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)
  
  \[
  \text{type}\ St\ a = \text{String} \rightarrow (a, \text{String})
  \]

  \[
  f : S t\ a \rightarrow S t\ (a, a)
  \]

  \[
  f\ c = c >>=(\ x \rightarrow c >>= (\ y \rightarrow \text{return}(x, y)))
  \]

- Using alg. effects and handlers (as in Eff, Frank, Koka)

  \[
  \text{effect}\ \text{Get} : \text{int}
  \]

  \[
  \text{effect}\ \text{Put} : \text{int} \rightarrow \text{unit}
  \]

  \[
  \text{let}\ g (c : \text{unit} \rightarrow \text{int}!) =
  \text{with}\ s t h e a d (\text{perform}(\text{Put} 42); c())
  \]

- 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 >>= (λx → c >>= (λy → return (x, y)))
  ```

- Using **alg. effects** and **handlers** (as in Eff, Frank, Koka)

  ```haskell
  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?
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 >>= (λ x → c >>= (λ y → return (x, y)))
```

- Using **alg. effects** and **handlers** (as in EFF, FRANK, KOKA)

```haskell
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 (*)

  ```ml
  type IO a
  openFile :: FilePath -> IOMode -> IO Handle
  ```

  (*)  pervasives.eff (*)

  ```ml
  effect RandomInt    : int -> int
  effect RandomFloat : float -> float
  ```

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

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

  ```ml
  let rec top_handle op =
      match op with
      | ...
External world in FP
External world in FP

Ohad 12:17 PM
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:

```
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

- \[
\begin{align*}
\text{let } & f (s : \text{string}) = \\
& \quad \text{let } fh = \text{fopen } "\text{foo.txt}" \text{ in} \\
& \quad \text{fwrite } fh (s^s); \\
& \quad \text{fclose } fh; \\
& \quad \text{return } fh \\
\text{let } & g s = \\
& \quad \text{let } fh = f s \text{ in } \text{fread } fh \quad (\ast \text{fh not open!}\ast)
\end{align*}
\]
Another issue — **linearity** or lack thereof

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

\[
\text{let } gs = \\
  \text{let } fh = f \ s \ \text{in} \ \text{fread } fh \ (* \ fh \ not \ open \ ! \ *)
\]

- Even worse when we wrap `f` in a **handler**?

\[
\text{let } h = \text{handler} \\
  \mid \text{effect } (\text{FWrite } fh \ s \ k) \rightarrow \text{return } ()
\]

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

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

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

- Even worse when we wrap \(f\) in a **handler**?

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

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

- We could try using existing PL techniques, e.g.,
  - Modules and abstraction, e.g., `System.IO` type `IO a` `close :: Handle -> IO ()`
  - Linear (and non-linear) types and effects
    - Linear type `f handle` effect `fclose :: (linear f handle) -> unit` `linear effect fclose :: handle -> unit`
  - 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 -> 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**
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

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

as

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

which is the same as

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

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

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

as

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

which is the same as

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

- 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**!

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

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

**Signature**

A signature $\Sigma$ is a set of operation symbols $\text{op}: A \rightarrow B$.

**Model**

A model/algebra/handler $M$ of $\Sigma$ is given by

$$M = \langle M: \text{Set}, \{ \text{op}_M: A \times M \rightarrow B \} \rangle_{\text{op} \in \Sigma}$$

**Comodel**

A comodel/coalgebra/cohandler $W$ of $\Sigma$ is given by

$$W = \langle W: \text{Set}, \{ \text{op}_W: A \times W \rightarrow B \times W \} \rangle_{\text{op} \in \Sigma}$$

Intuitively, 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 Eff² (Bauer & Pretnar).
Refresher: what’s comodel?

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

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

- Intuitively, 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 $\text{Eff}^2$ (Bauer & Pretnar)
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

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

• 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 $\mathcal{W}$

• 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$^2$ (Bauer & Pretnar)
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** $W$

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

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

Now external world explicit, but dangling `fh` etc still possible
Towards a general programming abstraction

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

Now external world explicit, but dangling \( fh \) etc still possible

- \[
    \text{let } f \ (s: \text{string}) = \\
    \quad \text{using IO cohandle} \\
    \quad \text{let } fh = \text{fopen "foo.txt" in} \\
    \quad \text{fwrite } fh \ (s^s) \quad (\ast \ \text{in IO } \ast) \\
    \quad \text{finally } (\text{fclose } fh) \\
    \]

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

- **let** \( f (s: \text{string}) = \)
  ```
  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** \( f (s: \text{string}) = \)
  ```
  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** \( fh \)

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

- For example

\[ \text{Fh} \rightarrow \text{IO} \rightarrow \text{(ext. world)} \]

\[ \text{(inner world)} \rightarrow \text{...} \rightarrow \text{Fh} \]

- 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

- 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

  - \( \text{Fh} \rightarrow \text{IO} \rightarrow \text{IO + CallStatistics} \rightarrow \ldots \) (ext. world)
  - \( \text{Str} \) (inner\(^2\) world)

- \( \text{Fh} \) — “world which consists of exactly one \( \text{fh} \)”
- \( \text{IO} \rightarrow \text{Fh} \) — “call \text{fopen} with \text{foo.txt}, store returned \text{fh}”
- \( \text{Fh} \rightarrow \text{IO} \) — “call \text{fclose} with stored \text{fh}”
- \( \text{Str} \) — “world that is blissfully unaware of \( \text{fh} \)”
Modular treatment of external worlds

• For example

- **Pure**
- **IO**
- **(ext. world)**
- **Fh**
- **IO + CallStatistics**
- **(inner world)**
- **Str**
- **(inner^2 world)**

- **Fh** — “world which consists of exactly one \( \text{fh} \)”
- **IO \( \rightarrow \) Fh** — “call fopen with foo.txt, store returned \( \text{fh} \)”
- **Fh \( \rightarrow \) IO** — “call fclose with stored \( \text{fh} \)”
- **Str** — “world that is blissfully unaware of \( \text{fh} \)”
Modular treatment of external worlds

- For example

\[
\text{Pure} \xrightarrow{\text{IO}} \text{Fh} \xrightarrow{\text{Str}} \text{IO + CallStatistics} \xrightarrow{\ldots} \text{(ext. world)}
\]

\[
\text{Fh} \xrightarrow{\text{IO}} \text{(inner world)}
\]

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

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

- \textbf{Observation:} \text{IO} \leftrightarrow \text{Fh} and other \leftrightarrow look a lot like \textit{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
```

(* in IO *)
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 (IO $\leftrightarrow$ Fh $\leftrightarrow$ Str)
Modular treatment of worlds (IO ↔ Fh ↔ Str)

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

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

  finally
    _ @ fh → fclose_of_io fh

where

Str = { co_write s @ s' → (* W = string *)
        return ((), s'^ s) }
Tracking the external world usage (IO ↔ Stats)

let f(s: string) = (* in IO *)
using Stats @ (fh = fopen of io "foo.txt" in return (fh, 0))
cohandle fwrite of stats (s âˆ’ s)
finally @ (fh, c) →
  let fh' = fopen of io "stats.txt" in
cwrite of io fh' c; fclose of io fh';
close of io fh
where
Stats = (* W = fh handle *)
  { cwrite @ (fh, c) → . . . ,
    creset @ (fh, c) → return ((), (fh, 0))
}

• Can also track results of nondet./prob. choices
Tracking the external world usage (IO ↔ Stats)

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

cohandle
  fwrite_of_stats (s^s)

finally
  _ @ (fh,c) →
    let fh' = fopen_of_io "stats.txt" in
    fwrite_of_io fh' c; fclose_of_io fh';
    fclose_of_io fh

where

Stats = (* W = fhandle * nat*)
{ co fwrite s @ (fh,c) → ... ,
  co reset _ @ (fh,c) → return ((),(fh,0)) }
```
**Tracking the external world usage (IO ←→ Stats)**

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

cohandle
    fwrite_of_stats (s^s)
finally
  _ @ (fh,c) →
    let fh' = fopen_of_io "stats.txt" in
    fwrite_of_io fh' c; fclose_of_io fh';
    fclose_of_io fh

where

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

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

```ocaml
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

```ocaml
Str = (* W = string *)
  { co_read _ @ s → return (s,s) ,
    co_write s @ _ → return ((),s) }
```
So what’s happening more formally?

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

• Types $A, B, W ::= b | 1 | A \times B | 0 | A + B | A^\omega \rightarrow B$

• Signatures of (external) worlds $\omega ::= \{ op_1 : A_1 \Rightarrow B_1, \ldots, op_n : A_n \Rightarrow B_n \}$

• Computation terms (value terms are unsurprising) $c ::= \text{return } v | \text{let } x = c_1 \text{ in } c_2 | v_1 v_2 | \hat{\text{op}} v (\text{comodel op.}) | \text{using } C @ c_i \text{ cohandle } c_f (\text{cohandling})$

• Comodels (cohandlers) $C ::= \{ op_1 x @ w \rightarrow c_1, \ldots, op_n x @ w \rightarrow c_n \}$
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**

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

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

- Computation terms (value terms are unsurprising)

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

  \[ \mid \text{\hat{op}} v \]

  \[ \mid \text{using } C \odot c_i \text{ cohandle } c \text{ finally } x \odot w \rightarrow c_f \] (comodel op.) (cohandling)
So what’s happening more formally?

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

• Types

\[ A, B, W ::= b \mid 1 \mid A \times B \mid 0 \mid A + B \mid A \leadsto B \]

• Signatures of (external) worlds

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

• Computation terms (value terms are unsurprising)

\[ c ::= \text{return } v \mid \text{let } x = c_1 \text{ in } c_2 \mid v_1 v_2 \]
\[ \mid \hat{\text{op}} v \]
\[ \mid \text{using } C @ c_i \text{ cohandle } c \text{ finally } x @ w \rightarrow c_f \] (comodel op.) (cohandling)

• Comodels (cohandlers)

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

- **Typing judgements**

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

- **Typing judgements**

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

- The two central **typing rules** are

\[
\begin{align*}
\Gamma \vdash \omega \text{ D comodel of } \omega' \text{ with carrier } W_D & \quad \Gamma \vdash c_i : W_D \\
\Gamma \vdash' c : A & \quad \Gamma, x : A, w : W_D \vdash \omega c_f : B \\
\Gamma \vdash \text{ using } D \odot c_i \text{ cohndle } c \text{ finally } x \odot w \rightarrow c_f : B
\end{align*}
\]

and

\[
\begin{align*}
op : \A_{op} \rightsquigarrow \B_{op} \in \omega & \quad \Gamma \vdash \nu : \A_{op} \\
\Gamma \vdash \hat{\nu} \exists \nu : \B_{op}
\end{align*}
\]
Denotational semantics

• Term interpretation looks very similar to alg. effects:

\[ J \Gamma \vdash v : A K : J \Gamma K \rightarrow J A K \]

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

• The interesting part is the interpretation of cohandling:

\[ \Gamma \vdash \omega D \text{ comodel of } \omega' \text{ with carrier } W D \]

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

\[ \Gamma \vdash \omega' c i : A \Gamma, x : A, w : W D \vdash \omega c f : B \]

\[ \Gamma \vdash \omega \text{ using } D @ c i \text{ cohandle } c \text{ finally } x @ w \rightarrow c f : B \]

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

\[ J \Gamma \in \text{Comod } \omega' (Kleisli(} T \omega) \]
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**!
Denotational semantics

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

  \[ \Gamma \vdash v : A \mid \Gamma \overrightarrow{} [A] \quad \Gamma \vdash c : A \mid \Gamma \overrightarrow{} T_\omega [A] \]

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

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

  \[
  \begin{align*}
  &\Gamma \vdash \omega \mid D \text{ comodel of } \omega' \text{ with carrier } W_D \\
  &\Gamma \vdash c_i : W_D \\
  &\Gamma \vdash c : A \\
  &\Gamma, x : A, w : W_D \vdash c_f : B \\
  \hline
  &\Gamma \vdash \omega \text{ using } D \otimes c_i \text{ cohandle } c \text{ finally } x \otimes w \rightarrow c_f : B
  \end{align*}
  \]

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

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

• Idea is to consider configurations $(C, w)$, $c$

• For example, consider the big-step evaluation of using $D @ c i cohandle c finally x @ w → c f$
Operational semantics

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

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

\[
\begin{align*}
( (\langle C, w_0 \rangle, \langle C', w_0' \rangle), \text{ using } D @ c_i \text{ cohandle } c \text{ finally } x @ w \rightarrow c_f ) & \Downarrow ( (\langle C, w_3 \rangle, \langle C', w_3' \rangle), \text{ return } v' )
\end{align*}
\]
Operational semantics

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

- For example, consider the **big-step evaluation** of \(\text{using } D \ldots\)

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

\[
\begin{align*}
&((C, w_0), (C', w_0')) \xrightarrow{\text{using } D \odot c_i \text{ cohandle } c \text{ finally } x \odot w \rightarrow c_f} \\
&\quad\downarrow \\
&\quad((C, w_3), (C', w_3')) \text{ return } v'
\end{align*}
\]

- The **interpretation of operations** uses the **co-operations** of Cs
But what about alg. effects and handlers?

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

In the following using C @ c_i cohandle c finally x @ w → c_f it is natural to want that

algebraic operations (in the sense of Eff) are allowed in c, but they must not be allowed to escape cohandle.

to escape, have to use the co-operations of the external world.

the continuations of handlers in c are delimited by cohandle.

Where do multi-handlers fit? Co-operating handlers-cohandlers?
But what about alg. effects and handlers?

- **First**: combining this with *standard alg. effects* and *handlers*
But what about **alg. effects** and **handlers**?

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

- In the following

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

  it is natural to want that

  - **algebraic operations** (in the sense of $\text{EFF}$) are allowed in $c$, but they must not be allowed to escape
  
  - to escape, have to use the **co-operations** of the **external world**
But what about *alg. effects and handlers*?

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

- In the following

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

  it is natural to want that

  - *algebraic operations* (in the sense of \( \text{EFF} \)) 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 \( \text{cohandle} \)
But what about alg. effects and handlers?

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

- In the following

  ```plaintext
  using C @ c_i
  cohandle c
  finally x @ w \rightarrow 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:
  \( f : A \rightarrow B \in \text{OCaml} \quad f : A \times W \rightarrow B \times W \in \text{OCaml} \)

Some ongoing work

• Interaction with algebraic effects and (multi-) handlers

• Clarify the connection with (effectful) lenses

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

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