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**, e.g.,

```
(* System.IO *)

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

```
(* pervasives.eff *)

effect RandomInt   : int → int
effect RandomFloat : float → float
```

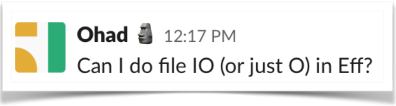
- And then **treat them specially** in the compiler, e.g.,

```
(* src/runtime/eval.ml *)

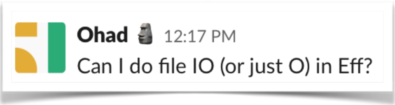
let rec top_handle op =
  match op with
  | ...
```

External world in FP

External world in FP



External world in FP



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

  $$a \to \mathrm{IO}\ b$$

  as

  $$a \to (\mathrm{RealWorld} \to (b, \mathrm{RealWorld}))$$

  which is the same as

  $$(a, \mathrm{RealWorld}) \to (b, \mathrm{RealWorld})$$

So, what is that essence then?

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

$$a \to \mathsf{IO}\ b$$

as

$$a \to (\mathsf{RealWorld} \to (b, \mathsf{RealWorld}))$$

which is the same as

$$(a, \mathsf{RealWorld}) \to (b, \mathsf{RealWorld})$$

- 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

$$\mathsf{a} \to \mathsf{IO\ b}$$

as

$$\mathsf{a} \to (\mathsf{RealWorld} \to (\mathsf{b}, \mathsf{RealWorld}))$$

which is the same as

$$(\mathsf{a}, \mathsf{RealWorld}) \to (\mathsf{b}, \mathsf{RealWorld})$$

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

$$\mathsf{hGetLine} : (\mathsf{Handle}, \mathsf{RealWorld}) \to (\mathsf{String}, \mathsf{RealWorld})$$

$$\mathsf{hClose} \quad : (\mathsf{Handle}, \mathsf{RealWorld}) \to ((), \mathsf{RealWorld})$$

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 $\mathsf{op} : A_{\mathsf{op}} \rightsquigarrow B_{\mathsf{op}}$

Refresher: what's **comodel**?

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

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

Refresher: what's comodel?

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

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

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

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

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

Refresher: what's comodel?

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

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

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

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

- Intutively, 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)                        (* in IO *)
    finally (fclose fh)
  ```

  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**

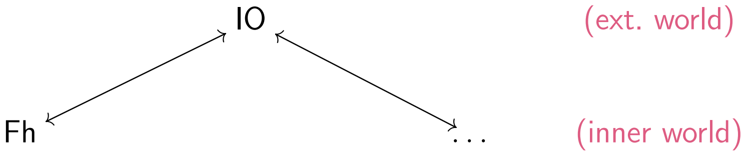
- ```
  let f (s:string) =
    using IO cohandle
      let fh = fopen "foo.txt" in
      fwrite fh (s^s)                             (* in IO *)
    finally (fclose fh)
  ```

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

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

Modular treatment of external worlds

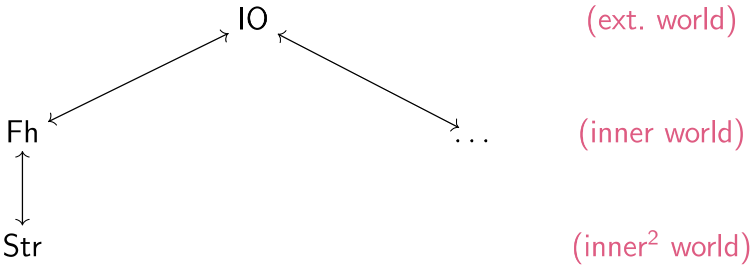
- For example



- 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

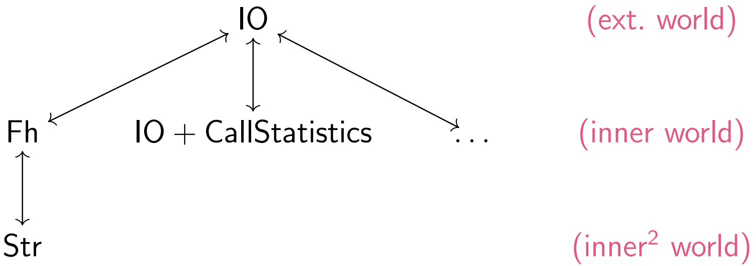
- For example



- Fh — "**world** which consists of **exactly one** `fh`"
- IO $\longrightarrow$ 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

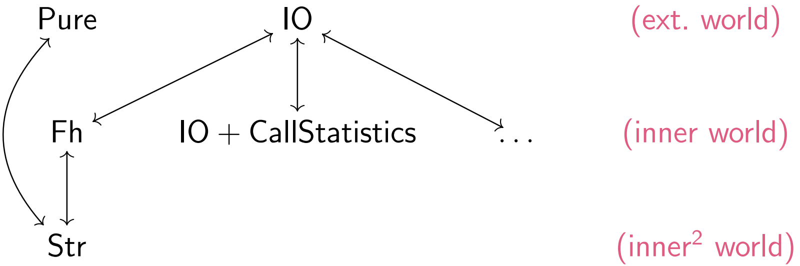
- For example



- 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

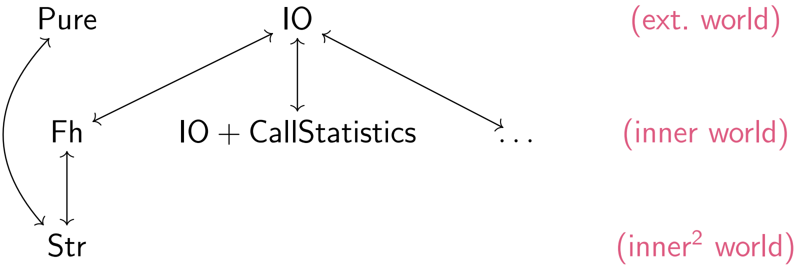
- For example



- Fh — "**world** which consists of **exactly one** `fh`"
- IO ⟶ 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`"
- **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 (IO $\longleftrightarrow$ Fh $\longleftrightarrow$ 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 (IO $\longleftrightarrow$ Stats)

Tracking the external world usage (IO ⟷ Stats)

```
let f (s:string) =                          (* in IO *)
  using
    Stats @ (let fh = fopen_of_io "foo.txt" in
              return (fh,0))
  cohandle
    fwrite_of_stats (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

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

Tracking the external world usage (IO ⟷ Stats)

```
let f (s:string) =                              (* in IO *)
  using
    Stats @ (let fh = fopen_of_io "foo.txt" in
                return (fh,0))
  cohandle
    fwrite_of_stats (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

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

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

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

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

```
let f (s:string) =                    (* in Pure *)
  using
    Str @ (return "default value")
  cohandle
    ...
    let s = read_of_str () in
    if (s == "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 ⇒ wait a few slides)
- **Types**

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

So what's happening more formally?

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

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

- **Signatures** of (**external**) **worlds**

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

So what's happening more formally?

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

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

- **Signatures** of (**external**) **worlds**

$$\omega \;::=\; \{\, \mathsf{op}_1 : A_1 \leadsto B_1 \,,\; \ldots \,,\; \mathsf{op}_n : A_n \leadsto B_n \,\}$$

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

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

So what's happening more formally?

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

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

- **Signatures** of (**external**) **worlds**

$$\omega \;::=\; \{\, \mathsf{op}_1 : A_1 \leadsto B_1 \,,\; \ldots \,,\; \mathsf{op}_n : A_n \leadsto B_n \,\}$$

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

$$\begin{array}{lll} c & ::= & \mathbf{return}\ v \mid \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \mid v_1 v_2 \\ & \mid & \widehat{\mathsf{op}}\ v \qquad\qquad \text{(comodel op.)} \\ & \mid & \mathbf{using}\ C \mathbin{@} c_i\ \mathbf{cohandle}\ c\ \mathbf{finally}\ x \mathbin{@} w \to c_f \qquad \text{(cohandling)} \end{array}$$

- **Comodels** (**cohandlers**)

$$C \;::=\; \{\, \overline{\mathsf{op}}_1\ x \mathbin{@} w \to c_1 \,,\; \ldots \,,\; \overline{\mathsf{op}}_n\ x \mathbin{@} w \to c_n \,\}$$

So what's happening more formally?

So what's happening more formally?

- **Typing judgements**

$$\Gamma \vdash v : A \qquad\qquad \Gamma \overset{\omega}{\vDash} c : A$$

So what's happening more formally?

- **Typing judgements**

$$\Gamma \vdash v : A \qquad \Gamma \vDash^{\omega} c : A$$

- The two central **typing rules** are

$$\frac{\Gamma \vDash \mathsf{D} \text{ comodel of } \omega' \text{ with carrier } W_{\mathsf{D}} \qquad \Gamma \vDash^{\omega} c_i : W_{\mathsf{D}} \qquad \Gamma \vDash^{\omega'} c : A \qquad \Gamma, x : A, w : W_{\mathsf{D}} \vDash^{\omega} c_f : B}{\Gamma \vDash^{\omega} \textbf{using } \mathsf{D} \mathbin{@} c_i \textbf{ cohandle } c \textbf{ finally } x \mathbin{@} w \to c_f : B}$$

and

$$\frac{\mathsf{op} : A_{\mathsf{op}} \rightsquigarrow B_{\mathsf{op}} \in \omega \qquad \Gamma \vdash v : A_{\mathsf{op}}}{\Gamma \vDash^{\omega} \widehat{\mathsf{op}}\ v : B_{\mathsf{op}}}$$

Denotational semantics

Denotational semantics

- **Term interpretation** looks very similar to **alg. effects**:

$$[\![\Gamma \vdash v : A]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!] \qquad [\![\Gamma \vDash^{\omega} c : A]\!] : [\![\Gamma]\!] \longrightarrow T_{\omega}\, [\![A]\!]$$

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

Denotational semantics

- **Term interpretation** looks very similar to **alg. effects**:

$$[\![\Gamma \vdash v : A]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!] \qquad [\![\Gamma \overset{\omega}{\vDash} c : A]\!] : [\![\Gamma]\!] \longrightarrow T_{\omega}\,[\![A]\!]$$

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

- The interesting part is the **interpretation of cohandling**

$$\frac{\Gamma \overset{\omega}{\vDash} \mathsf{D} \text{ comodel of } \omega' \text{ with carrier } W_{\mathsf{D}} \qquad \Gamma \overset{\omega}{\vDash} c_i : W_{\mathsf{D}} \qquad \Gamma \overset{\omega'}{\vDash} c : A \qquad \Gamma, x\!:\!A, w\!:\!W_{\mathsf{D}} \overset{\omega}{\vDash} c_f : B}{\Gamma \overset{\omega}{\vDash} \textbf{using } \mathsf{D} \mathbin{@} c_i \textbf{ cohandle } c \textbf{ finally } x \mathbin{@} w \to c_f : B}$$

which is based on M&S's **linear state-passing translation**, i.e.,

$$\frac{[\![\mathsf{D}]\!] \in \mathsf{Comod}_{\omega'}(\mathsf{Kleisli}(T_{\omega}))}{\mathsf{cohandle\_with}_{[\![\mathsf{D}]\!]} : T_{\omega'}\,[\![A]\!] \longrightarrow \Big([\![W_{\mathsf{D}}]\!] \to T_{\omega}\,([\![A]\!] \times [\![W_{\mathsf{D}}]\!])\Big)}$$

Operational semantics

Operational semantics

- Idea is to consider configurations $\left(\overrightarrow{(\mathsf{C}, w)}\ ,\ c \right)$

Operational semantics

- Idea is to consider configurations $\left(\overrightarrow{(\mathsf{C}, w)}\ ,\ c \right)$
- For example, consider the **big-step evaluation** of `using D ...`

Operational semantics

- Idea is to consider configurations $\left(\ \overrightarrow{(\mathsf{C}, w)}\ ,\ c\ \right)$
- For example, consider the **big-step evaluation** of **using** D ...

$$\left(\ (\overrightarrow{(\mathsf{C}, w_0)}, (\mathsf{C}', w_0'))\ ,\ c_i\ \right) \Downarrow \left(\ (\overrightarrow{(\mathsf{C}, w_1)}, (\mathsf{C}', w_1'))\ ,\ \textbf{return}\ w_0''\ \right)$$

$$\left(\ (\overrightarrow{(\mathsf{C}, w_1)}, (\mathsf{C}', w_1'), (\mathsf{D}, w_0''))\ ,\ c\ \right) \Downarrow \left(\ (\overrightarrow{(\mathsf{C}, w_2)}, (\mathsf{C}', w_2'), (\mathsf{D}, w_1''))\ ,\ \textbf{return}\ v\ \right)$$

$$\left(\ (\overrightarrow{(\mathsf{C}, w_2)}, (\mathsf{C}', w_2'))\ ,\ c_f[v/x, w_1''/w]\ \right) \Downarrow \left(\ (\overrightarrow{(\mathsf{C}, w_3)}, (\mathsf{C}', w_3'))\ ,\ \textbf{return}\ v'\ \right)$$

$$\left(\ (\overrightarrow{(\mathsf{C}, w_0)}, (\mathsf{C}', w_0'))\ ,\ \textbf{using}\ \mathsf{D}\ @\ c_i\ \textbf{cohandle}\ c\ \textbf{finally}\ x\ @\ w \to c_f\ \right)$$

$$\Downarrow$$

$$\left(\ (\overrightarrow{(\mathsf{C}, w_3)}, (\mathsf{C}', w_3'))\ ,\ \textbf{return}\ v'\ \right)$$

Operational semantics

- Idea is to consider configurations $\left(\ \overrightarrow{(\mathsf{C}, w)}\ ,\ c\ \right)$
- For example, consider the **big-step evaluation** of **using** D ...

$$\dfrac{\begin{array}{c}\left(\ (\overrightarrow{(\mathsf{C}, w_0)}, (\mathsf{C}', w_0'))\ ,\ c_i\ \right) \Downarrow \left(\ (\overrightarrow{(\mathsf{C}, w_1)}, (\mathsf{C}', w_1'))\ ,\ \mathbf{return}\ w_0''\ \right) \\ \left(\ (\overrightarrow{(\mathsf{C}, w_1)}, (\mathsf{C}', w_1'), (\mathsf{D}, w_0''))\ ,\ c\ \right) \Downarrow \left(\ (\overrightarrow{(\mathsf{C}, w_2)}, (\mathsf{C}', w_2'), (\mathsf{D}, w_1''))\ ,\ \mathbf{return}\ v\ \right) \\ \left(\ (\overrightarrow{(\mathsf{C}, w_2)}, (\mathsf{C}', w_2'))\ ,\ c_f[v/x, w_1''/w]\ \right) \Downarrow \left(\ (\overrightarrow{(\mathsf{C}, w_3)}, (\mathsf{C}', w_3'))\ ,\ \mathbf{return}\ v'\ \right)\end{array}}{\begin{array}{c}\left(\ (\overrightarrow{(\mathsf{C}, w_0)}, (\mathsf{C}', w_0'))\ ,\ \mathbf{using}\ \mathsf{D}\ @\ c_i\ \mathbf{cohandle}\ c\ \mathbf{finally}\ x\ @\ w \to c_f\ \right) \\ \Downarrow \\ \left(\ (\overrightarrow{(\mathsf{C}, w_3)}, (\mathsf{C}', w_3'))\ ,\ \mathbf{return}\ v'\ \right)\end{array}}$$

- 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 → c_f
```

  it is natural to want that

  - **algebraic operations** (in the sense of EFF) 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 → c_f
```

  it is natural to want that

  - **algebraic operations** (in the sense of EFF) 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 → c_f
```

  it is natural to want that

  - **algebraic operations** (in the sense of EFF) 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 ⟷ Fh *)
  Fh @ c_i
cohandle
  fwrite_of_d s;  (* co_fwrite_of_io throws e *)
  fread ()
finally
  | x @ w → c_f
  | throw e → c_do_some_cleanup
  | op x k → ...
```

Conclusions

Conclusions

- **Comodels** as a gateway for interacting with the **external world**
- `System.IO`, Koka's **initially** & **finally**, Python's **with**, ...
- Could also be convenient for **general FFI**

$$\frac{f : A \longrightarrow B \;\in\; \text{OCaml}}{\overline{f} : A \times W_{\text{OCaml}} \longrightarrow B \times W_{\text{OCaml}} \;\in\; \text{OCaml}}$$

Conclusions

- **Comodels** as a gateway for interacting with the **external world**
- `System.IO`, KOKA's `initially` & `finally`, PYTHON's `with`, ...
- Could also be convenient for **general FFI**

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

Some ongoing work

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