

Comodels as a gateway for interacting with the **external world**

Danel Ahman

(joint work with Andrej Bauer)

Shonan, 28 March 2019

Comodels as a gateway for interacting with the **external world**

Danel Ahman

(joint work with Andrej Bauer)



Shonan, 28 March 2019

Computational effects in FP

Computational effects in FP

- Using **monads** (as in HASKELL)

```
type St a = String → (a, String)
```

```
f :: St a → St (a, a)
```

```
f c = c >>= (\x → c >>= (\y → return (x, y)))
```

- Using **alg. effects** and **handlers** (as in EFF, FRANK, KOKA)

```
effect Get : int
```

```
effect Put : int → unit
```

```
(*: int → a*int!{} *)
```

```
let g (c: unit → a!{ Get, Put }) =
```

```
  with st_h handle (perform (Put 42); c ())
```

Computational effects in FP

- Using **monads** (as in HASKELL)

```
type St a = String → (a, String)
```

```
f :: St a → St (a, a)
```

```
f c = c >>= (\x → c >>= (\y → return (x, y)))
```

- Using **alg. effects** and **handlers** (as in EFF, FRANK, KOKA)

```
effect Get : int
```

```
effect Put : int → unit
```

```
(*: int → a*int!{} *)
```

```
let g (c: unit → a!{Get, Put}) =
```

```
  with st_h handle (perform (Put 42); c ())
```

- Both are good for **faking comp. effects** in a pure language!

But what about effects that need access to the **external world**?

External world in FP

- Declare a **signature** of monads or algebraic effects

```
type IO a
```

```
openFile  :: FilePath → IOMode → IO Handle
```

```
hGetLine  :: Handle → IO String
```

```
hClose    :: Handle → IO ()
```

```
effect Read    : string
```

```
effect Raise   : string → empty
```

```
effect RandomInt    : int → int
```

```
effect RandomFloat : float → float
```

- And then treat them **specially** in the compiler, e.g.,

```
let rec top_handle op =
```

```
    match op with
```

```
    | ...
```

External world in FP

External world in FP



Ohad 🤖 12:17 PM

Can I do file IO (or just O) in Eff?

External world in FP



Ohad 🤖 12:17 PM

Can I do file IO (or just O) in Eff?



Žiga Lukšič 12:18 PM

not currently

External world in FP



Ohad 🐼 12:17 PM

Can I do file IO (or just O) in Eff?



Žiga Lukšič 12:18 PM

not currently



Ohad 🐼 8:35 PM

So here's the hack I added. We should do something a bit more principled

In `pervasives.eff`:

```
effect Write : (string*string) -> unit
```

in `eval.ml`, under `let rec top_handle op =` add the case:

```
| "Write" ->
  (match v with
  | V.Tuple vs ->
    let (file_name :: str :: _) = List.map V.to_str vs in
    let file_handle = open_out_gen
      [Open_wronly
       ;Open_append
       ;Open_creat
       ;Open_text
      ] 0o666 file_name in
    Printf.fprintf file_handle "%s" str;
    close_out file_handle;
    top_handle (k V.unit_value)
  )
```

External world in FP



Ohad 12:17 PM

Can I do file IO (or just O) in Eff?



Žiga Lukšič 12:18 PM

not currently



Ohad 8:35 PM

So here's the hack I added. We should do something a bit more principled

In `pervasives.eff`:

```
effect Write : (string*string) -> unit
```

in `eval.ml`, under `let rec top_handle op =` add the case:

```
| "Write" ->
  (match v with
  | V.Tuple vs ->
    let (file_name :: str :: _) = List.map V.to_str vs in
    let file_handle = open_out_gen
      [Open_wronly
       ;Open_append
       ;Open_creat
       ;Open_text
      ] 0o666 file_name in
    Printf.fprintf file_handle "%s" str;
    close_out file_handle;
    top_handle (k V.unit_value)
  )
```

This talk — a principled (co)algebraic approach!

Another issue — **linearity** or lack thereof

Another issue — **linearity** or lack thereof

- ```
let f (s:string) =
 let fh = fopen "foo.txt" in
 fwrite fh (s^s);
 fclose fh;
 return fh

let g s =
 let fh = f s in fread fh
```

## Another issue — **linearity** or lack thereof

- ```
let f (s:string) =  
  let fh = fopen "foo.txt" in  
  fwrite fh (s^s);  
  fclose fh;  
  return fh  
  
let g s =  
  let fh = f s in fread fh  (* fh not open ! *)
```

Another issue — **linearity** or lack thereof

- ```
let f (s:string) =
 let fh = fopen "foo.txt" in
 fwrite fh (s^s);
 fclose fh;
 return fh

let g s =
 let fh = f s in fread fh (* fh not open ! *)
```

- Even worse when we wrap `f` in a **handler**?

```
let h = handler
 | effect (FWrite fh s k) → return ()

let g' s =
 with h handle f ()
```

## Another issue — **linearity** or lack thereof

- ```
let f (s:string) =  
  let fh = fopen "foo.txt" in  
  fwrite fh (s^s);  
  fclose fh;  
  return fh  
  
let g s =  
  let fh = f s in fread fh    (* fh not open ! *)
```

- Even worse when we wrap `f` in a **handler**?

```
let h = handler  
  | effect (FWrite fh s k) → return ()  
  
let g' s =  
  with h handle f ()          (* dangling fh ! *)
```


So, how could we solve these issues?

So, how could we solve these issues?

- We could try using **existing PL techniques**, e.g.,

- **Modules** and **abstraction**, e.g., `System.IO`

```
type IO a
```

```
hClose :: Handle → IO ()
```

- **Linear** (and **non-linear**) **types** and **effects**

```
linear type fhandle
```

```
effect FClose : (linear fhandle) → unit
```

```
linear effect FClose : fhandle → unit
```

- Handlers with **finally clauses**

So, how could we solve these issues?

- We could try using **existing PL techniques**, e.g.,

- **Modules** and **abstraction**, e.g., `System.IO`

```
type IO a
```

```
hClose :: Handle → IO ()
```

- **Linear** (and **non-linear**) **types** and **effects**

```
linear type fhandle
```

```
effect FClose : (linear fhandle) → unit
```

```
linear effect FClose : fhandle → unit
```

- Handlers with **finally clauses**
- **Problem:** They don't really capture the **essence of the problem**

So, what is that **essence** then?

So, what is that **essence** then?

- Let's look at HASKELL's **IO monad** again

So, what is that **essence** then?

- Let's look at HASKELL's **IO monad** again
- A common explanation is to think of functions

$$a \rightarrow \text{IO } b$$

as

$$a \rightarrow (\text{RealWorld} \rightarrow (b, \text{RealWorld}))$$

which is the same as

$$(a, \text{RealWorld}) \rightarrow (b, \text{RealWorld})$$

So, what is that **essence** then?

- Let's look at `HASKELL`'s **IO monad** again
- A common explanation is to think of functions

$$a \rightarrow \text{IO } b$$

as

$$a \rightarrow (\text{RealWorld} \rightarrow (b, \text{RealWorld}))$$

which is the same as

$$(a, \text{RealWorld}) \rightarrow (b, \text{RealWorld})$$

- With the `System.IO` **module abstraction** ensuring that
 - We **cannot get our hands on** `RealWorld` (no `get` and `put`)
 - We have the impression of `RealWorld` **used linearly**
 - We **don't ask more** from `RealWorld` than it can provide

So, what is that **essence** then?

- Let's look at `HASKELL`'s **IO monad** again
- A common explanation is to think of functions

$$a \rightarrow \text{IO } b$$

as

$$a \rightarrow (\text{RealWorld} \rightarrow (b, \text{RealWorld}))$$

which is the same as

$$(a, \text{RealWorld}) \rightarrow (b, \text{RealWorld})$$

But wait a minute! **RealWorld** looks a lot like a **comodel**!

`hGetLine` : $(\text{Handle}, \text{RealWorld}) \rightarrow (\text{String}, \text{RealWorld})$

`hClose` : $(\text{Handle}, \text{RealWorld}) \rightarrow ((), \text{RealWorld})$

Important: co-operations (`hClose`) make a **promise to return**!

Refresher: what's **comodel**?

Refresher: what's **comodel**?

- A **signature** Σ is a set of operation symbols $\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$

Refresher: what's **comodel**?

- A **signature** Σ is a set of operation symbols $\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$
- A **model/algebra/handler** \mathcal{M} of Σ is given by

$$\mathcal{M} = \langle M : \text{Set} , \{ \text{op}_{\mathcal{M}} : A_{\text{op}} \times M^{B_{\text{op}}} \longrightarrow M \}_{\text{op} \in \Sigma} \rangle$$

Refresher: what's **comodel**?

- A **signature** Σ is a set of operation symbols $\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$
- A **model/algebra/handler** \mathcal{M} of Σ is given by

$$\mathcal{M} = \langle M : \text{Set} , \{ \text{op}_{\mathcal{M}} : A_{\text{op}} \times M^{B_{\text{op}}} \longrightarrow M \}_{\text{op} \in \Sigma} \rangle$$

- A **comodel/coalgebra/cohandler** \mathcal{W} of Σ is given by

$$\mathcal{W} = \langle W : \text{Set} , \{ \overline{\text{op}}_{\mathcal{W}} : A_{\text{op}} \times W \longrightarrow B_{\text{op}} \times W \}_{\text{op} \in \Sigma} \rangle$$

- Intutively, comodels describe **evolution of the world** W

Refresher: what's **comodel**?

- A **signature** Σ is a set of operation symbols $\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$
- A **model/algebra/handler** \mathcal{M} of Σ is given by

$$\mathcal{M} = \langle M : \text{Set} , \{ \text{op}_{\mathcal{M}} : A_{\text{op}} \times M^{B_{\text{op}}} \longrightarrow M \}_{\text{op} \in \Sigma} \rangle$$

- A **comodel/coalgebra/cohandler** \mathcal{W} of Σ is given by

$$\mathcal{W} = \langle W : \text{Set} , \{ \overline{\text{op}}_{\mathcal{W}} : A_{\text{op}} \times W \longrightarrow B_{\text{op}} \times W \}_{\text{op} \in \Sigma} \rangle$$

- Intuitively, comodels describe **evolution of the world W**
 - Operational semantics using a tensor of a model and a comodel (Plotkin & Power, Abou-Saleh & Pattinson)
 - Stateful runners of effectful programs (Uustalu)
 - Linear state-passing translation (Møgelberg and Staton)
 - Top-level behaviour of alg. effects in EFF v2 (Bauer & Pretnar)

Towards a general programming abstraction

Towards a general programming abstraction

- ```
let f (s:string) =
 using IO cohandle
 let fh = fopen "foo.txt" in
 fwrite fh (s^s);
 fclose fh
```

 (\* in IO \*)

Now **external world** explicit, but **dangling** `fh` etc **still possible**

# Towards a general programming abstraction

- ```
let f (s:string) =  
    using IO cohandle  
        let fh = fopen "foo.txt" in  
        fwrite fh (s^s);  
        fclose fh
```

 (* in IO *)

Now **external world** explicit, but **dangling** `fh` etc **still possible**

- ```
let f (s:string) =
 using IO cohandle
 let fh = fopen "foo.txt" in
 fwrite fh (s^s)
 finally (fclose fh)
```

 (\* in IO \*)

Better, but **have to explicitly open and thread through** `fh`



# Towards a general programming abstraction

- ```
let f (s:string) =  
    using IO cohandle  
        let fh = fopen "foo.txt" in  
        fwrite fh (s^s);  
        fclose fh
```

 (* in IO *)

Now **external world** explicit, but **dangling** `fh` etc **still possible**

- ```
let f (s:string) =
 using IO cohandle
 let fh = fopen "foo.txt" in
 fwrite fh (s^s)
 finally (fclose fh)
```

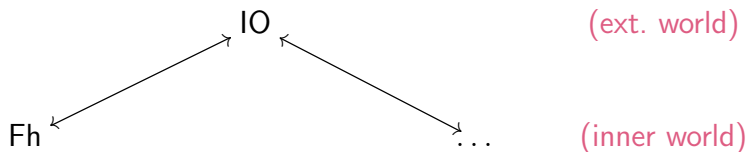
 (\* in IO \*)

Better, but **have to explicitly open and thread through** `fh`

- **Solution:** **Modular treatment** of **external worlds**

# Modular treatment of external worlds

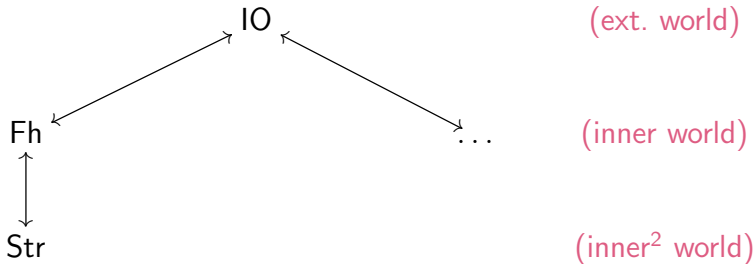
- For example



- Fh — “**world** which consists of **exactly one** fh”
- IO  $\longrightarrow$  Fh — “call `fopen` with `foo.txt`, store returned `fh`”
- Fh  $\longrightarrow$  IO — “call `fclose` with stored `fh`”

# Modular treatment of external worlds

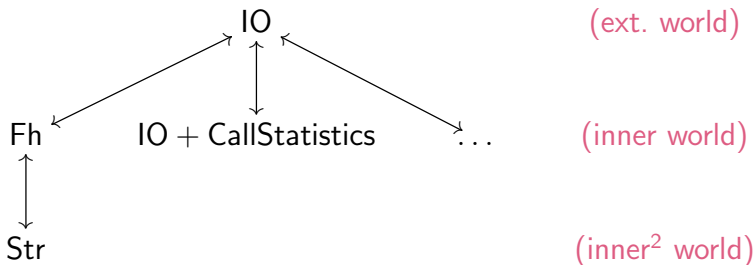
- For example



- Fh — “**world** which consists of **exactly one** fh”
- IO → Fh — “call `fopen` with `foo.txt`, store returned fh”
- Fh → IO — “call `fclose` with stored fh”
- Str — “world that is **blissfully unaware** of fh”

# Modular treatment of external worlds

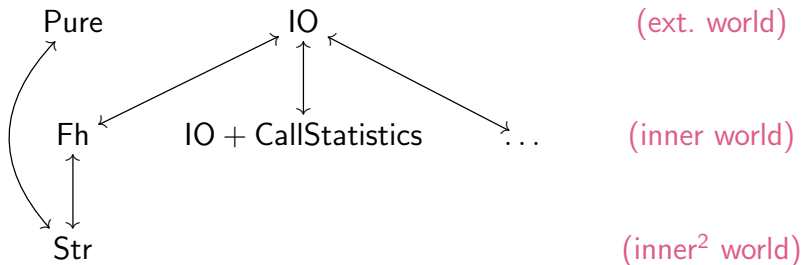
- For example



- Fh** — “**world** which consists of **exactly one** **fh**”
- IO**  $\longrightarrow$  **Fh** — “call `fopen` with `foo.txt`, store returned `fh`”
- Fh**  $\longrightarrow$  **IO** — “call `fclose` with stored `fh`”
- Str** — “world that is **blissfully unaware** of `fh`”

# Modular treatment of external worlds

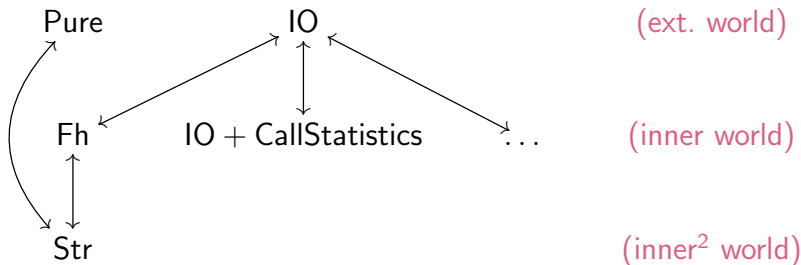
- For example



- Fh** — “**world** which consists of **exactly one** **fh**”
- IO**  $\longrightarrow$  **Fh** — “call `fopen` with `foo.txt`, store returned `fh`”
- Fh**  $\longrightarrow$  **IO** — “call `fclose` with stored `fh`”
- Str** — “world that is **blissfully unaware** of `fh`”

# Modular treatment of external worlds

- For example



- Fh** — “**world** which consists of **exactly one** **fh**”
- IO**  $\longrightarrow$  **Fh** — “call `fopen` with `foo.txt`, store returned `fh`”
- Fh**  $\longrightarrow$  **IO** — “call `fclose` with stored `fh`”
- Str** — “world that is **blissfully unaware** of `fh`”
- Observation:** **IO**  $\longleftrightarrow$  **Fh** and other  $\longleftrightarrow$  look a lot like **lenses**

**Comodels** as a gateway to the **external world**

## Comodels as a gateway to the external world

```
let f (s:string) =
 using
 Fh @ (fopen_of_io "foo.txt")
 cohandle
 fwrite_of_fh (s^s)
 finally
 x @ fh → fclose_of_io fh
```



## Comodels as a gateway to the external world

```
let f (s:string) = (* in IO *)
 using
 Fh @ (fopen_of_io "foo.txt") (* in IO *)
 cohandle
 fwrite_of_fh (s^s) (* in Fh *)
 finally
 x @ fh → fclose_of_io fh (* in IO *)
```

## Comodels as a gateway to the external world

```
let f (s:string) = (* in IO *)
 using
 Fh @ (fopen_of_io "foo.txt") (* in IO *)
 cohandle
 fwrite_of_fh (s^s) (* in Fh *)
 finally
 x @ fh → fclose_of_io fh (* in IO *)
```

where

```
Fh = (* W = fhandle *)
{ co_fread _ @ fh → ...,
 co_fwrite s @ fh → fwrite_of_io s fh;
 return ((),fh) }

(* co_fread : (unit * W) → (string * W) *)
(* co_fwrite : (string * W) → (unit * W) *)
```

**Modular treatment of worlds** ( $IO \longleftrightarrow Fh \longleftrightarrow Str$ )

## Modular treatment of worlds ( $\text{IO} \longleftrightarrow \text{Fh} \longleftrightarrow \text{Str}$ )

```
let f (s:string) = (* in IO *)
 using Fh @ (fopen_of_io "foo.txt")
 cohandle

 using Str @ (fread_of_fh ()) (* in Fh *)
 cohandle
 write_of_str (s^s) (* in Str *)
 finally
 _ @ s → fwrite_of_fh s

 finally
 _ @ fh → fclose_of_io fh
```

where

```
Str = { co_write s @ s' → (* W = string *)
 return ((), s'^s) }
```

**Tracking** the **external world** usage ( $\text{IO} \longleftrightarrow \text{CallStats}$ )

## Tracking the **external world** usage ( $\text{IO} \longleftrightarrow \text{CallStats}$ )

```
let f (s:string) = (* in IO *)
 using CallStats @ (fopen_of_io "foo.txt")
 cohandle
 fwrite_of_callstats (s^s)
 finally
 - @ (fh,c) →
 let fh' = fopen_of_io "stats.txt" in
 fwrite_of_io fh' c; fclose_of_io fh';
 fclose_of_io fh
```

where

```
CallStats = (* W = fhandle * nat *)
{ co_fread - @ (fh,c) → ... ,
 co_fwrite s @ (fh,c) → ... ,
 co_reset - @ (fh,c) → return ((),(fh,0)) }
```

## Tracking the **external world** usage ( $\text{IO} \longleftrightarrow \text{CallStats}$ )

```
let f (s:string) = (* in IO *)
 using CallStats @ (fopen_of_io "foo.txt")
 cohandle
 fwrite_of_callstats (s^s)
 finally
 _ @ (fh,c) →
 let fh' = fopen_of_io "stats.txt" in
 fwrite_of_io fh' c; fclose_of_io fh';
 fclose_of_io fh
```

where

```
CallStats = (* W = fhandle * nat *)
{ co_fread _ @ (fh,c) → ... ,
 co_fwrite s @ (fh,c) → ... ,
 co_reset _ @ (fh,c) → return ((),(fh,0)) }
```

- Can also track **nondet./prob. choice results**, etc

The **external world** can also be **pure** (Pure  $\longleftrightarrow$  Str)



The **external world** can also be **pure** ( $\text{Pure} \longleftrightarrow \text{Str}$ )

```
let f (s:string) = (* in Pure *)
 using
 Str @ (return "default value")
 cohandle
 ...
 if (read_of_str () == "foo")
 then (...; write_of_str "bar"; ...)
 else (...)
 ...
 finally
 x @ s → return x
```

where

```
Str = (* W = string *)
{ co_read _ @ s → return (s,s) ,
 co_write s @ s' → return ((),s') }
```

So what's happening **more formally?**

# So what's happening **more formally?**

- Core calculus for cohandlers (wo/ handlers  $\Rightarrow$  wait a few slides)

# So what's happening **more formally?**

- Core calculus for cohandlers (wo/ handlers  $\Rightarrow$  wait a few slides)
- **Types**

$$A, B, W ::= b \mid 1 \mid A \times B \mid 0 \mid A + B \mid A \overset{\omega}{\rightarrow} B$$

# So what's happening **more formally?**

- Core calculus for cohandlers (wo/ handlers  $\Rightarrow$  wait a few slides)
- **Types**

$$A, B, W ::= b \mid 1 \mid A \times B \mid 0 \mid A + B \mid A \xrightarrow{\omega} B$$

- **Signatures of (external) worlds**

$$\omega ::= \{ \text{op}_1 : A_1 \rightsquigarrow B_1, \dots, \text{op}_n : A_n \rightsquigarrow B_n \}$$

# So what's happening **more formally**?

- Core calculus for cohandlers (wo/ handlers  $\Rightarrow$  wait a few slides)
- **Types**

$$A, B, W ::= b \mid 1 \mid A \times B \mid 0 \mid A + B \mid A \xrightarrow{\omega} B$$

- **Signatures of (external) worlds**

$$\omega ::= \{ \text{op}_1 : A_1 \rightsquigarrow B_1, \dots, \text{op}_n : A_n \rightsquigarrow B_n \}$$

- **Computation terms** (value terms are unsurprising)

$$\begin{array}{l} c ::= \text{return } v \mid \text{let } x = c_1 \text{ in } c_2 \mid v_1 v_2 \\ \quad \mid \widehat{\text{op}} \ v \quad \quad \quad \text{(comodel op.)} \\ \quad \mid \text{using } C @ c_i \text{ cohandle } c \text{ finally } x @ w \rightarrow c_f \quad \text{(cohandling)} \end{array}$$

# So what's happening **more formally?**

- Core calculus for cohandlers (wo/ handlers  $\Rightarrow$  wait a few slides)
- **Types**

$$A, B, W ::= b \mid 1 \mid A \times B \mid 0 \mid A + B \mid A \xrightarrow{\omega} B$$

- **Signatures of (external) worlds**

$$\omega ::= \{ \text{op}_1 : A_1 \rightsquigarrow B_1, \dots, \text{op}_n : A_n \rightsquigarrow B_n \}$$

- **Computation terms** (value terms are unsurprising)

$$c ::= \text{return } v \mid \text{let } x = c_1 \text{ in } c_2 \mid v_1 v_2$$
$$\begin{array}{l} | \widehat{\text{op}} \ v \\ | \text{using } C @ c_i \text{ cohandle } c \text{ finally } x @ w \rightarrow c_f \end{array} \quad \begin{array}{l} \text{(comodel op.)} \\ \text{(cohandling)} \end{array}$$

- **Comodels (cohandlers)**

$$C ::= \{ \overline{\text{op}}_1 x @ w \rightarrow c_1, \dots, \overline{\text{op}}_n x @ w \rightarrow c_n \}$$

So what's happening **more formally**?



# So what's happening **more formally**?

- **Typing judgements**

$$\Gamma \vdash v : A$$
$$\Gamma \Vdash c : A$$

# So what's happening **more formally?**

- **Typing judgements**

$$\Gamma \vdash v : A \qquad \Gamma \Vdash c : A$$

- The two central **typing rules** are

$$\begin{array}{c} \Gamma \Vdash D \text{ comodel of } \omega' \text{ with carrier } W_D \qquad \Gamma \Vdash c_i : W_D \\ \Gamma \Vdash' c : A \qquad \Gamma, x:A, w:W_D \Vdash c_f : B \\ \hline \Gamma \Vdash \text{using } D @ c_i \text{ cohandle } c \text{ finally } x @ w \rightarrow c_f : B \end{array}$$

and

$$\frac{\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}} \in \omega \qquad \Gamma \vdash v : A_{\text{op}}}{\Gamma \Vdash \hat{\text{op}} v : B_{\text{op}}}$$

So what's happening **more formally?**

# So what's happening **more formally**?

- **Denotational semantics** is heavily inspired by  
Møgelberg and Staton's **linear state-passing translation**

# So what's happening **more formally**?

- **Denotational semantics** is heavily inspired by Møgelberg and Staton's **linear state-passing translation**
- **Term interpretation** looks very similar to **alg. effects**:

$$\llbracket \Gamma \vdash v : A \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket \qquad \llbracket \Gamma \stackrel{\omega}{\vdash} c : A \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow T_{\omega} \llbracket A \rrbracket$$

- **un-cohandled operations wait for a suitable external world!**

# So what's happening **more formally**?

- **Denotational semantics** is heavily inspired by Møgelberg and Staton's **linear state-passing translation**
- **Term interpretation** looks very similar to **alg. effects**:

$$\llbracket \Gamma \vdash v : A \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket \qquad \llbracket \Gamma \Vdash c : A \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow T_{\omega} \llbracket A \rrbracket$$

- **un-cohandled operations wait for a suitable external world!**
- The interesting part is the interpretation of **cohandling**

$$\Gamma \Vdash \text{using } D @ c_i \text{ cohandle } c \text{ finally } x @ w \rightarrow c_f : B$$

which is based on the **linear state-passing translation**, i.e.,

$$\frac{\llbracket D \rrbracket \in \text{Comod}_{\omega'}(\text{Kleisli}(T_{\omega}))}{\text{cohandle\_with}_{\llbracket D \rrbracket} : T_{\omega'} \llbracket A \rrbracket \longrightarrow \left( \llbracket W_D \rrbracket \rightarrow T_{\omega} (\llbracket A \rrbracket \times \llbracket W_D \rrbracket) \right)}$$

So what's happening **more formally**?

## So what's happening **more formally**?

- Regarding **op. semantics**, e.g., consider confs.  $(\overrightarrow{(C, w)}, c)$



## So what's happening **more formally**?

- Regarding **op. semantics**, e.g., consider confs.  $(\overrightarrow{(C, w)}, c)$
- For example, consider the **big-step evaluation** of **using** D ...

# So what's happening **more formally**?

- Regarding **op. semantics**, e.g., consider confs.  $( (\overrightarrow{C, w}), c )$
- For example, consider the **big-step evaluation** of **using D ...**

$$( (\overrightarrow{C, w_0}), (C', w'_0) ) , c_i ) \Downarrow ( (\overrightarrow{C, w_1}), (C', w'_1) ) , \text{return } w''_0 )$$

$$( (\overrightarrow{C, w_1}), (C', w'_1), (D, w''_0) ) , c ) \Downarrow ( (\overrightarrow{C, w_2}), (C', w'_2), (D, w''_1) ) , \text{return } v )$$

$$( (\overrightarrow{C, w_2}), (C', w'_2) ) , c_f[v/x, w''_1/w] ) \Downarrow ( (\overrightarrow{C, w_3}), (C', w'_3) ) , \text{return } v' )$$

---

$$( (\overrightarrow{C, w_0}), (C', w'_0) ) , \text{using D @ } c_i \text{ cohandle } c \text{ finally } x @ w \rightarrow c_f )$$

$$\Downarrow$$

$$( (\overrightarrow{C, w_3}), (C', w'_3) ) , \text{return } v' )$$

# So what's happening **more formally**?

- Regarding **op. semantics**, e.g., consider confs.  $( (\overrightarrow{C, w}), c )$
- For example, consider the **big-step evaluation** of **using D ...**

$$( (\overrightarrow{C, w_0}), (C', w'_0) ) , c_i ) \Downarrow ( (\overrightarrow{C, w_1}), (C', w'_1) ) , \text{return } w''_0 )$$

$$( (\overrightarrow{C, w_1}), (C', w'_1), (D, w''_0) ) , c ) \Downarrow ( (\overrightarrow{C, w_2}), (C', w'_2), (D, w''_1) ) , \text{return } v )$$

$$( (\overrightarrow{C, w_2}), (C', w'_2) ) , c_f[v/x, w''_1/w] ) \Downarrow ( (\overrightarrow{C, w_3}), (C', w'_3) ) , \text{return } v' )$$

---

$$( (\overrightarrow{C, w_0}), (C', w'_0) ) , \text{using D @ } c_i \text{ cohandle } c \text{ finally } x @ w \rightarrow c_f )$$

$$\Downarrow$$

$$( (\overrightarrow{C, w_3}), (C', w'_3) ) , \text{return } v' )$$

- The interpretation of **operations** uses the **co-operations** of Cs

But what about **alg. effects** and **handlers**?

## But what about **alg. effects** and **handlers**?

- **First:** combining this with **standard alg. effects** and **handlers**

# But what about **alg. effects** and **handlers**?

- **First:** combining this with **standard alg. effects** and **handlers**
- In the following

```
using C @ c_i
cohandle c
finally x @ w \rightarrow c_f
```

it is natural to want that

- **algebraic operations** (in the sense of  $E_{FF}$ ) are allowed in `c`,  
but they must not be allowed to escape `cohandle`
- to escape, have to use the **co-operations** of the **external world**

# But what about **alg. effects** and **handlers**?

- **First:** combining this with **standard alg. effects** and **handlers**
- In the following

```
using C @ c_i
cohandle c
finally x @ w \rightarrow c_f
```

it is natural to want that

- **algebraic operations** (in the sense of  $E_{FF}$ ) are allowed in `c`,  
but they must not be allowed to escape **cohandle**
- to escape, have to use the **co-operations** of the **external world**
- the **continuations of handlers** in `c` are delimited by **cohandle**

# But what about **alg. effects** and **handlers**?

- **First:** combining this with **standard alg. effects** and **handlers**
- In the following

```
using C @ c_i
cohandle c
finally x @ w \rightarrow c_f
```

it is natural to want that

- **algebraic operations** (in the sense of  $E_{FF}$ ) are allowed in `c`,  
but they must not be allowed to escape `cohandle`
- to escape, have to use the **co-operations** of the **external world**
- the **continuations of handlers** in `c` are delimited by `cohandle`
- Where do **multi-handlers** fit? Co-operating handlers-cohandlers?



But what about **alg. effects** and **handlers**?

# But what about **alg. effects** and **handlers**?

- **Second:** What if the **outer comodel beaks its promise**?
  - E.g., **IO** lost connection to the HDD where “foo.txt” was

# But what about **alg. effects** and **handlers**?

- **Second:** What if the **outer comodel beaks its promise**?
  - E.g., **IO** lost connection to the HDD where “foo.txt” was
- **Idea:**
  - Use algebraic effects to **communicate downwards**
  - (Algebraic ops. only allowed to appear in co-operations)
  - **finally** acts as a **handler** for **broken promises**

# But what about **alg. effects** and **handlers**?

- **Second:** What if the **outer comodel beaks its promise?**
  - E.g., **IO** lost connection to the HDD where “foo.txt” was
- **Idea:**
  - Use algebraic effects to **communicate downwards**
  - (Algebraic ops. only allowed to appear in co-operations)
  - **finally** acts as a **handler** for **broken promises**

```
using (* IO \longleftrightarrow Fh *)
 Fh @ c_i
cohandle
 fwrite_of_d s; (* co_fwrite_of_io throws e *)
 fread ()
finally
 | x @ w \rightarrow c_f
 | throw e \rightarrow c_do_some_cleanup
 | op x k \rightarrow ...
```

# Conclusions

# Conclusions

- **Comodels** as a gateway for interacting with the **external world**
- “**Linearity**” by leaving **outer worlds** implicit (via comodel ops.)
- `System.IO`, KOKA’s `initially` & `finally`, PYTHON’s `with`, ...
- Could also be convenient for **general FFI**

$$\frac{f : A \longrightarrow B \in \text{OCAML}}{\bar{f} : A \times W_{\text{OCaml}} \longrightarrow B \times W_{\text{OCaml}} \in \text{OCaml}} \text{ (FFI)}$$

# Conclusions

- **Comodels** as a gateway for interacting with the **external world**
- “**Linearity**” by leaving **outer worlds** implicit (via comodel ops.)
- `System.IO`, KOKA’s `initially` & `finally`, PYTHON’s `with`, ...
- Could also be convenient for **general FFI**

$$\frac{f : A \longrightarrow B \in \text{OCAML}}{\overline{f} : A \times W_{\text{OCaml}} \longrightarrow B \times W_{\text{OCaml}} \in \text{OCaml}} \text{ (FFI)}$$

## Some ongoing work

- Interaction with **algebraic effects** and **(multi-)handlers**
- Clarify the connection with **(effectful) lenses**
- **Combinatorics** of comodels and their lens-like relationships