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

- Using monads (as in Haskell)
  
  \[
  \text{type } \text{St} \ a = \text{String} \to (a, \text{String})
  \]
  
  \[
  f : : \text{St} \ a \to \text{St} (a, a)
  \]
  
  \[
  f \ c = c >> (x \to \text{return} (x, y))
  \]

- Using alg. effects and handlers (as in Eff, Frank, Koka)
  
  \[
  \text{effect} \ \text{Get} : \text{int}
  \]
  
  \[
  \text{effect} \ \text{Put} : \text{int} \to \text{unit} (\ast : \text{int} \to a \ast \text{int} !)\}
  \]
  
  \[
  \text{let} g (c : \text{unit} \to a ! \{} \text{Get}, \text{Put} \} ) =
  \]

  \[
  \text{with} s \text{t} \text{h} \text{handle} (\text{perform} (\text{Put} 42) ; c ())
  \]

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

But what about effects that need access to the external world?
Computational effects in FP

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

```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)

```haskell
effect Get : int
effect Put : int → unit

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?
Computational effects in FP

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

```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)

```haskell
effect Get : int

effect Put : int → unit

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.,

\[
\begin{align*}
\text{(\textit{System.IO})} \\
\text{type IO a} \\
\text{openFile : : FilePath \to IOMode \to IO Handle}
\end{align*}
\]

\[
\begin{align*}
\text{(\textit{pervasives.eff})} \\
\text{effect RandomInt : : int \to int} \\
\text{effect RandomFloat : : float \to float}
\end{align*}
\]

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

\[
\begin{align*}
\text{(\textit{src/runtime/eval.ml})} \\
\text{let rec top_handle op =} \\
\text{match op with} \\
\mid \ldots
\end{align*}
\]
External world in FP
Can I do file IO (or just O) in Eff?
External world in FP

Ohad 12:17 PM
Can I do file IO (or just O) in Eff?

Žiga Lukšič 12:18 PM
not currently
External world in FP

Ohad 12:17 PM
Can I do file IO (or just O) in Eff?

Žiga Lukšič 12:18 PM
not currently

Ohad 8:35 PM
So here’s the hack I added:

```
We should do something a bit more principled
```

```
In `pervasives.eff`:

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

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

```
    | "Write" ->
    |     (match v with
    |       | V.Tuple vs ->
    |         | let (file_name :: str :: _) = List.map V.to_str vs in
    |         | let file_handle = open_out_gen
    |         | [Open_wronly
    |         | ;Open_append
    |         | ;Open_creat
    |         | ;Open_text
    |         | ] 00666 file_name in
    |         | Printf.fprintf file_handle "%s" str;
    |         | close_out file_handle;
    |         | top_handle (k V.unit_value)
    | )
```
We should do something a bit more principled.

```
In pervasive.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_wromly
|  | ;Open_append
|  | ;Open_creat
|  | ;Open_text
|  | ] 0666 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

- \[ \text{let } f(s : \text{string}) = \]
  \[ \text{let } fh = \text{fopen } "\text{foo.txt}" \text{ in} \]
  \[ \text{fwrite } fh \text{ (s}^\text{s}); \]
  \[ \text{fclose } fh; \]
  \[ \text{return } fh \]

- \[ \text{let } g s = \]
  \[ \text{let } fh = f \text{ s in } \text{fread } fh \] (* fh not open ! *)
Another issue — linearity or lack thereof

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

\texttt{let } g s = \\
\texttt{let } fh = f s \texttt{ in} \texttt{fread} fh \hspace{1em} (* fh not open ! *)

- Even worse when we wrap \texttt{f} in a \texttt{handler}?

\texttt{let } h = \texttt{handler} \\
  | \texttt{effect} (FWrite fh s k) \to \texttt{return} ()

\texttt{let } g' s = \\
\texttt{with } h \texttt{ handle } f ()
Another issue — **linearity** or lack thereof

- \[
\begin{align*}
\textbf{let } & \ f \ (s:\text{string}) = \\
& \textbf{let } \ fh = \text{fopen } "\text{foo.txt}" \ \textbf{in} \\
& \text{fwrite } fh \ (s^s); \\
& \text{fclose } fh; \\
& \textbf{return } \ fh
\end{align*}
\]

- \[
\begin{align*}
\textbf{let } & \ g \ s = \\
& \textbf{let } \ fh = \text{f s in } \text{fread } fh \quad (* \text{fh not open !})
\end{align*}
\]

- Even worse when we wrap \( f \) in a \textbf{handler}?

\[
\begin{align*}
\textbf{let } & \ h = \textbf{handler} \\
& \text{effect } (\text{FWrite } fh \ s \ k) \rightarrow \textbf{return } ()
\end{align*}
\]

\[
\begin{align*}
\textbf{let } & \ g' \ s = \\
& \textbf{with } \ h \ \textbf{handle } f () \quad (* \text{dangling fh !})
\end{align*}
\]
So, how could we solve these issues?

• We could try using existing PL techniques, e.g.,
  • Modules and abstraction, e.g.,
    • System.IO type
    • Linear (and non-linear) types and effects
      • linear type
      • linear effect
      • Handlers with finally clauses
        • Problem: They don't really capture the essence of the problem
So, how could we solve these issues?

- We could try using **existing PL techniques**, e.g.,
  - **Modules** and **abstraction**, e.g., `System.IO`:
    ```haskell
    type IO a

    hClose :: Handle \rightarrow IO ()
    ```
  - **Linear** (and non-linear) **types** and **effects**:
    ```haskell
    linear type fhandle

    effect FClose : (linear fhandle) \rightarrow unit
    linear effect FClose : fhandle \rightarrow 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`

```haskell
type IO a

hClose :: Handle → IO ()
```

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

```haskell
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

\[ a \to \text{IO} \text{ b} \]

as

\[ a \to (\text{RealWorld} \to (\text{b, RealWorld})) \]

which is the same as

\[ (a, \text{RealWorld}) \to (\text{b, RealWorld}) \]
So, what is that **essence** then?

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

\[ a \rightarrow \text{IO} \ b \]

as

\[ a \rightarrow (\text{RealWorld} \rightarrow (b, \text{RealWorld})) \]

which is the same as

\[ (a, \text{RealWorld}) \rightarrow (b, \text{RealWorld}) \]

- With the **System.IO module abstraction** ensuring that
  - We **cannot get our hands on** RealWorld (no get and put)
  - We have the impression of RealWorld **used linearly**
  - We **don’t ask more** from RealWorld than it can provide
So, what is that essence then?

- Let’s look at Haskell’s IO monad again
- A common explanation is to think of functions
  \[ a \rightarrow \text{IO} \ b \]
  as
  \[ a \rightarrow (\text{RealWorld} \rightarrow (b, \text{RealWorld})) \]
  which is the same as
  \[ (a, \text{RealWorld}) \rightarrow (b, \text{RealWorld}) \]

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

\[ \text{hGetLine} : (\text{Handle}, \text{RealWorld}) \rightarrow (\text{String}, \text{RealWorld}) \]
\[ \text{hClose} : (\text{Handle}, \text{RealWorld}) \rightarrow (() \text{, RealWorld}) \]

Important: co-operations (hClose) make a promise to return!
Refresher: what’s comodel?
Refresher: what’s **comodel**?

- A **signature** \( \Sigma \) is a set of operation symbols \( \text{op} : A_{\text{op}} \leadsto B_{\text{op}} \).
Refresher: what’s comodel?

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

- A **model/algebra/handler** $\mathcal{M}$ of $\Sigma$ is given by

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

- 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 Effv2 (Bauer & Pretnar)
Refresher: what’s **comodel**?

- A **signature** \( \Sigma \) is a set of operation symbols \( \text{op} : A_{\text{op}} \xrightarrow{} B_{\text{op}} \)

- A **model/algebra/handler** \( \mathcal{M} \) of \( \Sigma \) is given by
  \[ \mathcal{M} = \langle M : \text{Set} , \{ \text{op}_\mathcal{M} : A_{\text{op}} \times M^{B_{\text{op}}} \rightarrow M \}_{\text{op} \in \Sigma} \rangle \]

- A **comodel/coalgebra/cohandler** \( \mathcal{W} \) of \( \Sigma \) is given by
  \[ \mathcal{W} = \langle W : \text{Set} , \{ \overline{\text{op}}_\mathcal{W} : A_{\text{op}} \times W \rightarrow B_{\text{op}} \times W \}_{\text{op} \in \Sigma} \rangle \]

- Intuitively, comodels describe **evolution of the world** \( \mathcal{W} \)
Refresher: what’s comodel?

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

- A **model/algebra/handler** $\mathcal{M}$ of $\Sigma$ is given by

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

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

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

- Intuitively, comodels describe **evolution of the world** $\mathcal{W}$

  - 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 $\text{EFF v2}$ (Bauer & Pretnar)
Towards a general programming abstraction

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

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

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

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

Solution: Modular treatment of external worlds
Towards a general programming abstraction

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

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

Now external world explicit, but dangling fh etc still possible

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

Better, but have to explicitly open and thread through fh
Towards a general programming abstraction

- \texttt{let f (s:string) =}
  \begin{verbatim}
  using IO cohandle
  let fh = fopen "foo.txt" in
  fwrite fh (s^s);
  fclose fh
  \end{verbatim}

Now \textbf{external world} explicit, but \textbf{dangling \texttt{fh} etc still possible}

- \texttt{let f (s:string) =}
  \begin{verbatim}
  using IO cohandle
  let fh = fopen "foo.txt" in
  fwrite fh (s^s) \texttt{\ (* in IO \*)}
  finally (fclose fh)
  \end{verbatim}

Better, but \textbf{have to explicitly open and thread through \texttt{fh}}

- \textbf{Solution: Modular treatment of \textbf{external worlds}}
Modular treatment of external worlds

- For example

```
  Fh ← IO
  Fh → IO
```

(ext. world)

(inner world)

- 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”
Modular treatment of external worlds

- For example

```
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  ↓  IO + CallStatistics  ↓  . . .  ↓  IO  ↓  Fh
    ↓                            ↓
Str
```

- 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

```
Pure

Fh

Str

IO

IO + CallStatistics

... (inner world)

(ext. world)
```

• 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

\[
\text{Pure} \quad \rightarrow \quad \text{IO} \quad \leftarrow \quad \text{Fh} \quad \rightarrow \quad \text{IO + CallStatistics} \quad \rightarrow \quad \ldots
\]

(ex. world)

\[\text{(inner world)}\]

\[\text{(inner}^2\text{ world)}\]

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

- Observation: IO $\leftrightarrow$ Fh and other $\leftrightarrow$ look a lot like lenses
Comodels as a gateway to the external world
Comodels as a gateway to the external world

```plaintext
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

```plaintext
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**

```plaintext
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 (\(\text{IO} \leftrightarrow \text{Fh} \leftrightarrow \text{Str}\))
Modular treatment of worlds \((\text{IO} \leftrightarrow \text{Fh} \leftrightarrow \text{Str})\)

```latex
let \(f\) \((s:\text{string})\) = \((\star \text{ in IO } \star)\)

\text{using Fh @ (fopen_of_io "foo.txt")}
\text{cohandle}

\text{using Str @ (fread_of_fh ())} \quad (\star \text{ in Fh } \star)
\text{cohandle}

\text{write_of_str (s^s)} \quad (\star \text{ in Str } \star)
\text{finally}

_ @ s \rightarrow \text{fwrite_of_fh s}

\text{finally}

_ @ fh \rightarrow \text{fclose_of_io fh}

\text{where}

\text{Str} = \{ \text{co_write s @ s'} \rightarrow \} \quad (\star \text{ W = string } \star)

\text{return} \(((),\text{s'}^\text{s})\) \}
```
Tracking the external world usage (IO \rightleftharpoons Stats)

let f (s : string) = (∗in IO∗)

using Stats @ (fh = fopen of io "foo.txt" in return (fh, 0))

cohandle fwrite of stats (sˆs)

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 = fh handle ∗nat ∗)

{ co fwrite s @ (fh, c) → . . . ,
  cor reset @ (fh, c) → return ((), (fh, 0)) }
Tracking the external world usage ($\text{IO} \leftrightarrow \text{Stats}$)

```ml
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 \rightleftharpoons \text{Stats})

```haskell
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) \rightarrow
    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) \rightarrow \ldots ,
     co_reset _ @ (fh,c) \rightarrow return ((() , (fh,0)) ) \} 
```

- Can also track results of nondet./prob. choices, etc
The external world can also be pure (Pure $\leftrightarrow$ Str)
The external world can also be pure \((\text{Pure} \leftrightarrow \text{Str})\)

```plaintext
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 ⇒ wait a few slides)
So what’s happening more formally?

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

\[ A, B, W ::= b \mid 1 \mid A \times B \mid 0 \mid A + B \mid A \rightarrow^\omega B \]
So what’s happening more formally?

• Core calculus for cohandlers (wo/ handlers ⇒ wait a few slides)

• Types

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

• Signatures of (external) worlds

\[ \omega ::= \{ \text{op}_1 : A_1 \rightsquigarrow B_1 , \ldots , \text{op}_n : A_n \rightsquigarrow B_n \} \]
So what’s happening more formally?

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

- Types

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

- Signatures of (external) worlds

\[ \omega ::= \{ \text{op}_1 : A_1 \xrightarrow{\sim} B_1 , \ldots , \text{op}_n : A_n \xrightarrow{\sim} B_n \} \]

- Computation terms (value terms are unsurprising)

\[ c ::= \text{return } v \mid \text{let } x = c_1 \text{ in } c_2 \mid v_1 v_2 \]
\[ \mid \hat{\text{op}} v \]
\[ \mid \text{using } C \circ c_i \text{ cohandle } c \text{ finally } x \circ w \rightarrow c_f \] (comodel op.)
\[ \text{(cohandling)} \]
So what’s happening more formally?

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

- **Types**

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

- **Signatures of (external) worlds**

  \[ \omega ::= \{ \text{op}_1 : A_1 \xrightarrow{\sim} B_1 , \ldots , \text{op}_n : A_n \xrightarrow{\sim} B_n \} \]

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

  \[ c ::= \text{return} \ v \mid \text{let} \ x = c_1 \ \text{in} \ c_2 \mid v_1 v_2 \]

  \[ \mid \hat{\text{op}} \ v \mid \text{using} \ C @ c_i \ \text{cohandle} \ c \ \text{finally} \ x @ w \rightarrow c_f \]

- **Comodels (cohandlers)**

  \[ C ::= \{ \overline{\text{op}}_1 x @ w \rightarrow c_1 , \ldots , \overline{\text{op}}_n x @ w \rightarrow c_n \} \]
So what’s happening more formally?
So what’s happening more formally?

- Typing judgements

\[
\Gamma \vdash v : A \quad \Gamma \vdash c : A
\]
So what’s happening more formally?

- **Typing judgements**

  \[ \Gamma \vdash \nu : A \quad \Gamma \vdash c : A \]

- The two central **typing rules** are

  \[ \Gamma \vdash \omega \text{ D comodel of } \omega' \text{ with carrier } W_D \quad \Gamma \vdash c_i : W_D \]
  \[ \Gamma \vdash' \ c : A \quad \Gamma, x : A, w : W_D \vdash c_f : B \]
  \[ \Gamma \vdash \text{ using } D @ c_i \text{ cohandle } c \text{ finally } x \odot w \rightarrow c_f : B \]

  and

  \[ \text{op : } A_{\text{op}} \rightsquigarrow B_{\text{op}} \in \omega \quad \Gamma \vdash \nu : A_{\text{op}} \]
  \[ \Gamma \vdash \widehat{\text{op}} \nu : B_{\text{op}} \]
Denotational semantics

Term interpretation looks very similar to alg. effects:

\[ \Gamma \vdash v : A \]

\[ \Gamma K \rightarrow J A K \Gamma \vdash \omega : c : A \]

\[ \Gamma K \vdash \omega D \text{ comodel of } \omega' \]

\[ \Gamma \vdash \omega D \Gamma \vdash \omega c \iota : W D \]

\[ \Gamma \vdash \omega_{\iota : W} c : A \]

\[ \Gamma \vdash \omega_{\iota : W} c f : B \]

\[ \Gamma \vdash \omega_{\iota : W} c f : B \]

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

\[ J D K \in \text{Comod } \omega' (\text{Kleisli}(T \omega)) \]
Denotational semantics

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

\[
\begin{align*}
[\Gamma \models \omega c : A] : [\Gamma] &\rightarrow T\omega [A]
\end{align*}
\]

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

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

\[
\begin{align*}
[\Gamma \vdash v : A] : [\Gamma] & \rightarrow [A] \\
[\Gamma \vdash c : A] : [\Gamma] & \rightarrow T_\omega [A]
\end{align*}
\]

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

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

\[
\begin{align*}
\Gamma \vdash c : A & \quad \Gamma, x : A, w : W_D \vdash c_f : B \\
\hline
\Gamma \vdash \text{using } D \odot c_i \text{ cohandle } c \text{ finally } x \odot w \rightarrow c_f : B
\end{align*}
\]

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

\[
[D] \in \text{Comod}_{\omega'}(\text{Kleisli}(T_\omega))
\]

\[
\text{cohandle} \_\text{with} [D] : T_{\omega'} [A] \rightarrow (\left([W_D] \rightarrow T_\omega ([A] \times [W_D])\right))
\]
Operational semantics

• Idea is to consider configurations $\langle C, w \rangle, c$

• For example, consider the big-step evaluation of using $D @$

\[
\begin{align*}
\langle \langle C', w'_{0} \rangle, (C', w'_{1}) \rangle & \downarrow \langle (C, w_{1}), (C', w'_{2}) \rangle, c_{f} \left[ v / x, w''_{0} / w \right] \\
\langle \langle C', w'_{1} \rangle, (C', w'_{2}) \rangle & \downarrow \langle (C, w_{2}), (C', w'_{3}) \rangle, c_{f}
\end{align*}
\]

• The interpretation of operations uses the co-operations of Cs
Operational semantics

- Idea is to consider configurations \( (\overrightarrow{C}, w), c \)
**Operational semantics**

- Idea is to consider configurations \( (C, w) , \ c \ )

- For example, consider the **big-step evaluation** of using D ...
Operational semantics

- Idea is to consider configurations \((C, w)\)

- For example, consider the **big-step evaluation** of using \(D\) ...

\[
\begin{align*}
&(((C, w_0), (C', w'_0)), \ c) \Downarrow (((C, w_1), (C', w'_1)), \ \text{return } w''_0) \\
&(((C, w_1), (C', w'_1), (D, w''_0)), \ c) \Downarrow (((C, w_2), (C', w'_2), (D, w''_1)), \ \text{return } v) \\
&(((C, w_2), (C', w'_2)), \ c_f[v/x, w''_1/w]) \Downarrow (((C, w_3), (C', w'_3)), \ \text{return } v')
\end{align*}
\]

\[
(((C, w_0), (C', w'_0)), \ \text{using } D \ @ \ c_i \ \text{cohandle } c \ \text{finally } x \ @ \ w \rightarrow c_f) \\
\Downarrow \\
(((C, w_3), (C', w'_3)), \ \text{return } v')
\]
Operational semantics

- Idea is to consider configurations \((C, w), c\)
- For example, consider the **big-step evaluation** of using \(D\) ...

\[
\begin{align*}
( ((C, w_0), (C', w'_0)), c_i ) &\downarrow ( ((C, w_1), (C', w'_1)), \text{return } w''_0 ) \\
( ((C, w_1), (C', w'_1), (D, w''_0)), c ) &\downarrow ( ((C, w_2), (C', w'_2), (D, w''_1)), \text{return } \nu ) \\
( ((C, w_2), (C', w'_2)), c_f[\nu/x, w''_1/w] ) &\downarrow ( ((C, w_3), (C', w'_3)), \text{return } \nu' )
\end{align*}
\]

\[
( ((C, w_0), (C', w'_0)), \text{using } D \odot c_i \text{ cohandle } c \text{ finally } x \odot w \rightarrow c_f ) \\
\downarrow \\
( ((C, w_3), (C', w'_3)), \text{return } \nu' )
\]

- The **interpretation of operations** uses the **co-operations** of Cs
But what about \textit{alg. effects} and \textit{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
cohhandle c
finally x @ w → c_f
```

it is natural to want that

- **algebraic operations** (in the sense of $\text{EFF}$) are allowed in $c$, but they must not be allowed to escape **cohhandle**

- to escape, have to use the **co-operations** of the **external world**
But what about \texttt{alg. effects} and \texttt{handlers}?

- \textbf{First}: combining this with \texttt{standard alg. effects} and \texttt{handlers}

In the following

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

it is natural to want that

- \textbf{algebraic operations} (in the sense of \texttt{EFF}) are allowed in \texttt{c}, but they must not be allowed to escape \texttt{cohandle}

- to escape, have to use the \texttt{co-operations} of the \texttt{external world}

- the \texttt{continuations of handlers} in \texttt{c} are delimited by \texttt{cohandle}
But what about **alg. effects** and **handlers**?

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

  ```
  using C @ c_i 
  cohandle c
  finally x @ w → c_f
  ```

  it is natural to want that

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

- 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**

```plaintext
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

• 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:
    \[ f : A \rightarrow B \in \text{OCaml} \]
    \[ f : A \times W \text{OCaml} \rightarrow B \times W \text{OCaml} \in \text{OCaml} \]

Some ongoing work

• Interaction with algebraic effects and \([\text{multi-}]\) handlers
• Clarify the connection with (effectful) lenses
• Combinatorics of comodels and their lens-like relationships
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**

\[
\begin{align*}
f : A &\rightarrow B \in \text{OCaml} \\
\bar{f} : A \times W_{\text{OCaml}} &\rightarrow B \times W_{\text{OCaml}} \in \text{OCaml}
\end{align*}
\]
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**

\[
\begin{align*}
  f &: A \rightarrow B \in OCAML \\
  \bar{f} &: A \times W_{OCaml} \rightarrow B \times W_{OCaml} \in OCaml
\end{align*}
\]

Some ongoing work

- Interaction with **algebraic effects** and (multi-)handlers

- Clarify the connection with (effectful) lenses

- **Combinatorics** of comodels and their lens-like relationships