Comodels as a gateway for interacting with the external world

Danel Ahman

(joint work with Andrej Bauer)

University of Ljubljana, Slovenia

MSR Redmond, 3 May 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)))
```

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
```

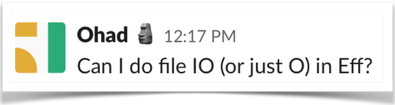
- And then **treat them specially** in the compiler, e.g.,

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

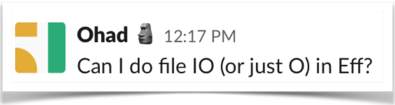
let rec top_handle op =
  match op with
  | ...
```

An issue — difficult to cover all use cases

An issue — difficult to cover all use cases



An issue — difficult to cover all use cases



An issue — difficult to cover all use cases

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)
       )
```

An issue — difficult to cover all use cases

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!

A bigger issue — linearity or lack thereof

A bigger 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
  ```
  ```

A bigger 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 ! *)
  ```
  ```

A bigger 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 ()
  ```
  ```

A bigger 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

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

as

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

which is the same as

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

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

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

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

  as

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

  which is the same as

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

- 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

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

as

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

which is the same as

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

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 a comodel?

Refresher: what's a **comodel**?

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

Refresher: what's a 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 a **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 worlds** $w_1, w_2, w_3, \ldots$

Refresher: what's a 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 worlds** $w_1, w_2, w_3, \ldots$
  - 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) =          (* in top level world *)
    using IO run
      let fh = fopen "foo.txt" in
      fwrite fh (s^s);
      fclose fh                     (* in IO world *)
  ```

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

Towards a general programming abstraction

- ```
  let f (s:string) =          (* in top level world *)
    using IO run
      let fh = fopen "foo.txt" in
      fwrite fh (s^s);
      fclose fh                         (* in IO world *)
  ```

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

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

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

Towards a general programming abstraction

- ```
  let f (s:string) =          (* in top level world *)
    using IO run
      let fh = fopen "foo.txt" in
      fwrite fh (s^s);
      fclose fh                       (* in IO world *)
  ```

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

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

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

- **Our solution: Modular treatment of external worlds**

Modular treatment of **external worlds**

- **For example**

$$\begin{tikzcd}
 & \mathsf{IO} \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & & \text{(ext. world)} \\
\mathsf{Fh} & & \dots & \text{(inner world)}
\end{tikzcd}$$

- 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**

$$\begin{tikzcd} & \mathsf{IO} \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & & \text{(ext. world)} \\ \mathsf{Fh} \arrow[d, leftrightarrow] & & \dots & \text{(inner world)} \\ \mathsf{Fc} & & & \text{(inner}^2\text{ world)} \end{tikzcd}$$

- 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`"
- Fc — "**world** that is **blissfully unaware** of `fh`"

Modular treatment of external worlds

- **For example**

$$\begin{tikzcd}
 & \mathsf{IO} \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & & \text{(ext. world)} \\
\mathsf{Fh} \arrow[d, leftrightarrow] & \mathsf{IO} + \mathsf{CallStatistics} & \dots & \text{(inner world)} \\
\mathsf{Fc} & & & \text{(inner}^2\text{ world)}
\end{tikzcd}$$

- 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`"
- Fc — "**world** that is **blissfully unaware** of `fh`"

Modular treatment of external worlds

- **For example**

$$
\begin{tikzcd}
\mathsf{Pure} \arrow[dd, leftrightarrow, bend right] & \mathsf{IO} \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & & \text{(ext. world)} \\
\mathsf{Fh} \arrow[d, leftrightarrow] & \mathsf{IO} + \mathsf{CallStatistics} & \dots & \text{(inner world)} \\
\mathsf{Fc} & & & \text{(inner}^2\text{ world)}
\end{tikzcd}
$$

- 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`”

- Fc — “**world** that is **blissfully unaware** of `fh`”

Modular treatment of external worlds

- **For example**

$$
\begin{tikzcd}
\mathsf{Pure} \arrow[dd, leftrightarrow, bend right] & \mathsf{IO} \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \mathsf{MLState} \arrow[dd, leftrightarrow, bend left] & \text{(ext. world)} \\
\mathsf{Fh} \arrow[d, leftrightarrow] & \mathsf{IO} + \mathsf{CallStatistics} & \ldots & \text{(inner world)} \\
\mathsf{Fc} & & \mathsf{FPState} & \text{(inner}^2\text{ world)}
\end{tikzcd}
$$

- 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`"
- Fc — "**world** that is **blissfully unaware** of `fh`"

Modular treatment of external worlds

- **For example**

$$\begin{tikzcd}
\mathsf{Pure} \arrow[dd, leftrightarrow, bend right] & \mathsf{IO} \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \mathsf{MLState} \arrow[dd, leftrightarrow, bend left] & \text{(ext. world)} \\
\mathsf{Fh} \arrow[d, leftrightarrow] & \mathsf{IO} + \mathsf{CallStatistics} & \ldots & \text{(inner world)} \\
\mathsf{Fc} & & \mathsf{FPState} & \text{(inner}^2\text{ world)}
\end{tikzcd}$$

- 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`"
- Fc — "**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

- **Running a program** on a comodel (**in external world**)

```
using
  C @ c_init
run
  c
finally @ w { return(x) ↦ c_fin(w,x) }
```

- **Comodels** are defined as follows

```
C =
  {
    op (x:A) @ (w:W) ↦ c_op(x,w),   (* : B * W *)
    ...
  }
```

for all operations op : $A \rightsquigarrow B$ in a given signature Σ

Focussing on a fragment of the external world

Focussing on a fragment of the external world

```
let f (s:string) =
  using
    Fh @ (fopen_of_io "foo.txt")
  run
    fwrite_of_fh (s^s)
  finally @ fh {
    return(x) ↦ fclose_of_io fh }
```

Focussing on a fragment of the external world

```
let f (s:string) =                              (* in IO *)
  using
    Fh @ (fopen_of_io "foo.txt")                (* in IO *)
  run
    fwrite_of_fh (s^s)                          (* in Fh *)
  finally @ fh {
    return(x) ↦ fclose_of_io fh }               (* in IO *)
```

Focussing on a fragment of the external world

```
let f (s:string) =                          (* in IO *)
  using
    Fh @ (fopen_of_io "foo.txt")            (* in IO *)
  run
    fwrite_of_fh (s^s)                      (* in Fh *)
  finally @ fh {
    return(x) ↦ fclose_of_io fh }           (* in IO *)
```

where

```
Fh =                                  (* W = fhandle *)
  { fread  _ @ fh ↦ ... ,
    fwrite s @ fh ↦ fwrite_of_io s fh;
                    return ((),fh)         }

        (* fread : (unit * W) ↦ (string * W) *)
       (* fwrite : (string * W) ↦ (unit * W) *)
```

Modular treatment of worlds (IO $\longleftrightarrow$ Fh $\longleftrightarrow$ Fc)

Modular treatment of worlds (IO $\longleftrightarrow$ Fh $\longleftrightarrow$ Fc)

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

    using Fc @ (fread_of_fh ())                       (* in Fh *)
    run
      fwrite_of_fc (s^s)                              (* in Fc *)
    finally @ s {
      return(_) ↦ fwrite_of_fh s }

  finally @ fh {
    return(_) ↦ fclose_of_io fh }
```

where

```
Fc = { fwrite s @ s' ↦ return ((),s'^s) }
```

Modular treatment of worlds ($\mathsf{IO} \longleftrightarrow \mathsf{Fh} \longleftrightarrow \mathsf{Fc}$)

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

    using Fc @ (fread_of_fh ())                 (* in Fh *)
    run
      fwrite_of_fc (s^s)                        (* in Fc *)
    finally @ s {
      return(_) ↦ fwrite_of_fh s }

  finally @ fh {
    return(_) ↦ fclose_of_io fh }
```

where

```
Fc = { fwrite s @ s' ↦ return ((),s'^s) }
```

- **More generally:** comodels allow **transactions** and **sandboxing**

Tracking the world usage (IO $\longleftrightarrow$ IO+Stats)

Tracking the world usage (IO $\longleftrightarrow$ IO+Stats)

```
let f (fh:fhandle) (s:string) =          (* in IO *)
  using
    IO+Stats @ (return 0)
  run
    fwrite_of_stats fh (s^s)     (* in IO+Stats *)
  finally @ c {
    return(_) ↦
          let fh' = fopen_of_io "stats.txt" in
          fwrite_of_io fh' c; fclose_of_io fh' }
```

where

```
IO+Stats =                                 (* W = nat*)
  { fwrite s @ c ↦ ...,
    ...,
    reset  _ @ c ↦ return ((),0) }
```

Tracking the world usage (IO $\longleftrightarrow$ IO+Stats)

```
let f (fh:fhandle) (s:string) =          (* in IO *)
  using
    IO+Stats @ (return 0)
  run
    fwrite_of_stats fh (s^s)     (* in IO+Stats *)
  finally @ c {
    return(_) ↦
          let fh' = fopen_of_io "stats.txt" in
          fwrite_of_io fh' c; fclose_of_io fh' }
```

where

```
IO+Stats =                                 (* W = nat*)
  { fwrite s @ c ↦ ...,
    ...,
    reset  _ @ c ↦ return ((),0) }
```

- **More generally:** allows to slot in **instrumentation/monitors**

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")
  run
    ...
    let s = get () in
    if (s == "foo")
    then (...; set "bar"; ...)
    else (...)
    ...
  finally @ _ {
    return(x) ↦ return x }
```

where

```
Str =                                          (* W = string *)
  { get _ @ s ↦ return (s,s) ,
    set s @ _ ↦ return ((),s) }
```

So what's happening **formally**?

So what's happening formally?

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

So what's happening **formally**?

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

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

So what's happening **formally**?

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

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

- **Interfaces** of **external worlds**

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

So what's happening formally?

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

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

- **Interfaces** of **external worlds**

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

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

$$\begin{array}{rcl} c & ::= & \mathbf{return}\ v \mid \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \mid \mathbf{let\ rec}\ f\ x = c_1\ \mathbf{in}\ c_2 \mid v_1\, v_2 \\ & \mid & \mathsf{op}\ v\ (x.c) \\ & \mid & \mathbf{using}\ C \ @\ c_i\ \mathbf{run}\ c\ \mathbf{finally}\ @\ w\ \{\, \mathbf{return}(x) \mapsto c_f \,\} \end{array}$$

So what's happening formally?

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

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

- **Interfaces** of **external worlds**

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

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

$$\begin{array}{lll} c & ::= & \mathbf{return}\ v \mid \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \mid \mathbf{let\ rec}\ f\ x = c_1\ \mathbf{in}\ c_2 \mid v_1\, v_2 \\ & \mid & \mathsf{op}\ v\ (x.c) \\ & \mid & \mathbf{using}\ C\ @\ c_i\ \mathbf{run}\ c\ \mathbf{finally}\ @\ w\ \{\ \mathbf{return}(x) \mapsto c_f\ \} \end{array}$$

- **Comodels (cohandlers)**

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

So what's happening **formally**?

So what's happening **formally**?

- **Typing judgements**

$$\Gamma \vdash v : A \qquad\qquad \Gamma \models^{\Sigma} c : A$$

So what's happening formally?

- **Typing judgements**

$$\Gamma \vdash v : A \qquad\qquad \Gamma \models^{\Sigma} c : A$$

- The two central **typing rules** are

$$\boxed{\dfrac{\begin{array}{c}\Gamma \models^{\Sigma} \mathsf{C} \text{ comodel of } \Sigma' \text{ with carrier } W_{\mathsf{C}} \\ \Gamma \models^{\Sigma} c_i : W_{\mathsf{C}} \qquad \Gamma \models^{\Sigma'} c : A \qquad \Gamma, w \!:\! W_{\mathsf{C}}, x \!:\! A \models^{\Sigma} c_f : B\end{array}}{\Gamma \models^{\Sigma} \mathbf{using}\ C\ @\ c_i\ \mathbf{run}\ c\ \mathbf{finally}\ @\ w\ \{\ \mathbf{return}(x) \mapsto c_f\ \} : B}}$$

and

$$\boxed{\dfrac{\mathsf{op} : A_{\mathsf{op}} \rightsquigarrow B_{\mathsf{op}} \in \Sigma \qquad \Gamma \vdash v : A_{\mathsf{op}} \qquad \Gamma, x \!:\! B_{\mathsf{op}} \models^{\Sigma} c : A}{\Gamma \models^{\Sigma} \mathsf{op}\ v\ (x.c) : A}}$$

(Denotational) semantics (in ω-cpos)

(Denotational) semantics (in ω-cpos)

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

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

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

(Denotational) semantics (in ω-cpos)

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

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

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

- The interesting part is the **interpretation of using ... run**

$$\frac{\begin{array}{c}\Gamma \vDash^{\Sigma} \mathsf{C} \text{ comodel of } \Sigma' \text{ with carrier } W_{\mathsf{C}} \\ \Gamma \vDash^{\Sigma} c_i : W_{\mathsf{C}} \qquad \Gamma \vDash^{\Sigma'} c : A \qquad \Gamma, w : W_{\mathsf{C}}, x : A \vDash^{\Sigma} c_f : B\end{array}}{\Gamma \vDash^{\Sigma} \textbf{using } C \text{ @ } c_i \textbf{ run } c \textbf{ finally } \text{@ } w \ \{\, \textbf{return}(x) \mapsto c_f \,\} : B}$$

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

$$\frac{[\![\mathsf{C}]\!] \in \mathrm{Comod}_{\Sigma'}(\mathrm{Kleisli}(T_{\Sigma}))}{\mathrm{run\_on}_{[\![\mathsf{C}]\!]} : T_{\Sigma'}\, [\![A]\!] \longrightarrow \Big([\![W_{\mathsf{C}}]\!] \to T_{\Sigma}([\![W_{\mathsf{C}}]\!] \times [\![A]\!])\Big)}$$

What if the world doesn't keep promises?

What if the world doesn't keep promises?

- Recall, the **semantics of our co-operations**

$$\overline{\mathsf{op}} : [\![A_{\mathsf{op}}]\!] \times [\![W]\!] \longrightarrow T_{\Sigma}([\![B_{\mathsf{op}}]\!] \times [\![W]\!])$$

  ensures that the **world always comes back with an answer**

What if the world doesn't keep promises?

- Recall, the **semantics of our co-operations**

$$\overline{\mathsf{op}} : [\![A_{\mathsf{op}}]\!] \times [\![W]\!] \longrightarrow T_{\Sigma}([\![B_{\mathsf{op}}]\!] \times [\![W]\!])$$

  ensures that the **world always comes back with an answer**

- What if **IO** lost connection to the HDD where "foo.txt" was?

What if the world doesn't keep promises?

- Recall, the **semantics of our co-operations**

$$\overline{\mathsf{op}} : [\![A_{\mathsf{op}}]\!] \times [\![W]\!] \longrightarrow T_{\Sigma}([\![B_{\mathsf{op}}]\!] \times [\![W]\!])$$

  ensures that the **world always comes back with an answer**

- What if **IO** lost connection to the HDD where "foo.txt" was?

- **Our solution (simplified):** Allow the **world** to **raise signals**

```
using C @ c_init
run c
finally @ w {
  return(x) ↦ c_fin(w,x),                    (* : B *)
  signal(s) ↦ c_sig(w,s) }                   (* : B *)
```

```
C =
  { op x @ w ↦ if b then (...) else (raise s) }
```

What if the world doesn't keep promises?

What if the world doesn't keep promises?

- When a signal `s` occurs in `run c`, control jumps to `signal(s)`

```
using C @ c_init
run c
finally @ w {
  return(x) ↦ c_fin(w,x),            (* : B *)
  signal(s) ↦ c_sig(w,s) }           (* : B *)
```

What if the world doesn't keep promises?

- When a signal `s` occurs in `run c`, control jumps to **signal**(s)

```
using C @ c_init
run c
finally @ w {
  return(x) ↦ c_fin(w,x),                    (* : B *)
  signal(s) ↦ c_sig(w,s) }                   (* : B *)
```

- To resume `run c`, our **program and/or world** have to support it

```
let rec ctr_printer i =
  using Out+Ctr @ (return i)
  run
    while(T) {let j = get_c in print j; incr_c}
  finally @ k {
    return(x) ↦ ...,
    signal(s) ↦ print "foo"; ctr_printer k }
```

What if the world doesn't keep promises?

- When a signal `s` occurs in **run** `c`, control jumps to **signal**(s)

```
using C @ c_init
run c
finally @ w {
  return(x) ↦ c_fin(w,x),                (* : B *)
  signal(s) ↦ c_sig(w,s) }               (* : B *)
```

- To resume **run** `c`, our **program** **and/or** **world** have to support it

```
let rec ctr_printer i =
  using Out+Ctr @ (return i)
  run
    while(T) {let j = get_c in print j; incr_c}
  finally @ k {
    return(x) ↦ ...,
    signal(s) ↦ print "foo"; ctr_printer k }
```

- **World-based:** could **store a trace** so as to **replay "old" co-ops**

What about **alg. effects** and **handlers**?

What about alg. effects and handlers?

- In the following

```
using C @ c_init
run c
finally @ w { return(x) ↦ c_fin(w,x) , ... }
```

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 `run`
  - to escape, have to use the **co-operations** of the external world

What about alg. effects and handlers?

- In the following

```
using C @ c_init
run c
finally @ w { return(x) ↦ c_fin(w,x) , ... }
```

  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 **run**
  - to escape, have to use the **co-operations** of the **external world**
  - the **continuations of handlers** in `c` are delimited by **run**
  - so that we ensure that **finally** block is **definitely reached**

What about alg. effects and handlers?

- In the following

```
using C @ c_init
run c
finally @ w { return(x) ↦ c_fin(w,x) , ... }
```

  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 **run**
  - to escape, have to use the **co-operations** of the **external world**
  - the **continuations of handlers** in `c` are delimited by **run**
  - so that we ensure that **finally** block is **definitely reached**

- Where do **multi-handlers** fit? Co-operating handlers-cohandlers?

Conclusions

Conclusions

- **Comodels** as a **gateway** for interacting with the **external world**
- System.IO, Koka's **initially** & **finally**, Python's **with**, ...
- Can also be a **basis for FFI**, e.g., in future Eff could have

$$\boxed{\dfrac{f : A \longrightarrow B \;\in\; \mathrm{OCaml}}{\overline{f} : A \times W_{\text{top-level}} \longrightarrow B \times W_{\text{top-level}} \;\in\; \text{top-level-comodel}}}$$

Conclusions

- **Comodels** as a **gateway** for interacting with the **external world**
- `System.IO`, Koka's **`initially`** & **`finally`**, Python's **`with`**, ...
- Can also be a **basis for FFI**, e.g., in future Eff could have

$$\boxed{\dfrac{f : A \longrightarrow B \ \in\ \text{OCaml}}{\overline{f} : A \times W_{\text{top-level}} \longrightarrow B \times W_{\text{top-level}} \ \in\ \text{top-level-comodel}}}$$

Some ongoing and future work

- A working Haskell **prototype**, in long-term add to Eff
- Exploring more **examples** and **use cases**
- **Combinatorics** of comodels and their lens-like relationships
- **Interface polymorphism**, **linear typing**, ...