# Comodels as a gateway for interacting with the external world

Danel Ahman

(joint work with Andrej Bauer)

Shonan, 26 March 2019

# Comodels as a gateway for interacting with the external world

Danel Ahman

(joint work with Andrej Bauer)


Shonan, 26 March 2019

# Computational effects in FP

# Computational effects in FP

- Using **monads** (e.g., 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 **algebraic effects** and **handlers** (e.g., as in EFF)

```
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** (e.g., 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 **algebraic effects** and **handlers** (e.g., as in EFF)

```
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 ())
```

- Works well for effects that can be **represented** as pure data!

  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
```

# 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 ()
```

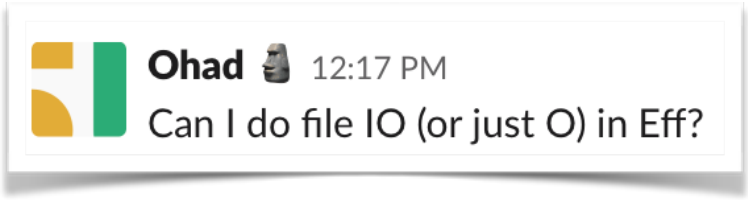
```
effect Read   : string
effect Raise  : string → empty

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

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

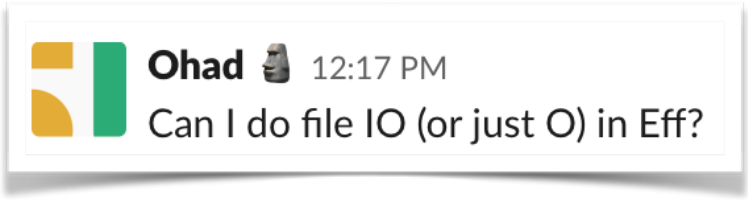
```
let rec top_handle op =
  match op with
  | ...
```

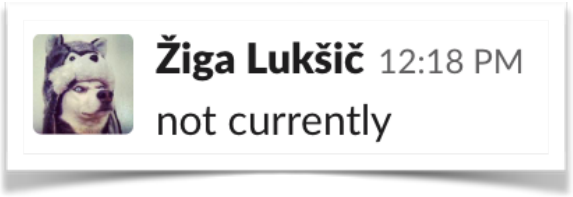
# External world in FP

# External world in FP

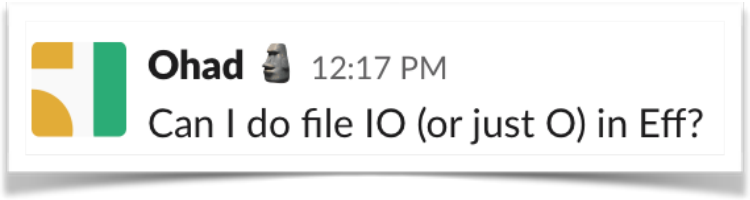


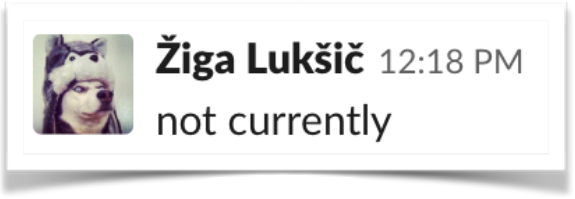
# External world in FP



# External world in FP




**Ohad** 8:35 PM

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

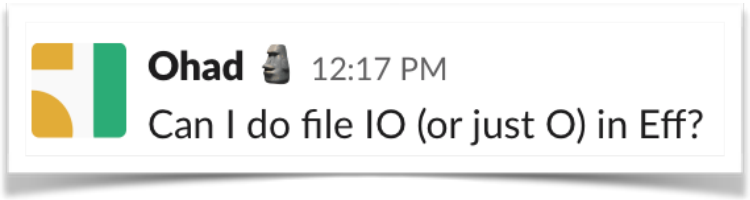
In `pervasives.eff`:

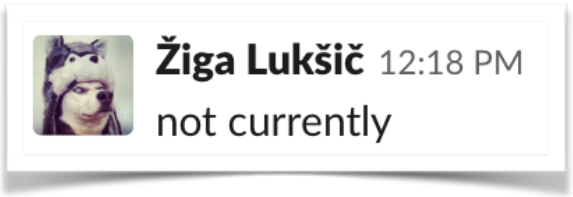
```
effect Write : (string*string) -> unit
```

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** 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 ! *)
  ```

- We could resolve this by typing `fh` **linearly** (but `s` **non-linearly**)
  ```

# 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 ! *)
  ```

- We could resolve this by typing `fh` **linearly** (but `s` **non-linearly**)

- But what if we wrap `f` in a **handler**?

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

  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 ! *)
  ```

- We could resolve this by typing `fh` **linearly** (but `s` **non-linearly**)

- But what if we wrap `f` in a **handler**?

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

  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 programming mechanisms**, 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 programming mechanisms**, 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

$$a \to IO\ b$$

as

$$a \to (RealWorld \to (b, RealWorld))$$

which is the same as

$$(a, RealWorld) \to (b, RealWorld)$$

- With the `System.IO` **module abstraction** ensuring that
  - We **cannot get our hands on** RealWorld
  - The RealWorld is **affected 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

$$a \to IO\ b$$

as

$$a \to (RealWorld \to (b, RealWorld))$$

which is the same as

$$(a, RealWorld) \to (b, RealWorld)$$

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

hGetLine : (Handle, RealWorld) → (String, RealWorld)

hClose : (Handle, RealWorld) → ((), RealWorld)

I.e., **IO** is about the **external world** rather than internal effects!

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

# Refresher: what is a comodel?

# Refresher: what is a comodel?

- A **signature** $\Sigma$ is a set of operation symbols $\mathrm{op} : A \rightsquigarrow B$

# Refresher: what is a comodel?

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

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

# Refresher: what is a comodel?

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

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

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

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

# Refresher: what is a comodel?

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

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

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

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

- Intutively, comodels describe a **evolution of one's world**

# Refresher: what is a comodel?

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

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

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

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

- Intutively, comodels describe a **evolution of one's world**
  - 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)

# Comodels as a gateway to the external world

- ```
  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**
  ```

# Comodels as a gateway to the external world

- ```
  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`

# Comodels as a gateway to the external world

- ```
  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**

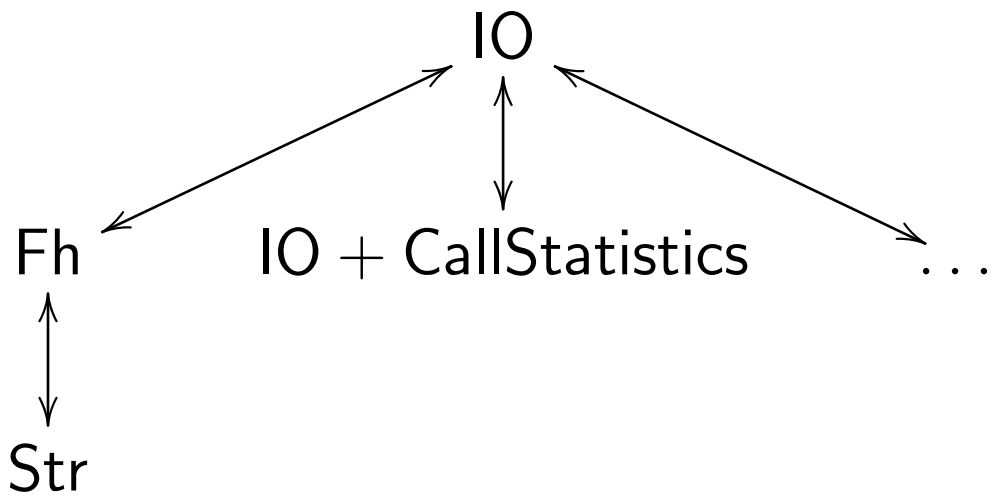
- ```
  let f (s:string) =
    using IO cohandle
      let fh = fopen "foo.txt" in
      fwrite fh (s^s)              (* in IO *)
    finally (fclose fh)
  ```

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

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

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

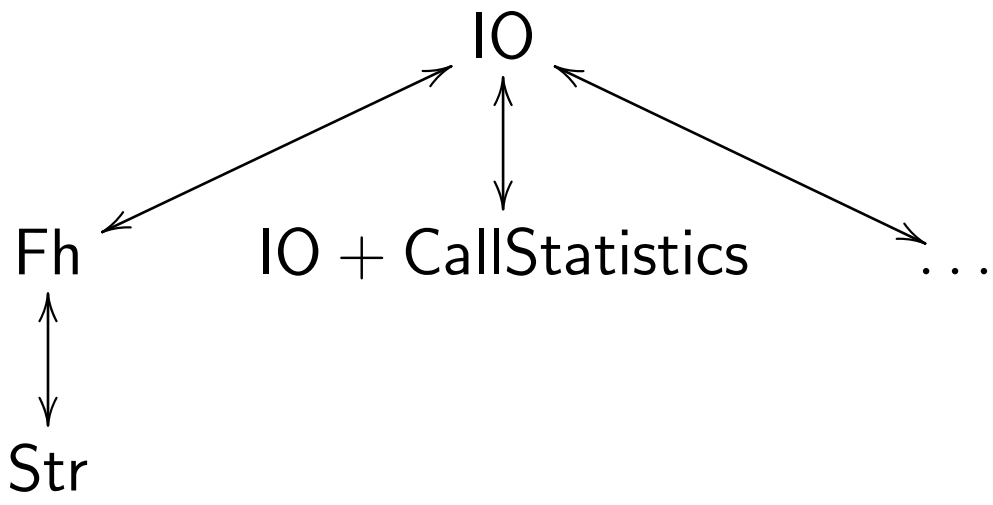
- Examples of **modularity** we might want from comodels




- 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`"

# Comodels as a gateway to the external world

- Examples of **modularity** we might want from comodels

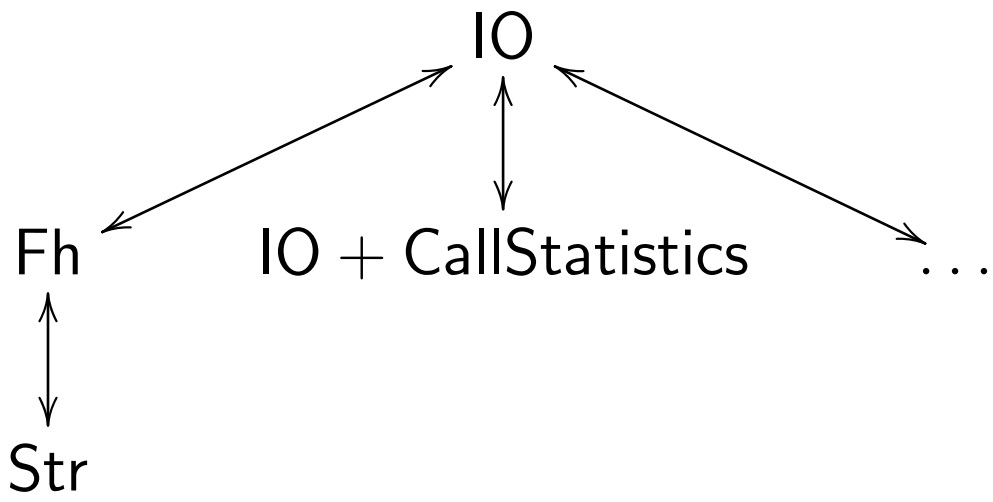



- Fh — "**world** which consists of **exactly one** `fh`"
- IO $\longrightarrow$ Fh — "call `fopen` with `foo.txt`, store returned `fh`"
- Fh $\longrightarrow$ IO — "call `fclose` with stored `fh`"

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

# Comodels as a gateway to the external world

- Examples of **modularity** we might want from comodels




- 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

- Our **general framework** on the file operations example

```
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

- Our **general framework** on the file operations example

```
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

- Our **general framework** on the file operations example

```
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) *)
```

# Comodels as a gateway to the external world

# Comodels as a gateway to the external world

- The **modularity aspect** of our general framework

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

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

  finally
    _ @ fh → fclose_of_io fh
```

where

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

# Comodels as a gateway to the external world

# Comodels as a gateway to the external world

- Comodels can also **extend** the (intermediate) external world

```
let f (s:string) =
  using Stats @ (fopen_of_io "foo.txt")
  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 * string *)
  { co_fread   _ @ (fh,c) → ...,
    co_fwrite  s @ (fh,c) → ...,
    co_reset   _ @ (fh,c) → return ((),(fh,0)) }
```

# Comodels as a gateway to the external world

- Comodels can also **extend** the (intermediate) external world

```
let f (s:string) =
  using Stats @ (fopen_of_io "foo.txt")
  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 * string *)
  { co_fread   _ @ (fh,c) → ...,
    co_fwrite  s @ (fh,c) → ...,
    co_reset   _ @ (fh,c) → return ((),(fh,0)) }
```

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

# So what's happening more formally?

# So what's happening more formally?

- **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 more formally?

- **Types**

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

- **Signatures**

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

# So what's happening more formally?

- **Types**

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

- **Signatures**

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

- **Terms**

$$v \;::=\; x \mid \ldots$$

$$c \;::=\; \mathbf{return}\ v \mid \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \mid v_1 v_2 \mid$$
$$\mathsf{op}\ v \mid \qquad \text{(comodel op.)}$$
$$\mathbf{using}\ C\ @\ c_i\ \mathbf{cohandle}\ c\ \mathbf{finally}\ x\ @\ w \to c_f$$

(simple setting, only comodel ops. and no handlers (wait few slides))

# So what's happening more formally?

- **Types**

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

- **Signatures**

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

- **Terms**

$$v \;::=\; x \mid \ldots$$

$$c \;::=\; \mathbf{return}\ v \mid \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \mid v_1 v_2 \mid$$
$$\mathsf{op}\ v \mid \qquad \text{(comodel op.)}$$
$$\mathbf{using}\ C\ @\ c_i\ \mathbf{cohandle}\ c\ \mathbf{finally}\ x\ @\ w \to c_f$$

(simple setting, only comodel ops. and no handlers (wait few slides))

- **Comodels (cohandlers)**

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

# So what's happening more formally?

# So what's happening more formally?

- **Typing judgements**

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

# So what's happening more formally?

- **Typing judgements**

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

- The two central **typing rules** are

$$\frac{\Gamma \overset{\Sigma}{\vdash} \mathrm{D} \text{ comodel of } \Sigma' \text{ with carrier } W_{\mathrm{D}} \qquad \Gamma \overset{\Sigma}{\vdash} c_i : W_{\mathrm{D}} \qquad \Gamma \overset{\Sigma'}{\vdash} c : A \qquad \Gamma, x : A, w : W_{\mathrm{D}} \overset{\Sigma}{\vdash} c_f : B}{\Gamma \overset{\Sigma}{\vdash} \textbf{using } \mathrm{D} \textbf{ @ } c_i \textbf{ cohandle } c \textbf{ finally } x \textbf{ @ } w \to c_f : B}$$

# So what's happening more formally?

- **Typing judgements**

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

- The two central **typing rules** are

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

and

$$\frac{\mathsf{op} : A \rightsquigarrow B \in \Sigma \qquad \Gamma \vdash v : A}{\Gamma \overset{\Sigma}{\vdash} \mathsf{op}\ v : B}$$

# 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**:

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

  - **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**:

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

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

- The interesting part is the interpretation of

$$\Gamma \overset{\Sigma}{\vdash} \textbf{using } \mathsf{D} \mathbin{@} c_i \textbf{ cohandle } c \textbf{ finally } x \mathbin{@} w \to c_f : B$$

  which is based on the **linear state-passing translation**

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

# So what's happening more formally?

# So what's happening more formally?

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

# So what's happening more formally?

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

# So what's happening more formally?

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

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

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

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

---

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

$$\Downarrow$$

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

# So what's happening more formally?

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

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

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

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

---

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

$$\Downarrow$$

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

- The interpretation of **operations** uses the **co-operations** of Cs
  - In fact, is parametric in the semantics of (outer) co-operations

# 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 that

  - **algebraic operations** (in the sense of $\mathrm{Eff}$) allowed in `c`, but they must not be allowed to escape **`cohandle`** (for linearity)
  - 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 that

  - **algebraic operations** (in the sense of $\mathrm{EFF}$) allowed in `c`, but they must not be allowed to escape **`cohandle`** (for linearity)
  - 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 that

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

- How do **multi-handlers** fit here? Interacting 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., it 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., it lost connection to the HDD where "foo.txt" was
- **Simple idea**: `finally` can act as a **handler** for **broken promises**

# But what about alg. effects and handlers?

- **Second:** What if the **outer comodel beaks its promise**?
  - E.g., it lost connection to the HDD where "foo.txt" was
- **Simple idea**: `finally` can act as a **handler** for **broken promises**

```
using
  C @ c_i
cohandle
  fwrite_of_d s;          (* promise broken here *)
  fread ()
finally @ w →
  | return x → c_f
  | throw e → c_do_some_cleanup
  | op x k → ...
```

# But what about alg. effects and handlers?

- **Second:** What if the **outer comodel beaks its promise**?
  - E.g., it lost connection to the HDD where "foo.txt" was
- **Simple idea**: `finally` can act as a **handler** for **broken promises**

```
using
  C @ c_i
cohandle
  fwrite_of_d s;          (* promise broken here *)
  fread ()
finally @ w →
  | return x → c_f
  | throw e → c_do_some_cleanup
  | op x k → ...
```

  - **Important:** `finally` **does not (!)** jump back into `cohandle`
  - Algebraic operations only allowed to appear in co-operations

# But what about alg. effects and handlers?

# But what about alg. effects and handlers?

- Of course also **initialisation** might **break the promise**

```
using
  C
initially @ c_i
  | op x k → ...
  | ...
cohandle
  c
finally @ w →
  | return x → c_f
  | op x k → ...
  | ...
```

# Conclusions

# Conclusions

- **Comodels** as a gateway for interacting with the **external world**
- We're making them into a **modular programming abstraction**
- **Linearity** by leaving **outer worlds** implicit (via comodel ops.)
- `System.IO`, Koka's **initially** & **finally**, Python's **with**, ...

# Conclusions

- **Comodels** as a gateway for interacting with the **external world**
- We're making them into a **modular programming abstraction**
- **Linearity** by leaving **outer worlds** implicit (via comodel ops.)
- `System.IO`, Koka's `initially` & `finally`, Python's `with`, ...

# Ongoing and future work

- Work out all the **formal details** of what I have shown you today
- **Algebraic effects** and **(multi-)handlers**
- More **examples** and **use cases**
- Clarify the connection with **(effectful) lenses**
- **Combinatorics** of comodels and their lens-like relationships