

# **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

# 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)))
```

# 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
```

```
(*: int → a*int!{ } *)
```

```
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 >>= (\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 ())
```

- 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:

```
| "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
      ] 0o666 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

- ```
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
```

# 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?**

# 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
```

```
hClose :: Handle → IO ()
```

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

```
linear type fhandle
```

```
effect FClose : (linear fhandle) → unit
```

```
linear effect FClose : fhandle → unit
```

- Handlers with **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`

```
type IO a
```

```
hClose :: Handle → IO ()
```

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

```
linear type fhandle
```

```
effect FClose : (linear fhandle) → unit
```

```
linear effect FClose : fhandle → unit
```

- Handlers with **initially** and **finally** clauses
- **Problem:** They don't really capture the **essence of the problem**

**Algebraic digression: What's a **comodel**?**

# Algebraic digression: What's a **comodel**?

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

# 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

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

# 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

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

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

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

- Intutively, comodels describe **evolution of worlds**  $w_1, w_2, w_3, \dots$



# 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

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

- 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 worlds**  $w_1, w_2, w_3, \dots$ 
  - Operational semantics using a tensor of a model and a comodel  
(Plotkin & Power, Abou-Saleh & Pattinson)
  - Stateful runners of effectful programs (Uustalu)
  - Linear state-passing translation (Møgelberg and Staton)
  - Top-level behaviour of alg. effects in EFF v2 (Bauer & Pretnar)

**Back to the **essence**: What is it then?**

# 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 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 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
- 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` :  $(\text{Handle}, \text{RealWorld}) \rightarrow (\text{String}, \text{RealWorld})$

`hClose` :  $(\text{Handle}, \text{RealWorld}) \rightarrow ((), \text{RealWorld})$

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

# **Towards a general programming abstraction**

# 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**



# 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`

# 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**

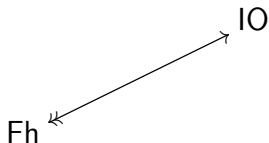
- ```
let f (s:string) =      (* in top level world *)
    using IO run
    let fh = fopen "foo.txt" in
    fwrite fh (s^s)      (* in IO world *)
    finally (fclose fh)
```

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

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

# Modular treatment of external worlds

- For example



(ext. world)

(inner world)

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

# Modular treatment of external worlds

- For example



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

# Modular treatment of external worlds

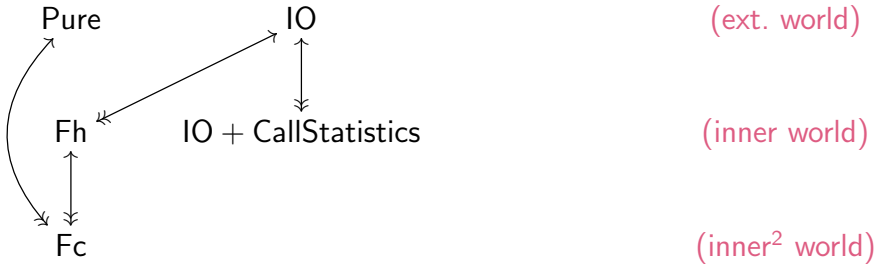
- For example



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

# Modular treatment of external worlds

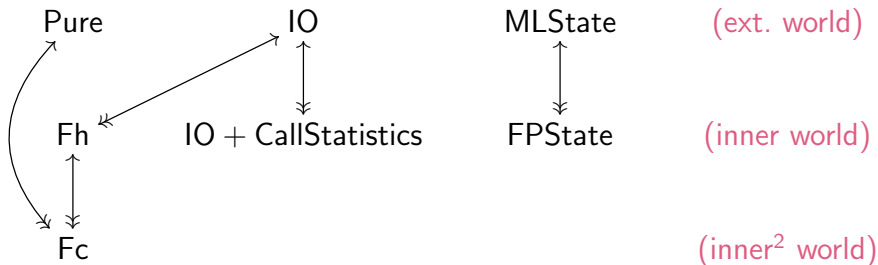
- For example



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

# Modular treatment of external worlds

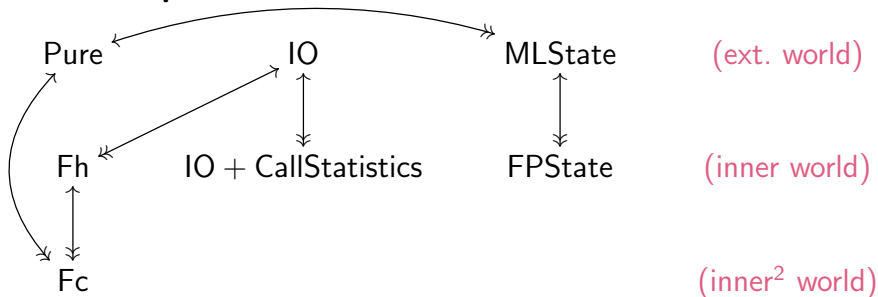
- For example



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

# Modular treatment of external worlds

- For example

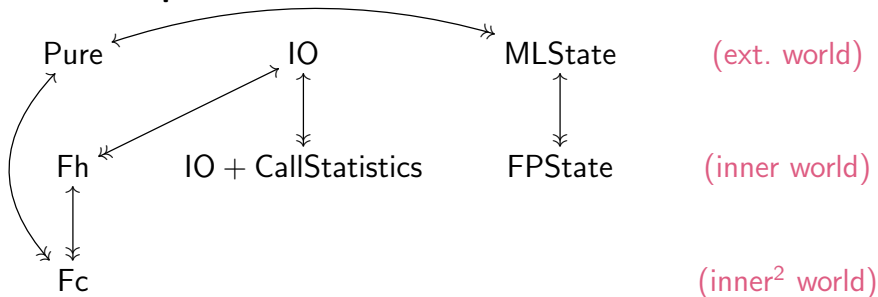


- Fh — “**world** which consists of **exactly one** fh”
- IO  $\longrightarrow$  Fh — “call `fopen` with `foo.txt`, store returned `fh`”
- Fh  $\longrightarrow$  IO — “call `fclose` with stored `fh`”
- Fc — “**world** that is **blissfully unaware** of `fh`”



# Modular treatment of external worlds

- For example



- Fh — “**world** which consists of **exactly one** fh”
- IO  $\longrightarrow$  Fh — “call `fopen` with `foo.txt`, store returned `fh`”
- Fh  $\longrightarrow$  IO — “call `fclose` with stored `fh`”
- Fc — “**world** that is **blissfully unaware** of `fh`”
- Observation:** IO  $\longleftrightarrow$  Fh and other  $\longleftrightarrow$  look a lot like **lenses**

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

## Comodels as a gateway to the external world

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

```
using
  C (* : Comodel(Sig,W) *) @ c_init (* : W *)
run
  c (* : A *)
finally @ (w:W) {
  return (x:A)  $\mapsto$  c_fin(w,x) (* : B *) } (* : B *)
```

- Comodels are defined as follows

```
C =
{
  op (x:A) @ (w:W)  $\mapsto$  c_op(x,w),   (* : B * W *)
  ...
}
```

for all **operations**  $\text{op} : A \rightsquigarrow B$  in a given signature  $\Sigma$

**Focussing on a fragment of the external world**

## Focussing on a fragment of the external world

```
let f (s:string) =  
  using  
    Fh @ (fopen_of_io "foo.txt")  
  run  
    fwrite_of_fh (s^s)  
  finally @ fh {  
    return(x)  $\mapsto$  fclose_of_io fh }
```

## Focussing on a fragment of the external world

```
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)  $\mapsto$  fclose_of_io fh } (* in IO *)
```

## Focussing on a fragment of the external world

```
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** ( $IO \longleftrightarrow Fh \longleftrightarrow Fc$ )



## Modular treatment of worlds ( $IO \longleftrightarrow Fh \longleftrightarrow Fc$ )

```
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(_) ↦ fwrite_of_fh s }

  finally @ fh {
    return(_) ↦ fclose_of_io fh }
```

where

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

## Modular treatment of worlds ( $IO \longleftrightarrow Fh \longleftrightarrow Fc$ )

```
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(_) ↦ fwrite_of_fh s }

  finally @ fh {
    return(_) ↦ fclose_of_io fh }
```

where

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

- **More generally:** comodels allow **transactions** and **sandboxing**

Tracking the world usage ( $IO \longleftrightarrow IO + \text{Stats}$ )

## Tracking the world usage ( $\text{IO} \longleftrightarrow \text{IO} + \text{Stats}$ )

```
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(_) ↦
      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 ↦ fwrite_of_io fh s;
    return ((),c+1),
    ... }
```

## Tracking the world usage ( $\text{IO} \longleftrightarrow \text{IO} + \text{Stats}$ )

```
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(_) ↦
      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 ↦ fwrite_of_io fh s;
    return ((),c+1),
    ... }
```

- More generally: allows to slot in instrumentation/monitors

The **external world** can also be **pure** (Pure  $\longleftrightarrow$  Str)

The **external world** can also be **pure** ( $\text{Pure} \longleftrightarrow \text{Str}$ )

```
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)  $\mapsto$  return x }
```

```
Str = (* W = string *)
{ get _ @ s  $\mapsto$  return (s, s) ,
  set s @ _  $\mapsto$  return ((), s) }
```

The **external world** can also be **pure** ( $\text{Pure} \longleftrightarrow \text{Str}$ )

```
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)  $\mapsto$  return x }
```

```
Str = (* W = string *)
{ get _ @ s  $\mapsto$  return (s, s) ,
  set s @ _  $\mapsto$  return (( ), s) }
```

- Similar to **ambient values** (and **ambient functions**) in KOKA



More on **ambient values/functions** ( $\text{Pure} \longleftrightarrow \text{Amb}$ )

## More on **ambient values/functions** ( $\text{Pure} \longleftrightarrow \text{Amb}$ )

```
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)  $\mapsto$  return x }
```

```
Amb = { get _ @ s  $\mapsto$  return (s, s) }
```

## More on **ambient values/functions** ( $\text{Pure} \longleftrightarrow \text{Amb}$ )

```
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)  $\mapsto$  return x }
```

$\text{Amb} = \{ \text{get } _ @ s \mapsto \text{return } (s, s) \}$

- **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  $\Rightarrow$  wait a few slides)

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

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

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

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

- Core calculus for comodels (wo/ handlers  $\Rightarrow$  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**

$$\Sigma ::= \{ \text{op}_1 : A_1 \rightsquigarrow B_1, \dots, \text{op}_n : A_n \rightsquigarrow B_n \}$$

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

- Core calculus for comodels (wo/ handlers  $\Rightarrow$  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**

$$\Sigma ::= \{ \text{op}_1 : A_1 \rightsquigarrow B_1, \dots, \text{op}_n : A_n \rightsquigarrow B_n \}$$

- Computation terms** (value terms are unsurprising)

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



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

- Core calculus for comodels (wo/ handlers  $\Rightarrow$  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**

$$\Sigma ::= \{ \text{op}_1 : A_1 \rightsquigarrow B_1, \dots, \text{op}_n : A_n \rightsquigarrow B_n \}$$

- Computation terms** (value terms are unsurprising)

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

- Comodels (cohandlers)**

$$C ::= \{ \overline{\text{op}}_1 \ x \ @ \ w \mapsto c_1, \dots, \overline{\text{op}}_n \ x \ @ \ w \mapsto c_n \}$$

So what's happening **formally**?

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

- **Typing judgements**

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

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

- **Typing judgements**

$$\Gamma \vdash v : A$$

$$\Gamma \vdash^{\Sigma} c : A$$

- The two central **typing rules** are

$\Gamma \vdash^{\Sigma} C$  comodel of  $\Sigma'$  with carrier  $W_C$

$$\Gamma \vdash^{\Sigma} c_i : W_C \quad \Gamma \vdash^{\Sigma'} c : A \quad \Gamma, w : W_C, x : A \vdash^{\Sigma} c_f : B$$

$$\Gamma \vdash^{\Sigma} \text{using } C @ c_i \text{ run } c \text{ finally } @ w \{ \text{return}(x) \mapsto c_f \} : B$$

and

$$\frac{\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}} \in \Sigma \quad \Gamma \vdash v : A_{\text{op}} \quad \Gamma, x : B_{\text{op}} \vdash^{\Sigma} c : A}{\Gamma \vdash^{\Sigma} \text{op } v (x.c) : A}$$

**(Denotational) semantics (in  $\omega$ -cpos)**

# (Denotational) semantics (in $\omega$ -cpos)

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

$$\llbracket \Gamma \vdash v : A \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket \qquad \llbracket \Gamma \stackrel{\Sigma}{\vdash} c : A \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow T_{\Sigma_{\perp}} \llbracket A \rrbracket$$

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

# (Denotational) semantics (in $\omega$ -cpos)

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

$$\llbracket \Gamma \vdash v : A \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket \quad \llbracket \Gamma \stackrel{\Sigma}{\vdash} c : A \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow T_{\Sigma_{\perp}} \llbracket A \rrbracket$$

- **un-cohandled operations** **wait for a suitable external world!**
- The interesting part is the **interpretation of using ... run**

$\Gamma \stackrel{\Sigma'}{\vdash} C$  comodel of  $\Sigma'$  with carrier  $W_C$

$$\Gamma \stackrel{\Sigma}{\vdash} c_i : W_C \quad \Gamma \stackrel{\Sigma'}{\vdash} c : A \quad \Gamma, w : W_C, x : A \stackrel{\Sigma}{\vdash} c_f : B$$

$$\Gamma \stackrel{\Sigma}{\vdash} \mathbf{using} \ C @ c_i \mathbf{run} \ c \mathbf{finally} @ w \{ \mathbf{return}(x) \mapsto c_f \} : B$$

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

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

$$\text{run\_on}_{\llbracket C \rrbracket} : T_{\Sigma'_{\perp}} \llbracket A \rrbracket \longrightarrow \left( \llbracket W_C \rrbracket \rightarrow T_{\Sigma_{\perp}}(\llbracket W_C \rrbracket \times \llbracket A \rrbracket) \right)$$

**Computational behaviour of `using ... run`**



# Computational behaviour of **using ... run**

- Two semantically valid program equations

$$\begin{aligned} & \text{using } C @ c_i \text{ run } (\text{return } v) \text{ finally } @ w \{ \text{return}(x) \mapsto c_f \} \\ &= \\ & \text{let } w' = c_i \text{ in } c_f[w'/w, v/x] \end{aligned}$$
$$\begin{aligned} & \text{using } C @ c_i \text{ run } (\text{op } v(y.c)) \text{ finally } @ w \{ \text{return}(x) \mapsto c_f \} \\ &= \\ & \text{let } w' = c_i \text{ in } \left( \begin{aligned} & \text{let } z = C_{\text{op}}[w'/w, v/x] \text{ in } \left( \begin{aligned} & \text{match } z \text{ with } \{ \langle y', w'' \rangle \mapsto \\ & \quad \text{using } C @ (\text{return } w'') \\ & \quad \text{run } (c[y'/y]) \\ & \quad \text{finally } @ w \{ \text{return}(x) \mapsto c_f \} \} \} \right) \end{aligned} \right) \end{aligned}$$

What if the **world** doesn't keep promises?

# What if the **world** doesn't keep promises?

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

$$\overline{\text{op}} : \llbracket A_{\text{op}} \rrbracket \times \llbracket W \rrbracket \longrightarrow T_{\Sigma_{\perp}}(\llbracket B_{\text{op}} \rrbracket \times \llbracket W \rrbracket)$$

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**

$$\overline{\text{op}} : \llbracket A_{\text{op}} \rrbracket \times \llbracket W \rrbracket \longrightarrow T_{\Sigma_{\perp}}(\llbracket B_{\text{op}} \rrbracket \times \llbracket W \rrbracket)$$

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**

$$\overline{\text{op}} : \llbracket A_{\text{op}} \rrbracket \times \llbracket W \rrbracket \longrightarrow T_{\Sigma_{\perp}}(\llbracket B_{\text{op}} \rrbracket \times \llbracket W \rrbracket)$$

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 = (* : A x W → T((B x W) + S) *)  
{ op x @ w ↦ if b then (...) else (raise s) }
```

# What if the **world** doesn't keep promises?

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

$$\overline{\text{op}} : \llbracket A_{\text{op}} \rrbracket \times \llbracket W \rrbracket \longrightarrow T_{\Sigma_{\perp}}(\llbracket B_{\text{op}} \rrbracket \times \llbracket W \rrbracket)$$

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 = (* : A x W → T((B x W) + S) *)  
{ op x @ w ↦ if b then (...) else (raise s) }
```

```
using C @ c_init  
run c (* : A ! S *)  
finally @ w {  
  return(x) ↦ c_fin(w,x), (* : B ! S' *)  
  signal(s) ↦ c_sig(w,s) } (* : B ! S' *)
```

# What if the **world** doesn't keep promises?

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

```
try x = (raise s) in c unless {signal(s)  $\mapsto$  c_sig}
```

# What if the **world** doesn't keep promises?

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

```
try x = (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 = (raise s) in c unless {(**) ...})  
finally @ w {  
  return(x)  $\mapsto$  c_fin(w,x),  
  signal(s)  $\mapsto$  c_sig(w,s) }
```

vs

```
using C @ c_init  
run (try x = (op v) in c unless {...})  
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 `run c`, control jumps to `c_sig(w,s)`

```
using C @ c_init
run c
finally @ w {
  return(x)  $\mapsto$  c_fin(w,x), signal(s)  $\mapsto$  c_sig(w,s) }
```

from which there is **no automatic resume** back to `run c`

# What if the **world** doesn't keep promises?

- When a signal `s` occurs in `run c`, control jumps to `c_sig(w,s)`

```
using C @ c_init
run c
finally @ w {
  return(x)  $\mapsto$  c_fin(w,x), signal(s)  $\mapsto$  c_sig(w,s) }
```

from which there is **no automatic resume** back to `run c`

- To resume `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 `run c`, control jumps to `c_sig(w,s)`

```
using C @ c_init
run c
finally @ w {
  return(x)  $\mapsto$  c_fin(w,x), signal(s)  $\mapsto$  c_sig(w,s) }
```

from which there is **no automatic resume** back to `run c`

- To resume `run c`, the **program** and/or **world** have to support it

```
let rec ctr_printer i =
  using Out+Ctr @ (return i)
  run
    while(T) { let j = get_c in print j; incr_c }
  finally @ k {
    return(x)  $\mapsto$  ...,
    signal(s)  $\mapsto$  print "foo"; ctr_printer k }
```

# What if the **world** doesn't keep promises?

- When a signal `s` occurs in `run c`, control jumps to `c_sig(w,s)`

```
using C @ c_init
run c
finally @ w {
  return(x)  $\mapsto$  c_fin(w,x), signal(s)  $\mapsto$  c_sig(w,s) }
```

from which there is **no automatic resume** back to `run c`

- To resume `run c`, the **program** and/or **world** have to support it

```
let rec ctr_printer i =
  using Out+Ctr @ (return i)
  run
    while(T) { let j = get_c in print j; incr_c }
  finally @ k {
    return(x)  $\mapsto$  ...,
    signal(s)  $\mapsto$  print "foo"; 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)  $\mapsto$  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 `run`
- to escape, have to use the **co-operations** of the **external world**

# What about **alg. effects** and **handlers**?

- In the following

```
using C @ c_init  
run c  
finally @ w { return(x)  $\mapsto$  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 `run`
- to escape, have to use the **co-operations** of the **external world**
- the **continuations of handlers** in `c` are delimited by `run`
- so that we ensure that `finally` block is **definitely reached**



# What about **alg. effects** and **handlers**?

- In the following

```
using C @ c_init
run c
finally @ w { return(x)  $\mapsto$  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 `run`
- to escape, have to use the **co-operations** of the **external world**
- the **continuations of handlers** in `c` are delimited by `run`
- so that we ensure that `finally` block is **definitely reached**
- Where do **multi-handlers** fit? Co-operating handlers-cohandlers?

# Conclusions

# 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 combinators**
- Two prototypes: **library** in HASKELL, and **toy language** COOP
- Can also be a **basis for FFI**, e.g., in COOP (and future EFF)

$$f : A \longrightarrow B \in \text{OCAML}$$

---

$$\text{external } f : A \times W_{\text{top-level}} \longrightarrow B \times W_{\text{top-level}} \in \text{top-level-comodel}$$

# 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**
- Two prototypes: **library** in HASKELL, and **toy language** COOP
- Can also be a **basis for FFI**, e.g., in COOP (and future EFF)

$$f : A \longrightarrow B \in \text{OCAML}$$

$$\text{external } f : A \times W_{\text{top-level}} \longrightarrow B \times W_{\text{top-level}} \in \text{top-level-comodel}$$

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