Comodels as a gateway for interacting with the external world

Danel Ahman

(joint work with Andrej Bauer)

Ljubljana, 21 March 2019

Comodels as a gateway for interacting with the external world

Danel Ahman

(joint work with Andrej Bauer)

Ljubljana, 21 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)))
```

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/effects

```
type IO a

openFile  :: FilePath → IOMode → IO Handle
hGetLine  :: Handle → IO String
hClose    :: Handle → IO ()
```

External world in FP

- Declare a **signature** of monads/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/effects

```
type IO a

openFile  :: FilePath → IOMode → IO Handle
hGetLine  :: Handle → IO String
hClose    :: Handle → IO ()
```

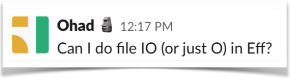
```
effect Read  : string
effect Raise : string → empty

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

- And then treat it **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

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 more 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;

    fread fh
  ```
  ```

Another issue — linearity or lack thereof

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

    fread fh          (* file handle not open !!! *)
  ```
  ```

Another issue — linearity or lack thereof

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

    fread fh          (* file handle not open !!! *)
  ```
- We could resolve this by typing file handles **linearly**
  - But we want other vars. (e.g., strings) to be used **non-linearly**
  ```

Another issue — linearity or lack thereof

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

    fread fh          (* file handle not open !!! *)
  ```
- We could resolve this by typing file handles **linearly**
  - But we want other vars. (e.g., strings) to be used **non-linearly**
- But what if we wrap `f ()` in a **handler**?

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

let g () =
  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;

    fread fh              (* file handle not open !!! *)
  ```
- We could resolve this by typing file handles **linearly**
  - But we want other vars. (e.g., strings) to be used **non-linearly**
- But what if we wrap `f ()` in a **handler**?

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

let g () =
  with h handle f ()          (* dangling fh !!! *)
```
```

So, how could we solve these issues?

So, how could we solve these issues?

- Using **existing programming mechanisms**, e.g.,
  - Modules and abstraction

```
module System.IO where

type IO a

hClose :: Handle → IO ()
```

  - Linear (and non-linear) types

```
linear type fhandle

effect FClose : (linear fhandle) → unit

linear effect FClose : fhandle → unit
```

So, how could we solve these issues?

- Using **existing programming mechanisms**, e.g.,
  - Modules and abstraction

```
module System.IO where

type IO a

hClose :: Handle → IO ()
```

  - Linear (and non-linear) types

```
linear type fhandle

effect FClose : (linear fhandle) → unit

linear effect FClose : fhandle → unit
```

- **Problem:** They don't really explain the **essence of the solution**

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

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

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 ensuring that
  - We cannot get our hands on RealWorld — it's just an idea
  - The **real world** is used linearly
  - We don't ask more from the **real world** 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

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

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

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

So, **IO** is more about in which **external world** our program is in!

Comodels as a gateway to the external world

Comodels as a gateway to the external world

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

Comodels as a gateway to the external world

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

Comodels as a gateway to the external world

- ```
  let f (s:string) =
    using IO run
      let fh = fopen "foo.txt" in
      fwrite fh (s^s);
      fclose fh                          (* @ IO : unit *)
  ```
- Now **external world** explicit, but **dangling** `fh` etc **still possible**
  ```

Comodels as a gateway to the external world

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

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

- ```
  let f (s:string) =
    using IO run
      let fh = fopen "foo.txt" in
      fwrite fh (s^s)                    (* @ IO : unit *)
    ending_with (fclose fh)
  ```

Comodels as a gateway to the external world

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

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

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

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

- ```
  let f (s:string) =
    using IO run
      let fh = fopen "foo.txt" in
      fwrite fh (s^s)                    (* @ IO : unit *)
    ending_with (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

$$
\begin{tikzcd}
 & \mathsf{IO} \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \\
\mathsf{Fh} \arrow[d, leftrightarrow] & \mathsf{IO} + \mathsf{CallStatistics} & \dots \\
\mathsf{Str} & &
\end{tikzcd}
$$

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

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

$$
\begin{tikzcd}
& \mathsf{IO} \arrow[dl, leftrightarrow] \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \\
\mathsf{Fh} \arrow[d, leftrightarrow] & \mathsf{IO} + \mathsf{CallStatistics} & \dots \\
\mathsf{Str} & &
\end{tikzcd}
$$

- 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`"

- **Observation:** IO $\longleftrightarrow$ Fh and others 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) =                    (* @ IO : unit *)
  using Fh
  starting_with (fopen_of_io "foo.txt")
  run
    fwrite_of_fh (s^s)                (* @ Fh : unit *)
  ending_with (fun _ 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) =                    (* @ IO : unit *)
  using Fh
  starting_with (fopen_of_io "foo.txt")
  run
    fwrite_of_fh (s^s)                (* @ Fh : unit *)
  ending_with (fun _ fh → fclose_of_io fh)
```

where

```
Fh =
  ⟨W = fhandle ,
   co_fread ((),fh) = ... ,
   co_fwrite (s,fh) = fwrite_of_io s fh;
                      return ((),fh)⟩

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

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) =                    (* @ IO : unit *)
  using Fh
  starting_with (fopen_of_io "foo.txt")
  run

    using Str
    starting_with (fread_of_fh ())
    run
      fwrite_of_str (s^s)    (* @ Str : unit *)
    ending_with (fun _ s → fwrite_of_fh s)

  ending_with (fun _ fh → fclose_of_io fh)
```

where

```
Str = ⟨W = string , ... ⟩
```

Comodels as a gateway to the external world

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

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

```
let f (s:string) =                    (* @ IO : unit *)
  using Stats
  starting_with (fopen_of_io "foo.txt")
  run
    fwrite_of_stats (s^s)  (* @ Stats : unit *)
  ending_with
    (fun _ (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 , ... ⟩
```

Comodels as a gateway to the external world

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

```
let f (s:string) =                   (* @ IO : unit *)
  using Stats
  starting_with (fopen_of_io "foo.txt")
  run
    fwrite_of_stats (s^s)  (* @ Stats : unit *)
  ending_with
    (fun _ (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 , ... ⟩
```

- Can also keep track of **nondet./prob. choice results**, and alike

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

So what's happening more formally?

- **Typing judgement** for computations $\Gamma \vdash c \mathbin{@} \vec{\mathsf{C}} : A$

So what's happening more formally?

- **Typing judgement** for computations $\Gamma \vdash c \mathbin{@} \vec{\mathsf{C}} : A$
- The two central **typing rules** are (U is the universe, aka IO)

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{D} \text{ comodel} \mathbin{@} \vec{\mathsf{C}} \qquad \mathsf{D} \neq \mathsf{U} \\ \Gamma \vdash c_s \mathbin{@} \vec{\mathsf{C}} : \mathsf{D}.W \quad \Gamma \vdash c \mathbin{@} \vec{\mathsf{C}}, \mathsf{D} : A \quad \Gamma, x\!:\!A, w\!:\!\mathsf{D}.W \vdash c_e \mathbin{@} \vec{\mathsf{C}} : A\end{array}}{\begin{array}{ll}\Gamma \vdash & \textbf{using } \mathsf{D} \\ & \textbf{starting\_with } c_s \\ & \textbf{run } c \\ & \textbf{ending\_with } (x.w.c_e) \quad \mathbin{@} \vec{\mathsf{C}} : A\end{array}}$$

and

$$\frac{\Gamma \vdash \mathsf{D} \text{ comodel} \mathbin{@} \vec{\mathsf{C}} \qquad \mathsf{op} : A \rightsquigarrow B \in \mathsf{D}.\Sigma \qquad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{op}\ v \mathbin{@} \vec{\mathsf{C}}, \mathsf{D} : B}$$

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

So what's happening more formally?

- Regarding the **denotational semantics**, the idea is to interpret

  $$\Gamma \vdash c \mathbin{@} \vec{\mathsf{C}} : A$$

  as

  $$[\![\Gamma \vdash c \mathbin{@} \vec{\mathsf{C}} : A]\!] : [\![\vec{\mathsf{C}}]\!] \longrightarrow [\![A]\!] \times [\![\vec{\mathsf{C}}]\!]$$

  which in its essence is very similar to Møgelberg and Staton's comodels-based **linear state-passing transformation**

So what's happening more formally?

- Regarding the **denotational semantics**, the idea is to interpret

  $$\Gamma \vdash c \mathbin{@} \vec{\mathsf{C}} : A$$

  as

  $$[\![\Gamma \vdash c \mathbin{@} \vec{\mathsf{C}} : A]\!] : [\![\vec{\mathsf{C}}]\!] \longrightarrow [\![A]\!] \times [\![\vec{\mathsf{C}}]\!]$$

  which in its essence is very similar to Møgelberg and Staton's comodels-based **linear state-passing transformation**

- Regarding **operational semantics**, the idea is to consider confs.

  $$(\ \overrightarrow{(\mathsf{C}, w)}\ ,\ \ c\ )$$

  either in a big- or small-step style

  - where $\overrightarrow{(\mathsf{C}, w)}$ is a stack of worlds

So what's happening more formally?

So what's happening more formally?

- For example, consider the big-step evaluation of **using** C ...

So what's happening more formally?

- For example, consider the big-step evaluation of **using** C ...

$$\dfrac{\begin{array}{c} (\ (\overrightarrow{(\mathsf{C}, w_0)}, (\mathsf{C}', w'_0))\ ,\ c_s\ ) \Downarrow (\ (\overrightarrow{(\mathsf{C}, w_1)}, (\mathsf{C}', w'_1))\ ,\ \textbf{return}\ w''_0\ ) \\ (\ (\overrightarrow{(\mathsf{C}, w_1)}, (\mathsf{C}', w'_1), (\mathsf{D}, w''_0))\ ,\ c\ ) \Downarrow (\ (\overrightarrow{(\mathsf{C}, w_2)}, (\mathsf{C}', w'_2), (\mathsf{D}, w''_1))\ ,\ \textbf{return}\ v\ ) \\ (\ (\overrightarrow{(\mathsf{C}, w_2)}, (\mathsf{C}', w'_2))\ ,\ c_e[v/x, w''_1/w]\ ) \Downarrow (\ (\overrightarrow{(\mathsf{C}, w_3)}, (\mathsf{C}', w'_3))\ ,\ \textbf{return}\ v'\ ) \end{array}}{\begin{array}{c} (\ (\overrightarrow{(\mathsf{C}, w_0)}, (\mathsf{C}', w'_0))\ ,\ \textbf{using}\ \mathsf{D}\ \textbf{s\_w}\ c_s\ \textbf{run}\ c\ \textbf{e\_w}\ (x.w.c_e)\ ) \\ \Downarrow \\ (\ (\overrightarrow{(\mathsf{C}, w_3)}, (\mathsf{C}', w'_3))\ ,\ \textbf{return}\ v'\ ) \end{array}}$$

So what's happening more formally?

- For example, consider the big-step evaluation of **using** C ...

$$\dfrac{\begin{array}{c}(\ (\overrightarrow{(\mathsf{C}, w_0)}, (\mathsf{C}', w_0'))\ ,\ c_s\ ) \Downarrow (\ (\overrightarrow{(\mathsf{C}, w_1)}, (\mathsf{C}', w_1'))\ ,\ \textbf{return}\ w_0''\ ) \\ (\ (\overrightarrow{(\mathsf{C}, w_1)}, (\mathsf{C}', w_1'), (\mathsf{D}, w_0''))\ ,\ c\ ) \Downarrow (\ (\overrightarrow{(\mathsf{C}, w_2)}, (\mathsf{C}', w_2'), (\mathsf{D}, w_1''))\ ,\ \textbf{return}\ v\ ) \\ (\ (\overrightarrow{(\mathsf{C}, w_2)}, (\mathsf{C}', w_2'))\ ,\ c_e[v/x, w_1''/w]\ ) \Downarrow (\ (\overrightarrow{(\mathsf{C}, w_3)}, (\mathsf{C}', w_3'))\ ,\ \textbf{return}\ v'\ )\end{array}}{\begin{array}{c}(\ (\overrightarrow{(\mathsf{C}, w_0)}, (\mathsf{C}', w_0'))\ ,\ \textbf{using}\ \mathsf{D}\ \textbf{s\_w}\ c_s\ \textbf{run}\ c\ \textbf{e\_w}\ (x.w.c_e)\ ) \\ \Downarrow \\ (\ (\overrightarrow{(\mathsf{C}, w_3)}, (\mathsf{C}', w_3'))\ ,\ \textbf{return}\ v'\ )\end{array}}$$

- The interpretation of **operations** uses the co-operations of C
  - Their interpretation naturally traverses the stack of worlds

But what about algebraic effects and handlers?

But what about algebraic effects and handlers?

- An interesting question for **future work**, but feels natural that in

```
using C
starting_with c_s
run c
ending_with (fun x w → c_e)
```

  - one can use **algebraic operations** (in the sense of EFF) in `c`
  - but they must not be allowed to escape **`run`** (for linearity)

  - to escape, have to use the **co-operations** of the **external world**
  - it might make sense to allow alg. ops. to escape `C`, `c_s`, `c_e`

  - the continuations of **handlers** in `c` are delimited by **`run`**

Conclusions

Conclusions

- **Comodels** as a gateway for interacting with the **external world**
- We made them into a **modular programming abstraction**
- **Linearity** by leaving **outer worlds** implicit (via alg. ops.)
- **Natural semantics** similar to the linear state-passing translation

Conclusions

- **Comodels** as a gateway for interacting with the **external world**
- We made them into a **modular programming abstraction**
- **Linearity** by leaving **outer worlds** implicit (via alg. ops.)
- **Natural semantics** similar to the linear state-passing translation

Ongoing and future work

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