Comodels as a gateway for interacting with the external world

Danel Ahman
(joint work with Andrej Bauer)

University of Ljubljana, Slovenia

MSR Redmond, 15 May 2019
A modular programming abstraction for using external resources

Danel Ahman
(joint work with Andrej Bauer)

University of Ljubljana, Slovenia

MSR Redmond, 15 May 2019
Computational effects in FP

• Using monads (as in Haskell)

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

\[
f :: \text{St } a \rightarrow \text{St } (a, a)
\]

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

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

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

\[
\text{effect Put : int} \rightarrow \text{unit} (\ast : \text{int} \rightarrow \text{a} \ast \text{int} ! \{} \ast \}
\]

\[
\text{let } g (c : \text{unit} \rightarrow \text{a} ! \{ \text{Get}, \text{Put} \}) =
\]

\[
\text{with st} \text{st} \text{handle} (\text{perform}(\text{Put} 42) ; c())
\]

• 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 (∗ : int → a ∗ int ! {} ∗)
  
  let g (c : unit → a ! {Get, Put}) = with s t 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**)

  ```
  type St a = String -> (a, String)
  
f :: St a -> St (a, a)
f c = c >>= (
x -> c >>= (\y -> return (x, y)))
  ```

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

  ```
  effect Get : int
  effect Put : int -> unit

  let g (c:unit -> a!{Get, Put}) =
    with st_h handle (perform (Put 42); c ())
  ```
Computational effects in FP

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

  ```
  type St a = String → (a, String)
  
  f :: St a → St (a, a)
  f c = c >>= (λ x → c >>= (λ y → return (x, y)))
  ```

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

  ```
  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.,

```
(* System.IO *)
type IO a
openFile :: FilePath → IOMode → IO Handle
```

```
(* pervasives.eff *)
effect RandomInt : int → int
effect RandomFloat : float → float
```

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

```
(* eff/src/backends/runtime/eval.ml *)
let rec top_handle op =
  match op with
  | ... but ...
```
An issue — difficult to cover all use cases
An issue — difficult to cover all use cases

Ohad  12:17 PM
Can I do file IO (or just O) in Eff?
An issue — difficult to cover all use cases

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

Žiga Lukšič  12:18 PM
not currently
An issue — difficult to cover all use cases

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

Žiga Lukšič 12:18 PM
not currently

Ohad 8:35 PM
So here’s the hack I added. **We should do something a bit more principled**

In `pervasives.eff`

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

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

```ml
| "Write" ->
| (match v with
|   | V.Tuple vs ->
|       let (file_name :: str :: _) = List.map V.to_str vs in
|       let file_handle = open_out_gen
|          [Open_wronly
|             ;Open_append
|             ;Open_creat
|             ;Open_text
|             ] 0o666 file_name in
|       Printf.fprintf file_handle "%s" str;
|       close_out file_handle;
|       top_handle (k V.unit_value)
|   )
```
An issue — difficult to cover all use cases

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

Žiga Lukšič 12:18 PM
not currently

Ohad 8:35 PM
So here's the hack I added. We should do something a bit more principled

In pervasives.eff:

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

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

```| "Write" ->
| (match v with
| V.Tuple vs ->
| let (file_name :: str :: _) = List.map V.to_str vs in
| let file_handle = open_out_gen
| [Open_wronly
| ;Open_append
| ;Open_creat
| ;Open_text
| ] 0666 file_name in
| Printf.fprintf file_handle "%s" str;
| close_out file_handle;
| top_handle (k V.unit_value)
| )```

This talk — a principled modular (co)algebraic approach!
A bigger issue — linearity or lack thereof
A bigger issue — **linearity** or lack thereof

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

- \[
\text{let } g s = \\
\text{let } fh = f s \text{ in } \text{fread } fh
\]
A bigger issue — **linearity** or lack thereof

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

- `let g s =`
  `let fh = f s in fread fh (* fh not open! *)`
A bigger issue — **linearity** or lack thereof

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

let g s =
  let fh = f s in fread fh (* fh not open! *)

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

  let h = handler
  | effect (fwrite fh s k) ↦ return ()

  let g' s =
  with h handle f ()
A bigger issue — **linearity** or lack thereof

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

  let g s =
  let fh = f s in fread fh (* fh not open ! *)

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

  let h = handler |
  | effect (fwrite fh s k) ↦ return ()

  let g' s =
  with h handle f () (* dangling fh ! *)
So, how could we solve these issues?

• We could try using existing PL techniques, e.g.,
  • Modules and abstraction, e.g., System.IO
    type IO a h Close :: Handle
  • Linear (and non-linear) types and effects
    linear type f handle effect Close ::
    linear handle -> unit
    linear effect Close :: handle -> unit
  • Handlers with initially and 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 **initially** and **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 **initially** and **finally** clauses

  • **Problem:** They don’t really capture the **essence of the problem**
Algebraic digression: What’s a comodel?

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

A model/algebra/handler $M$ of $\Sigma$ is given by $M = \langle M : \text{Set}, \{ \text{op}_M : A \times M \rightarrow B \times M \}_{\text{op} \in \Sigma} \rangle$.

A comodel/coalgebra/cohandler $W$ of $\Sigma$ is given by $W = \langle W : \text{Set}, \{ \text{op}_W : A \times W \rightarrow B \times W \}_{\text{op} \in \Sigma} \rangle$.

Intuitively, comodels describe evolution of worlds $w_1, w_2, w_3, \ldots$.

Operational semantics using a tensor of a model and a comodel (Plotkin & Power, Abou-Saleh & Pattinson).

Stateful runners of effectful programs (Uustalu).

Linear state-passing translation (Møgelberg and Staton).

Top-level behaviour of alg. effects in $\text{Eff}^\text{v2}$ (Bauer & Pretnar).
Algebraic digression: What’s a comodel?

- A **signature** $\Sigma$ is a set of operation symbols $op : A_{op} \rightsquigarrow B_{op}$
Algebraic digression: What’s a comodel?

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

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

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

Intuitively, comodels describe evolution of worlds $w_1, w_2, w_3, \ldots$

Operational semantics using a tensor of a model and a comodel (Plotkin & Power, Abou-Saleh & Pattinson)

Stateful runners of effectful programs (Uustalu)

Linear state-passing translation (Møgelberg and Staton)

Top-level behaviour of alg. effects in Eff² (Bauer & Pretnar)
Algebraic digression: What’s a 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

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

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

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

- Intuitively, comodels describe evolution of worlds $w_1, w_2, w_3, \ldots$
Algebraic digression: What’s a 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
  \[ \mathcal{M} = \langle M : \text{Set} , \{ \text{op}_M : A_{\text{op}} \times M^{B_{\text{op}}} \longrightarrow M \}_{\text{op} \in \Sigma} \rangle \]

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

- Intuitively, comodels describe **evolution of worlds** $w_1, w_2, w_3, \ldots$
  - Operational semantics using a tensor of a model and a comodel (Plotkin & Power, Abou-Saleh & Pattinson)
  - Stateful runners of effectful programs (Uustalu)
  - Linear state-passing translation (Møgelberg and Staton)
  - **Top-level behaviour** of alg. effects in $\text{EFF v2}$ (Bauer & Pretnar)
Back to the essence: What is it then?

- Let's look at Haskell's IO monad again.

A common informal 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
Back to the **essence**: What is it then?

- Let’s look at **Haskell**’s **IO monad** again
Back to the essence: What is it then?

- Let’s look at Haskell’s IO monad again
- A common informal 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

\[ (a, \text{RealWorld}) \rightarrow (b, \text{RealWorld}) \]
Back to the **essence**: What is it then?

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

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

as

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

which is the same as

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

- 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
Back to the **essence**: What is it 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**!

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

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

**Important**: co-operations (hClose) make a **promise to return**!
Towards a general programming abstraction
Towards a general programming abstraction

- \( \text{let } f (s : \text{string}) = (\ast \text{ in top level world } \ast) \)
  \( \text{using } \text{IO } \text{run} \)
  \( \text{let } fh = \text{fopen } "\text{foo.txt}" \text{ in} \)
  \( \text{fwrite } fh (s^s); \)
  \( \text{fclose } fh \quad (\ast \text{ in IO world } \ast) \)

Now \text{external world} explicit, but \text{dangling } fh \text{ etc still possible}
Towards a general programming abstraction

- \text{let } f \ (s: \text{string}) = \begin{array}{c}
  \text{(* in top level world *)} \\
  \text{using IO run} \\
  \text{let } fh = \text{fopen } "\text{foo.txt}" \text{ in} \\
  \text{fwrite } fh \ (s^s); \\
  \text{fclose } fh \quad \text{(* in IO world *)}
\end{array}

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

- \text{let } f \ (s: \text{string}) = \begin{array}{c}
  \text{(* in top level world *)} \\
  \text{using IO run} \\
  \text{let } fh = \text{fopen } "\text{foo.txt}" \text{ in} \\
  \text{fwrite } fh \ (s^s) \quad \text{(* in IO world *)} \\
  \text{finally } (\text{fclose } fh)
\end{array}

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

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

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

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

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

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

- For example

- **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

```
  IO
  ↙
  Fh
  ↘
  Fc
```

- \( Fh \) — "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 \( fh \)"
- \( Fc \) — "world that is blissfully unaware of \( fh \)"
Modular treatment of external worlds

• For example

Fh ←→ IO + CallStatistics
Fh ←→ IO
Fh ←→ Fc
Fh ←→ Fc

IO ↓

(outer world)

Fh — "world which consists of exactly one fh"
IO — "call fopen with foo.txt, store returned fh"
Fh — "call fclose with stored fh"
Fh — "world that is blissfully unaware of fh"
Modular treatment of external worlds

- For example

```
Pure  ➔  IO
      ➔  Fh
      ➔  IO + CallStatistics
      ➔  Fc

• 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”
• Fc — “world that is blissfully unaware of fh”
```
Modular treatment of external worlds

- For example

  Pure \rightarrow IO \rightarrow MLState \rightarrow (ext. world)

  Fh \rightarrow IO \rightarrow CallStatistics \rightarrow FPState \rightarrow (inner world)

  Fc \rightarrow (inner^2 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”
- Fc — “world that is blissfully unaware of 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"
- Fc — "world that is blissfully unaware of fh"
Modular treatment of external worlds

- For example

```
Pure ←→ IO ←→ MLState (ext. world)
```

```
Fh ←→ Fc (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"
```

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

```
Observation: IO ↔ Fh and other ↔ look a lot like lenses
```
Comodels as a gateway to the external world

• Running a program on a comodel (using external resources) using C (\(\ast\): Comodel (Sig, W) \(\ast\)) @ init (\(\ast\): W \(\ast\)) run c (\(\ast\): A \(\ast\)) finally @ (w : W) { return (x : A) \(\mapsto\) c fin (w, x) (\(\ast\): B \(\ast\)) (\(\ast\): B \(\ast\)) }

• Comodels are defined as follows C = \{ op (x : A) @ (w : W) \(\mapsto\) c op (x, w), (\(\ast\): B \(\ast\): W \(\ast\)) \ldots\} for all operations op : A \(\Rightarrow\) B in a given signature \(\Sigma\)
Comodels as a gateway to the external world

- **Running a program** on a comodel (using external resources)

```plaintext
using
C (∗ : Comodel(Sig,W) ∗) @ c_init (∗ : W ∗)
run
c (∗ : A ∗)
finally @ (w : W) {
  return (x : A) ↦ c_fin (w, x) (∗ : B ∗) }
(∗ : B ∗)
```

- **Comodels** are defined as follows

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

for all **operations** op : A → B in a given signature Σ
Focussing on a fragment of the external world
Focussing on a fragment of the external world

```csharp
let f (s : string) =
  using
    Fh @ (fopen_of_i o "foo.txt")
  run
    fwrite_of_fh (s ^ s)
  finally @ fh {
    return (x) ↦ fh (fclose_of_i o)
  }
```
Focussing on a fragment of the external world

```ocaml
let f (s : string) = (* in IO *)
  using
    Fh @ (fopen_of_io "foo.txt") (* in IO *)
  run
    fwrite_of_fh (s^s) (* in Fh *)
  finally @ fh {
    return(x) ↦ fclose_of_io fh }
  (* in IO *)
```
Focussing on a fragment of the external world

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

where

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

(* fread : (unit * W) → (string * W) *)
(* fwrite : (string * W) → (unit * W) * )
```
Modular treatment of worlds \((IO \leftrightarrow Fh \leftrightarrow Fc)\)
Modular treatment of worlds \((\text{IO} \leftrightarrow \text{Fh} \leftrightarrow \text{Fc})\)

```plaintext
let f (s: string) =
    using Fh @ (fopen_of_io "foo.txt")
    run
        using Fc @ (fread_of_fh ())
        run
            fwrite_of_fc (s^s)
        finally @ s {
            return(_) => fwrite_of_fh s }
    finally @ fh {
        return(_) => fclose_of_io fh }

where

Fc = { fwrite s @ s' => return((),s'^s) }
```
Modular treatment of worlds $(\text{IO} \leftrightarrow \text{Fh} \leftrightarrow \text{Fc})$

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

    using Fc @ (fread_of_fh ()) (* in Fh *)
    run

    fwrite_of_fc (s^s) (* in Fc *)
    finally @ s {
        return(_) \mapsto fwrite_of_fh s
    }

    finally @ fh {
        return(_) \mapsto fclose_of_io fh
    }

where

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

- More generally: comodels allow **transactions** and **sandboxing**
Tracking the world usage \((\text{IO} \leftrightarrow \text{IO} + \text{Stats})\)
Tracking the world usage \((\text{IO} \leftrightarrow \text{IO}+\text{Stats})\)

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

where

\[
\text{IO+Stats } = \quad (* W = \text{nat} *) \\
\quad \{ \text{fwrite } fh s @ c \mapsto \text{fwrite_of_io } fh s; \text{return } (( ),c+1), \\\n\quad \ldots \} \\
\]
Tracking the world usage (IO ←→ IO+Stats)

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

where

IO+Stats = (* W = nat *)
  \{ fwrite fh s @ c \mapsto fwrite_of_io fh s ;
      return ((),c+1),
    ...
  \}
```

- More generally: allows to slot in instrumentation/monitors
The external world can also be pure (Pure $\leftrightarrow$ Str)
The **external world** can also be **pure** \((\text{Pure} \leftrightarrow \text{Str})\)

\[
\begin{align*}
\text{let } f() = \\
\quad \text{using} \\
\quad \text{Str } @ (\text{return } "\text{some default initial value}" ) \\
\text{run} \\
\quad \ldots \\
\quad \text{let } s = \text{get}() \text{ in} \\
\quad \text{if } (s == "\text{foo}") \\
\quad \text{then } (\ldots ; \text{set } "\text{bar}" ; \ldots ) \\
\quad \text{else } (\ldots ) \\
\quad \ldots \\
\text{finally } @ _ { \{ \\
\quad \text{return}(x) \mapsto \text{return } x \} }
\end{align*}
\]

\[
\text{Str } = \\
\quad \{ \quad \text{get } _ { s } @ s \mapsto \text{return } (s,s) , \\
\quad \text{set } s @ _ { } \mapsto \text{return } ((),s) \quad \}
\]
The external world can also be pure \((\text{Pure} \leftrightarrow \text{Str})\)

```haskell
let f () = (* in Pure *)
    using
    Str @ (return "some default initial value")
run

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

\(\text{Str} = (* \ W = \text{string} *)\)

\[
\{ \text{get } _ @ s ↦ \text{return} (s,s) , \text{set } s @ _ ↦ \text{return} ((),s) \} \]

• Similar to ambient values (and ambient functions) in \text{KOKA}
More on ambient values/functions (Pure ←→ Amb)
More on ambient values/functions (Pure $\leftrightarrow$ Amb)

```plaintext
let f (s: string) =
    using (* with val amb = ... *)
    Amb @ (return "some default initial value")
    run
...

let amb = get () in
if (amb === "foo")
then (...; (* with val amb = ... *)
    using Amb @ ... run ... finally ...);
...
else (...)
...

finally @ _ { return(x) ↦ return x }
```

Amb = { get _ @ s ↦ return (s,s) }
```
More on **ambient values/functions** (Pure $\leftrightarrow$ Amb)

```plaintext
let f (s: string) =
  using  (* with val amb = ... *)
  Amb @ (return "some default initial value")
  run

  ...
  let amb = get () in
  if (amb == "foo")
    then (...;  (* with val amb = ... *)
      using Amb @ ... run ... finally ...);
    ...
  else (...)
  ...
  finally @ _ { return(x) ↦ return x }

Amb = { get _ @ s ↦ return (s, s) }
```

- **Amb. functions** by amb. function application as a co-operation
So what’s happening formally?
So what’s happening formally?

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

Types

\[ A, B, W ::= b | 1 | A \times B | 0 | A + B | A \Sigma - \rightarrow B \]

Interfaces (signatures) of external worlds

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

Computation terms (value terms are unsurprising)

\[ c ::= \text{return } v | \text{let } x = c_1 \text{ in } c_2 | \text{let rec } f x = c_1 \text{ in } c_2 | v_1 v_2 | \text{op} \ v_1 (x. c) | \text{using } C @ c_i \text{ run } c \text{ finally } @ w \{ \text{return } (x) \mapsto \cdots \} \]

- Comodels (cohandlers)

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

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

\[
A, B, W ::= b \mid 1 \mid A \times B \mid 0 \mid A + B \mid A \Sigma \to B
\]
So what’s happening formally?

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

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

- **Interfaces (signatures) of external worlds**

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

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

- Types

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

- Interfaces (signatures) of external worlds

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

- Computation terms (value terms are unsurprising)

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

\[
\mid v_1 \ v_2
\]

\[
\mid \text{op } v \ (x \cdot c)
\]

\[
\mid \text{using } C @ c_i \text{ run } c \text{ finally } @ w \{ \text{return}(x) \mapsto c_f \}
\]
So what’s happening **formally**?

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

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

- **Interfaces (signatures) of external worlds**

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

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

  \[
  c ::= \text{return } v \mid \text{let } x = c_1 \text{ in } c_2 \mid \text{let rec } f \ x = c_1 \text{ in } c_2 \\
  \mid v_1 \ v_2 \\
  \mid \text{op } v \ (x.c) \\
  \mid \text{using } C \ @ \ c_i \text{ run } c \text{ finally } @ w \ \{ \text{return} (x) \mapsto c_f \}
  \]

- **Comodels (cohandlers)**

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

- **Typing judgements**

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

- **Typing judgements**

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

- The two central *typing rules* are

\[
\begin{align*}
\Gamma &\vdash \text{C comodel of } \Sigma' \text{ with carrier } W_C \\
\Gamma &\vdash c_i : W_C \\
\Gamma &\vdash \Sigma' c : A \\
\Gamma, w : W_C, x : A &\vdash c_f : B \\
\Gamma &\vdash \text{using } C \ominus c_i \text{ run } c \text{ finally } \ominus w \{ \text{return}(x) \mapsto c_f \} : B
\end{align*}
\]

and

\[
\begin{align*}
\text{op} : A_{\text{op}} &\rightsquigarrow B_{\text{op}} \in \Sigma \\
\Gamma &\vdash v : A_{\text{op}} \\
\Gamma, x : B_{\text{op}} &\vdash \Sigma c : A \\
\Gamma &\vdash \text{op } v (x.c) : A
\end{align*}
\]
(Denotational) semantics (in $\omega$-cpos)
(Denotational) **semantics** (in $\omega$-cpos)

- **Term interpretation** looks very similar to **alg. effects**:
  
  $\left[ \Gamma \vdash v : A \right] : \left[ \Gamma \right] \rightarrow \left[ A \right] \quad \left[ \Gamma \vdash c : A \right] : \left[ \Gamma \right] \rightarrow T_{\Sigma_{\perp}} \left[ A \right]$

- **un-cohandled operations** wait for a suitable **external world**!
(Denotational) **semantics** (in ω-cpos)

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

  \[ \llbracket \Gamma \vdash v : A \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket \quad [\Gamma \vdash \Sigma c : A] : [\Gamma] \rightarrow T_{\Sigma \perp} [A] \]

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

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

  \[
  \Gamma \vdash \Sigma C \text{ comodel of } \Sigma' \text{ with carrier } W_C
  \]

  \[
  \Gamma \vdash \Sigma c_i : W_C \quad \Gamma \vdash \Sigma' c : A \quad \Gamma, w : W_C, x : A \vdash \Sigma c_f : B
  \]

  \[
  \Gamma \vdash \text{using } C \odot c_i \text{ run } c \text{ finally } @ w \{ \text{return}(x) \mapsto c_f \} : B
  \]

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

  \[
  \llbracket C \rrbracket \in \text{Comod}_{\Sigma'}(\text{Kleisli}(T_{\Sigma \perp}))
  \]

  \[
  \text{run\_on}_{[C]} : T_{\Sigma \perp} [A] \rightarrow (\llbracket W_C \rrbracket \rightarrow T_{\Sigma \perp} ([W_C] \times [A]))
  \]
Computational behaviour of \texttt{using ... run}
Computational behaviour of using . . . run

- Two semantically valid program equations

\[
\begin{align*}
\text{using } & C \odot c_i \ \text{run} \ (\text{return } \nu) \ \text{finally } \odot w \ \{ \ \text{return}(x) \mapsto c_f \ \} \\
= & \ \text{let } w' = c_i \ in \ c_f[w'/w, v/x]
\end{align*}
\]

\[
\begin{align*}
\text{using } & C \odot c_i \ \text{run} \ (\text{op } \nu \ (y.c)) \ \text{finally } \odot w \ \{ \ \text{return}(x) \mapsto c_f \ \} \\
= & \ \text{let } w' = c_i \ in \ \Bigg( \text{let } z = C_{\text{op}}[w'/w, v/x] \ in \ \Big( \ \text{match } z \ \text{with} \ \{ \ \langle y', w'' \rangle \mapsto \\
& \ \text{using } C \odot (\text{return } w'') \\
& \ \text{run } (c[y'/y]) \\
& \ \text{finally } \odot w \ \{ \ \text{return}(x) \mapsto c_f \ \} \ \Big) \ \Bigg)
\end{align*}
\]
What if the world doesn’t keep promises?
What if the **world** doesn’t keep promises?

- Recall that the **semantics of co-operations**

\[ \overline{\text{op}} : \llbracket A_{\text{op}} \rrbracket \times \llbracket W \rrbracket \longrightarrow T_{\Sigma_{\bot}}(\llbracket B_{\text{op}} \rrbracket \times \llbracket W \rrbracket) \]

ensures that the **world always comes back with an answer**
What if the world doesn’t keep promises?

- Recall that the semantics of co-operations

\[
\overline{\text{op}} : \llbracket A_{\text{op}} \rrbracket \times \llbracket W \rrbracket \longrightarrow T_{\Sigma_1}(\llbracket B_{\text{op}} \rrbracket \times \llbracket W \rrbracket)
\]

ensures that the world always comes back with an answer

- What if IO lost connection to the HDD where "foo.txt" was?
What if the world doesn’t keep promises?

• Recall that the semantics of co-operations

\[ \text{op} : [A_{op}] \times [W] \to T_{\Sigma}(B_{op} \times [W]) \]

ensures that the world always comes back with an answer

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

• Our solution: Allow the world to raise signals to talk back

\[
C = (\ast : A \times W \to T((B \times W) + S) \ast) \\
\{ \text{op} \times @w \mapsto \text{if } b \text{ then } (\ldots) \text{ else (raise } s) \}
\]
What if the **world** doesn’t keep promises?

- Recall that the **semantics of co-operations**

  \[ \overline{\text{op}} : \llbracket A_{\text{op}} \rrbracket \times \llbracket W \rrbracket \longrightarrow T_{\Sigma}(\llbracket B_{\text{op}} \rrbracket \times \llbracket W \rrbracket) \]

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

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

- **Our solution:** Allow the **world** to **raise signals** to talk back

  \[
  C = \left( \star : A \times W \rightarrow T((B \times W) + S) \star \right)
  \{ \text{op} \times @ w \mapsto \text{if b then } (\ldots) \text{ else } (\text{raise } s) \} \]

  **using** \( C @ c \_\text{init} \)

  **run** \( c \)

  **finally** \( @ w \{ \)

  **return** \( (x) \mapsto c \_\text{fin}(w, x) \),

  **signal** \( (s) \mapsto c \_\text{sig}(w, s) \} \)

  (** : A ! S \star **)

  (** : B ! S’ \star **)
What if the **world** doesn’t keep promises?

- **User-raised signals** can be handled locally (exceptional syntax)

```plaintext
try x = (raise s) in c unless { signal(s) ↦ c_sig }
```

...
What if the world doesn’t keep promises?

- **User-raised signals** can be handled locally (exceptional syntax)
  
  ```
  try \( x = \text{raise } s \) in c unless \{ signal(s) \mapsto c\_sig \}
  ```

- But **worldly signals** cannot be handled locally, e.g., consider
  
  ```
  using C @ c\_init
  run (try \( x = \text{raise } s \) in c unless \{(***) \ldots \})
  finally @ w {
      return(x) \mapsto c\_fin(w,x),
      signal(s) \mapsto c\_sig(w,s)  
  }  
  ```

vs

```  using C @ c\_init
run (try \( x = \text{op } v \) in c unless \{\ldots\})
finally @ w {
    return(x) \mapsto c\_fin(w,x),
    signal(s) \mapsto (***) c\_sig(w,s)  
  }  ```
What if the world doesn’t keep promises?
What if the world doesn’t keep promises?

- When a signal $s$ occurs in $\text{run } c$, control jumps to $c_{\text{sig}}(w,s)$

\[
\begin{align*}
\text{using } & C @ c_{\text{init}} \\
\text{run } & c \\
\text{finally } @ w \{ \\
\text{return} (x) & \mapsto c_{\text{fin}}(w,x), \text{signal}(s) \mapsto c_{\text{sig}}(w,s) \}
\end{align*}
\]

from which there is no automatic resume back to $\text{run } c$
What if the world doesn’t keep promises?

- When a signal $s$ occurs in $\text{run } c$, control jumps to $c_{\text{sig}}(w, s)$

```plaintext
using $C @ c_{\text{init}}$
run $c$
finally $@ w$
{
  return $(x) \mapsto c_{\text{fin}}(w, x)$, signal $(s) \mapsto c_{\text{sig}}(w, s)$
}
```

from which there is **no automatic resume** back to $\text{run } c$

- To resume $\text{run } c$, the **program and/or world** have to support it
What if the world doesn’t keep promises?

- When a signal $s$ occurs in \texttt{run c}, control jumps to $c_{\text{sig}}(w,s)$

```plaintext
using \texttt{C @ c\_init}
run \texttt{c}
finally @ w {
  return($x$) $\mapsto c_{\text{fin}}(w,x)$, signal($s$) $\mapsto c_{\text{sig}}(w,s)$}
```
from which there is \textbf{no automatic resume} back to \texttt{run c}

- To resume \texttt{run c}, the \texttt{program} and/or \texttt{world} have to support it

```plaintext
let rec \texttt{ctr\_printer i =}
  using \texttt{Out+Ctr @ (return i)}
run
    while(T) {\texttt{let j = get_c in print j; incr_c}}
finally @ k {
  return($x$) $\mapsto ...$
  signal($s$) $\mapsto$ print "foo"; \texttt{ctr\_printer k }}
```
What if the world doesn’t keep promises?

- When a signal \( s \) occurs in \( \text{run } c \), control jumps to \( \text{c\_sig}(w,s) \)

\[
\text{using } C @ \text{c\_init} \\
\text{run } c \\
\text{finally } @ w \{ \\
\quad \text{return}(x) \mapsto \text{c\_fin}(w,x), \text{signal}(s) \mapsto \text{c\_sig}(w,s) \}
\]

from which there is no automatic resume back to \( \text{run } c \)

- To resume \( \text{run } c \), the program and/or world have to support it

\[
\text{let rec } \text{ctr\_printer } i = \\
\quad \text{using } \text{Out+Ctr} @ (\text{return } i) \\
\quad \text{run} \\
\quad \quad \text{while}(T) \{ \text{let } j = \text{get\_c in print } j; \text{incr\_c} \} \\
\quad \text{finally } @ k \{ \\
\quad \quad \text{return}(x) \mapsto \ldots, \\
\quad \quad \text{signal}(s) \mapsto \text{print } "\text{foo}"; \text{ctr\_printer } k \}
\]

- World-based: could store a trace so as to replay “old” co-ops
What about **alg. effects** and **handlers**?
What about alg. effects and handlers?

- In the following

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

it is natural to want that

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

- to escape, have to use the **co-operations** of the **external world**
What about **alg. effects** and **handlers**?

- In the following
  ```c
  using  C @ c_init
  run  c
  finally  @ w { return (x) ↦ c_fin (w, x) , . . . }
  ```
  it is natural to want that
  - **algebraic operations** (in the sense of \( \text{Eff} \)) are allowed in \( c \), but they must not be allowed to escape \( \text{run} \)
  - to escape, have to use the **co-operations** of the **external world**
  - the **continuations of handlers** in \( c \) are delimited by \( \text{run} \)
  - so that we ensure that **finally** block is **definitely reached**
What about alg. effects and handlers?

- In the following

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

it is natural to want that

- **algebraic operations** (in the sense of \(\text{EFF}\)) are allowed in \(c\), but they must not be allowed to escape \(\text{run}\)
- to escape, have to use the **co-operations** of the **external world**
- the **continuations of handlers** in \(c\) are delimited by \(\text{run}\)
- so that we ensure that **finally** block is **definitely reached**

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

• Comodels as a gateway for interacting with the external world

• System.IO, Koka’s initially & finally, Python’s with, . . .

• Promising examples: sandboxing, instrumentation, monitors, . . .

• Comodels and init.-fin. lenses admit natural combinators

• Prototypes: a library in Haskell, and a small language Coop

• Can also be a basis for FFI, e.g., in Coop (and future Eff)

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

\[
\text{external f: } A \times W \text{ top-level } \rightarrow B \times W \text{ top-level } \in \text{top-level-comodel}
\]

• For the future: interface polymorphism, linear typing, . . .
Conclusions

- **Comodels** as a *gateway* for interacting with the *external world*
- System.IO, *Koka’s* `initially` & `finally`, *Python’s* `with`, …
- **Promising examples**: sandboxing, instrumentation, monitors, …
- Comodels and init.-fin. lenses admit **natural combinators**
Conclusions

- **Comodels** as a **gateway** for interacting with the **external world**

- System.IO, **Koka**'s **initially** & **finally**, Python's **with**, ...  

- **Promising examples**: sandboxing, instrumentation, monitors, ...  

- Comodels and **init.-fin. lenses admit natural combiners**  

- Prototypes: a **library** in **Haskell**, and a **small language** **Coop**  

- Can also be a **basis for FFI**, e.g., in **Coop** (and future **Eff**)  

\[
f : A \rightarrow B \in OCAML
\]

\[
\text{external } f : A \times W_{\text{top-level}} \rightarrow B \times W_{\text{top-level}} \in \text{top-level-comodel}
\]
Conclusions

- **Comodels** as a **gateway** for interacting with the **external world**
- **System.IO**, **Koka’s initially & finally**, **Python’s with**, . . .
- **Promising examples**: sandboxing, instrumentation, monitors, . . .
- Comodels and init.-fin. lenses admit **natural combinators**
- Prototypes: a **library** in **Haskell**, and a **small language** **Coop**
- Can also be a **basis for FFI**, e.g., in **Coop** (and future **Eff**)

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

\[ \text{external } f : A \times W_{\text{top-level}} \rightarrow B \times W_{\text{top-level}} \in \text{top-level-comodel} \]

- **For the future**: **interface polymorphism, linear typing**, . . .