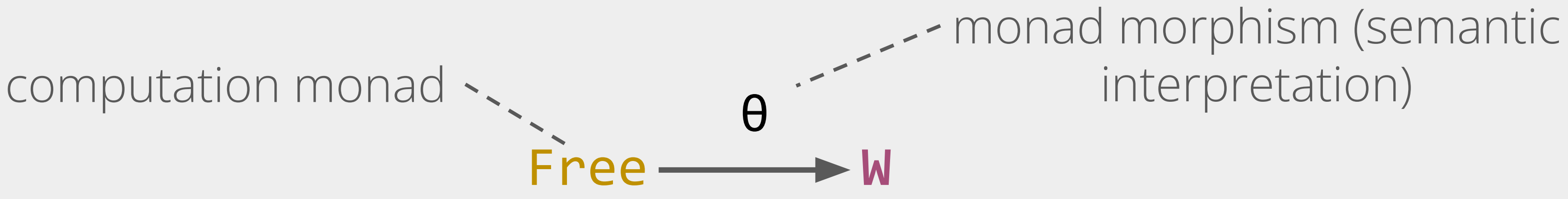
Defining `IO` effect for terminating programs

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



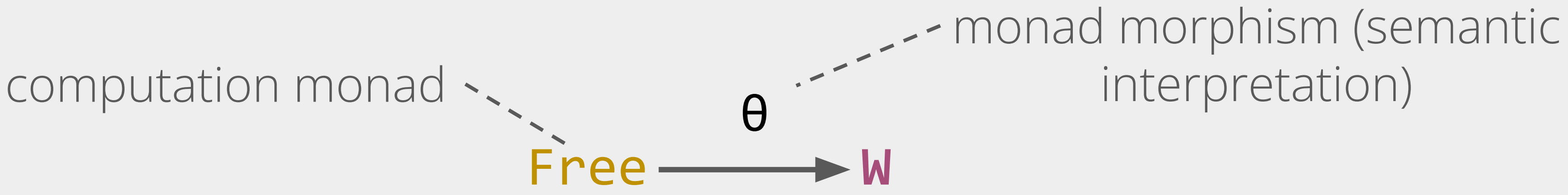

```
D a (w : W a) = { c : M a | θ c ≤ w }
```

Our IO effect for termination




```
IO a (w : W a) = { c : Free a | θ c ≤ w }
```

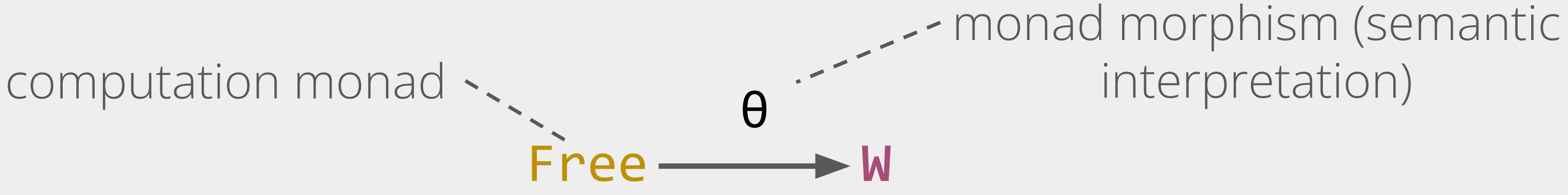
Our IO effect for termination



$$\text{IO}\ a\ (w : W\ a) = \{\ c : \text{Free}\ a \mid \theta\ c \leq w\ \}$$

```
Free #sig a =
| Call : (o : sig.act) → sig.in o → (sig.out o → Free a) → Free a
| Return : a → Free a
```

Our IO effect for termination

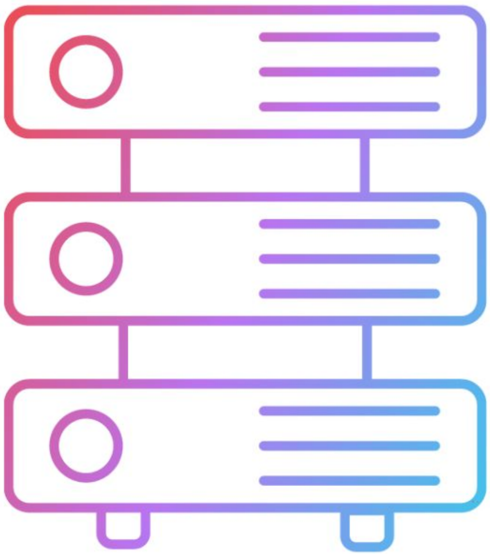


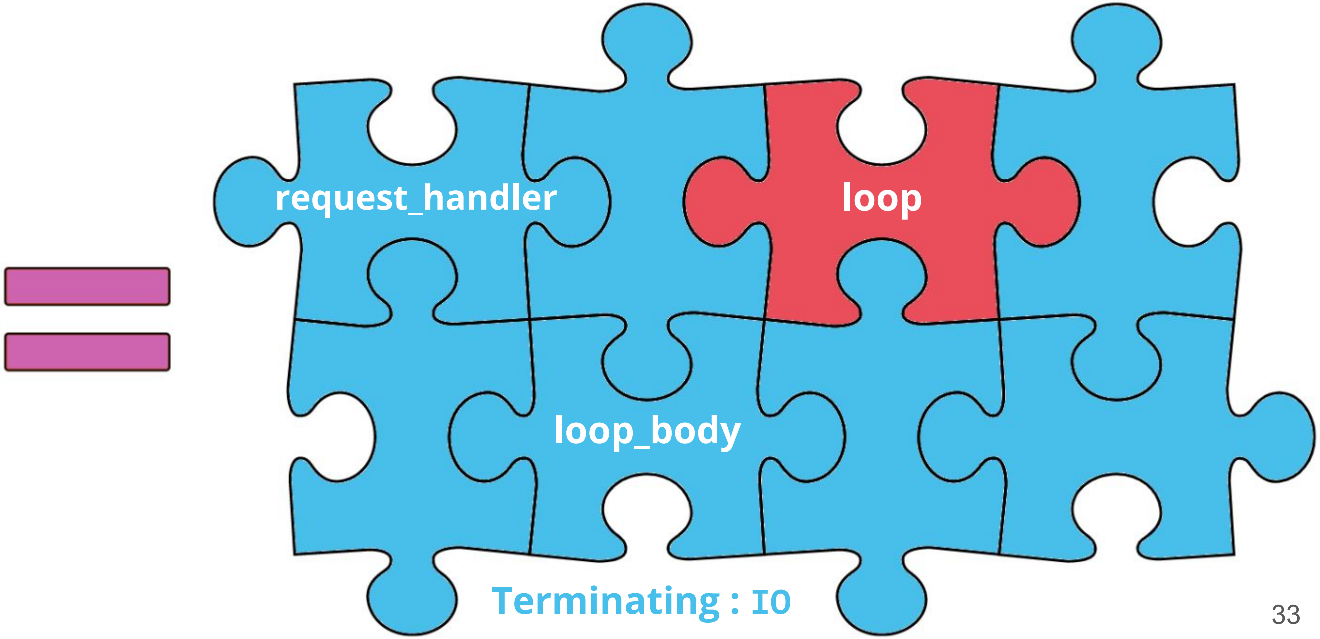
$$\mathtt{IO}\ a\ (w : \mathtt{W}\ a) = \{\ c : \mathtt{Free}\ a \mid \theta\ c \le w\ \}$$

```
θ c =
  match c with
  | Return x → return^W x
  | Call act args fnc →
      bind^W (act^W args) (λ r → θ (fnc r))
```

Using IO, we verified the terminating parts of the Web Server

Web Server





Program example: Forever Echo

```
let loop_echo fd = repeat echo fd
```

- F★ does not support co-induction.

Program example: Forever Echo

```
let loop_echo fd = repeat echo fd
```

- F★ does not support co-induction.
- This is what we would like, but can't write:

ML

```
let rec iter f i =
  match f i with
  | Inl j → iter f j
  | Inr x → x
```

Program example: Forever Echo

```
let loop_echo fd = repeat echo fd
```

- F★ does not support co-induction.
- This is what we would like, but can't write:

```
let rec iter f i =                                              ML
  match f i with
  | Inl j → iter f j
  | Inr x → x
```

Add extra constructor to **Free** monad corresponding to unbounded iteration.

```
Free a = | ...
| Iter : f:(b → Free (b + c)) → i : b → (c → Free a) → Free a
```

`repeat` can be written using `iter`.

IODiv for non-termination

specification monad

$$\text{pre} = \text{event}^{\star} \rightarrow \text{prop}$$
$$\text{post}\ A = ((\text{event}^{\star} \times A) + \text{event}^{\omega}) \rightarrow \text{prop}$$

type of infinite traces
(stream of events)

IODiv - monad morphism

```
θ c =
  match c with
  | …
  | Iter f i fnc → bindᵂ (iterᵂ (fun j → θ (f j)) i)
                         (λ r → θ (fnc r))
```

IODiv - monad morphism

```
θ c =
  match c with
  | …
  | Iter f i fnc → bindᵂ (iterᵂ (fun j → θ (f j)) i)
                         (λ r → θ (fnc r))
```

IODiv - monad morphism

```
θ c =
  match c with
  | …
  | Iter f i fnc → bindᵂ (iterᵂ (fun j → θ (f j)) i)
                         (λ r → θ (fnc r))

iterᵂ w i ≈
  match w i with
  | Inl j → bindᵂ tauᵂ (iterᵂ w j)
  | Inr x → returnᵂ x
```

IODiv - monad morphism

```
θ c =
  match c with
  | …
  | Iter f i fnc → bindᵂ (iterᵂ (fun j → θ (f j)) i)
                         (λ r → θ (fnc r))

iterᵂ w i ≈
  match w i with
  | Inl j → bindᵂ tauᵂ (iterᵂ w j)
  | Inr x → returnᵂ x
```

Tau is a silent step (Dijkstra Monads for Ever, ITrees)

```
ERead fd₁ m₁; EWrite (fd,m₁); Tau; ERead fd m₂; EWrite (fd,m₂); Tau; …
```

Take advantage of SMT automation

```
let loop_echo (fd:file_descr) :
  IODiv unit
   (requires λ h → is_open client h)
   (ensures  λ h run → diverges run ∧
        run ≈ [ERead fd m; EWrite (fd,m); ERead fd m;...]) =
  repeat echo fd
```

This does not verify automatically yet.

Take advantage of SMT automation

```
let loop_echo (fd:file_descr) :
  IODiv unit
   (requires λ h → is_open client h)
   (ensures  λ h run → diverges run ∧
        run ≈ [ERead fd m; EWrite (fd,m); ERead fd m;...]) =
  repeat echo fd
```

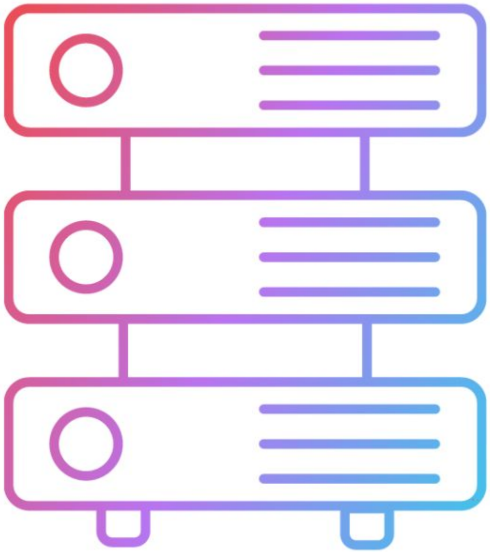
This does not verify automatically yet.

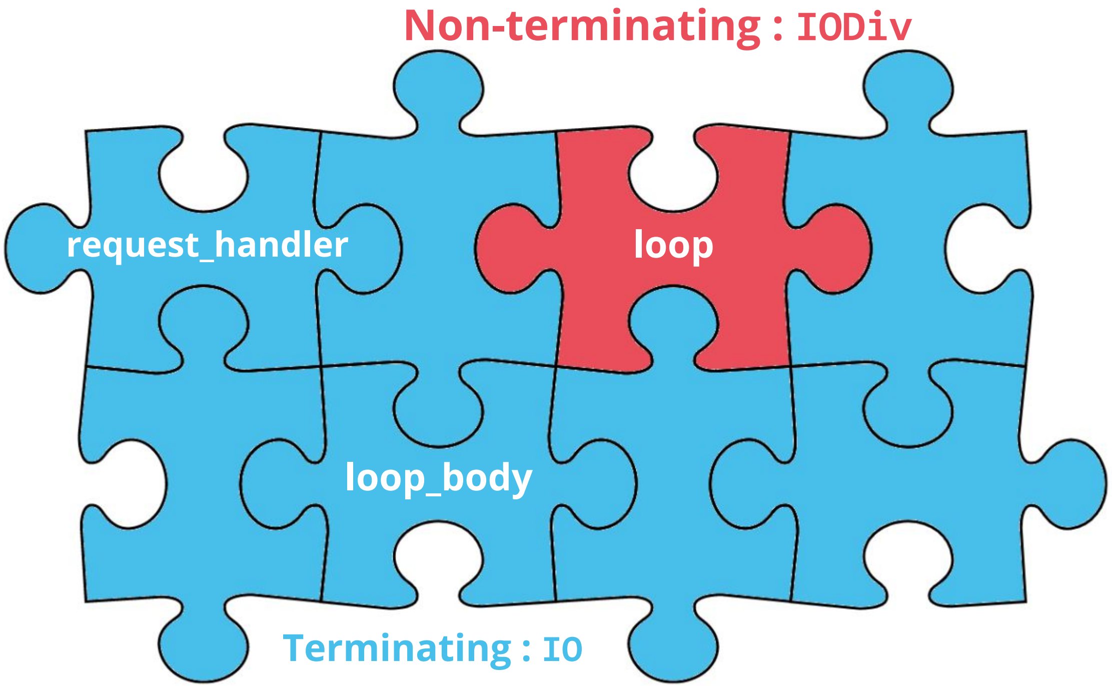
We actively tune the verification condition to take advantage of the SMT:

- Keeping the history backwards simplifies verification of pre-conditions;
- Making definitions abstract for the SMT;
- Changing **bind**^{W} simplified by a factor of 4 the verification condition.

We want to use IODiv to verify only non-terminating parts

Web Server





IODiv is more complex for the SMT than IO

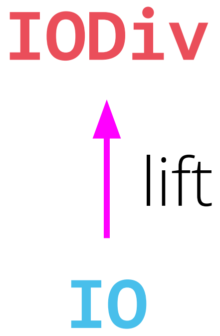
Sub-effecting

`IODiv`

↑ lift

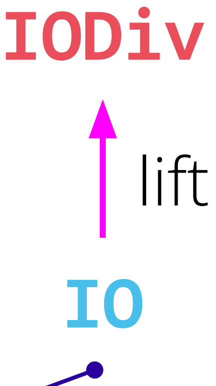
`IO`

Sub-effecting



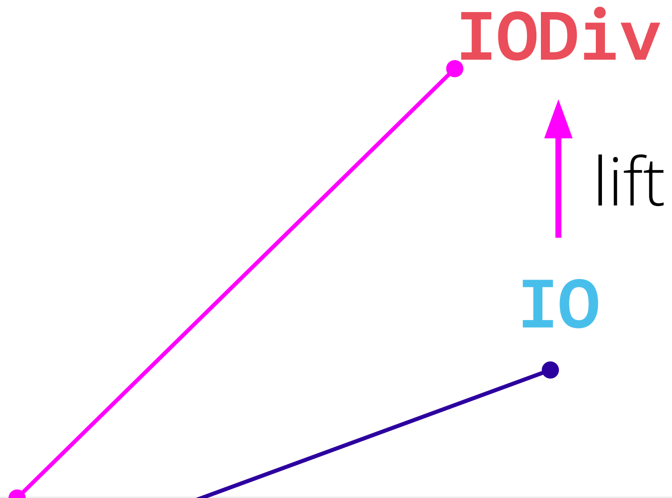

```
let loop_echo (fd:file_descr) : IODiv unit ...
  repeat echo fd
```

Sub-effecting




```
let loop_echo (fd:file_descr) : IODiv unit ...
  repeat echo fd
```

Sub-effecting




```
let loop_echo (fd:file_descr) : IODiv unit ...
  repeat echo fd
```

Conclusion

- Dijkstra monads with Free monads seem fit for the task;
- F★ hides the complexity of the monads;
- Tuning the verification conditions is necessary;
- Sub-effecting is important to alleviate the proof burden;
- There is HOPE this can be practical.

Conclusion

- Dijkstra monads with Free monads seem fit for the task;
- F★ hides the complexity of the monads;
- Tuning the verification conditions is necessary;
- Sub-effecting is important to alleviate the proof burden;
- There is HOPE this can be practical.

Ongoing and future work

- Tune verification conditions to take advantage of automation;
- Study how to verify properties of infinite runs such as liveness;
- Case study: verify a stateless web server that serves files over HTTP;
- Add State and Exceptions effects;
- Part of secure F★ - ML interoperability line of work;
- **Hiring!** My team is looking for a PostDoc to work on formal verification!