

# Interacting with the external world using comodels (aka runners)

Danel Ahman

(joint work with Andrej Bauer)

University of Ljubljana, Slovenia

Gallinette seminar, Nantes, 14.10.2019

# The plan

- **Computational effects** and **external resources** in PL
- **Runners** – a natural model for **top-level runtime**
- **T-runners** – for also modelling **non-top-level runtimes**
- Turning **T**-runners into a **useful programming construct**
- Some **programming examples**
- Some **implementation details**

**Computational effects**  
**and**  
**external resources**

# Computational effects in PL

# Computational effects in PL

- 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 PL

- 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 state_handler handle (perform (Put 42); c ())
```

# Computational effects in PL

- 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 state_handler 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 resources in PL



# External resources in PL

- 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/eval.ml *)  
let rec top_handle op =  
  match op with  
  | ...
```

# External resources in PL

- 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/eval.ml *)  
let rec top_handle op =  
  match op with  
  | ...
```

but there are some issues with that approach ...

**First issue**

# First issue

- Difficult to cover all possible use cases
  - **external resources hard-coded** into the top-level runtime
  - **non-trivial to change** what's available and how it's implemented

# First issue

- Difficult to cover all possible use cases
  - **external resources hard-coded** into the top-level runtime
  - **non-trivial to change** what's available and how it's implemented



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)
  )
```

# First issue

- Difficult to cover all possible use cases
  - **external resources hard-coded** into the top-level runtime
  - **non-trivial to change** what's available and how it's implemented

 **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!**

## Second issue

# Second issue

- **Lack of linearity** for external resources

```
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
```



# Second issue

- **Lack of linearity** for external resources

```
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 ! \*)

# Second issue

- **Lack of linearity** for external resources

```
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 ! \*)

- We shall address these kinds of issues **indirectly**,
  - by **not** introducing a linear typing discipline
  - but instead make it convenient to **hide** external resources

## Third issue

# Third issue

- **Excessive generality** of effect handlers

```
let f (s:String) =  
  let fh = fopen "foo.txt" in  
  fwrite (fh,s^s);  
  fclose fh  
  
let h = handler { fwrite (fh,s) k → return () }  
  
let f' s = handle (f "bar") with h
```

# Third issue

- **Excessive generality** of effect handlers

```
let f (s:String) =  
  let fh = fopen "foo.txt" in  
  fwrite (fh,s^s);  
  fclose fh  
  
let h = handler { fwrite (fh,s) k → return () }  
  
let f' s = handle (f "bar") with h
```

where misuse of external resources can also be **purely accidental**

```
let g (s:String) =  
  let fh = fopen "foo.txt" in  
  let b = choose () in  
  if b then (fwrite (fh,s)) else (fwrite (fh,s^s));  
  fclose fh  
  
let nondet_handler =  
  handler { choose () k → return (k true ++ k false) }
```

# Third issue

- **Excessive generality** of effect handlers

```
let f (s:String) =  
  let fh = fopen "foo.txt" in  
  fwrite (fh,s^s);  
  fclose fh  
  
let h = handler { fwrite (fh,s) k → return () }  
  
let f' s = handle (f "bar") with h
```

- We shall address these kinds of issues **directly**,
  - by proposing a **restricted form** of handlers for resources
  - that support **controlled initialisation** and **finalisation**,
  - and **limit** how general handlers can be used

**Runners** enter the spotlight

# A natural model of **top-level runtime**



# A natural model of **top-level runtime**

- Given a **signature**<sup>1</sup>  $\Sigma$  of operation symbols ( $A_{\text{op}}, B_{\text{op}}$  countable)

$$\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$$

a **runner**<sup>2</sup>  $\mathcal{R}$  for  $\Sigma$  is given by a carrier  $|\mathcal{R}|$  and co-operations

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \times |\mathcal{R}| \longrightarrow B_{\text{op}} \times |\mathcal{R}| \right)_{\text{op} \in \Sigma}$$

---

<sup>1</sup>We consider runners for signatures, but the work generalises to alg. theories.

<sup>2</sup>In the literature also known as **comodels** for  $\Sigma$  (or an alg. theory).

# A natural model of **top-level runtime**

- Given a **signature**<sup>1</sup>  $\Sigma$  of operation symbols ( $A_{\text{op}}, B_{\text{op}}$  countable)

$$\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$$

a **runner**<sup>2</sup>  $\mathcal{R}$  for  $\Sigma$  is given by a carrier  $|\mathcal{R}|$  and co-operations

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \times |\mathcal{R}| \longrightarrow B_{\text{op}} \times |\mathcal{R}| \right)_{\text{op} \in \Sigma}$$

- For example, a natural runner  $\mathcal{R}$  for **S-valued state**

$$\text{get} : \mathbb{1} \rightsquigarrow S \quad \text{set} : S \rightsquigarrow \mathbb{1}$$

is given by

$$|\mathcal{R}| \stackrel{\text{def}}{=} S \quad \overline{\text{get}}_{\mathcal{R}}(\star, s) \stackrel{\text{def}}{=} (s, s) \quad \overline{\text{set}}_{\mathcal{R}}(s, s) \stackrel{\text{def}}{=} (\star, s)$$

---

<sup>1</sup>We consider runners for signatures, but the work generalises to alg. theories.

<sup>2</sup>In the literature also known as **comodels** for  $\Sigma$  (or an alg. theory).

# A natural model of **top-level runtime** ctd.

- Runners/comodels have been used for
  - **operational semantics** using tensors of models and comodels  
[Plotkin and Power '08]  
and
  - **stateful running** of algebraic effects [Uustalu '15]
  - **linear-use state-passing translation**  
[Møgelberg and Staton '11, '14]

# A natural model of **top-level runtime** ctd.

- Runners/comodels have been used for
  - **operational semantics** using tensors of models and comodels [Plotkin and Power '08]  
and
  - **stateful running** of algebraic effects [Uustalu '15]
  - **linear-use state-passing translation** [Møgelberg and Staton '11, '14]
- The latter explicitly rely on one-to-one correspondence between
  - **runners**  $\mathcal{R}$  and
  - **monad morphisms**<sup>3</sup>  $r : \mathbf{Free}_\Sigma(-) \longrightarrow \mathbf{St}_{|\mathcal{R}|}$

where

$$\mathbf{St}_C X \stackrel{\text{def}}{=} C \Rightarrow X \times C$$

---

<sup>3</sup> $\mathbf{Free}_\Sigma(X)$  is the free monad ind. defined with leaves  $\text{val } x$  and nodes  $\text{op}(a, \kappa)$ .

# A natural model of **top-level runtime** ctd.

- For our purposes, we see runners

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \times |\mathcal{R}| \longrightarrow B_{\text{op}} \times |\mathcal{R}| \right)_{\text{op} \in \Sigma}$$

as describing how operations affect **runtime configurations**  $|\mathcal{R}|$

# A natural model of **top-level runtime** ctd.

- For our purposes, we see runners

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \times |\mathcal{R}| \longrightarrow B_{\text{op}} \times |\mathcal{R}| \right)_{\text{op} \in \Sigma}$$

as describing how operations affect **runtime configurations**  $|\mathcal{R}|$

- But what if this runtime is not **the** runtime?
  - hardware vs OS
  - OS vs VMs
  - VMs vs sandboxes

# A natural model of **top-level runtime** ctd.

- For our purposes, we see runners

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \times |\mathcal{R}| \longrightarrow B_{\text{op}} \times |\mathcal{R}| \right)_{\text{op} \in \Sigma}$$

as describing how operations affect **runtime configurations**  $|\mathcal{R}|$

- But what if this runtime is not **the** runtime?
  - hardware vs OS
  - OS vs VMs
  - VMs vs sandboxes
- Unfortunately, runners, as defined above, are **not readily able to**
  - use **external resources**
  - **signal failure** caused by unavoidable circumstances

# A natural model of **top-level runtime** ctd.

- For our purposes, we see runners

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \times |\mathcal{R}| \longrightarrow B_{\text{op}} \times |\mathcal{R}| \right)_{\text{op} \in \Sigma}$$

as describing how operations affect **runtime configurations**  $|\mathcal{R}|$

- But what if this runtime is not **the** runtime?
  - hardware vs OS
  - OS vs VMs
  - VMs vs sandboxes
- Unfortunately, runners, as defined above, are **not readily able to**
  - use **external resources**
  - **signal failure** caused by unavoidable circumstances
- But is there a **useful generalisation** that would achieve this?



# Effectful runners for modular top-levels

# Effectful runners for modular top-levels

- Møgelberg and Staton usefully observed that a **runner**  $\mathcal{R}$  is equivalently simply a family of **generic effects** for  $\mathbf{St}_{|\mathcal{R}|}$ , i.e.,

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{St}_{|\mathcal{R}|} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

# Effectful runners for modular top-levels

- Møgelberg and Staton usefully observed that a **runner**  $\mathcal{R}$  is equivalently simply a family of **generic effects** for  $\mathbf{St}_{|\mathcal{R}|}$ , i.e.,

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{St}_{|\mathcal{R}|} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

- Building on this, we define a **T-runner**  $\mathcal{R}$  for  $\Sigma$  to be given by

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{T} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

# Effectful runners for modular top-levels

- Møgelberg and Staton usefully observed that a **runner**  $\mathcal{R}$  is equivalently simply a family of **generic effects** for  $\mathbf{St}_{|\mathcal{R}|}$ , i.e.,

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{St}_{|\mathcal{R}|} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

- Building on this, we define a **T-runner**  $\mathcal{R}$  for  $\Sigma$  to be given by

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{T} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

- The one-to-one correspondence with **monad morphisms**

$$r : \mathbf{Free}_{\Sigma}(-) \longrightarrow \mathbf{T}$$

now simply amounts to the **univ. property of free models**, e.g.,

$$r_X(\text{val } x) = \eta x \qquad r_X(\text{op}(a, \kappa)) = (r_X \circ \kappa)^{\dagger}(\overline{\text{op}}_{\mathcal{R}} a)$$

# Effectful runners for modular top-levels

- Møgelberg and Staton usefully observed that a **runner**  $\mathcal{R}$  is equivalently simply a family of **generic effects** for  $\mathbf{St}_{|\mathcal{R}|}$ , i.e.,

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{St}_{|\mathcal{R}|} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

- Building on this, we define a **T-runner**  $\mathcal{R}$  for  $\Sigma$  to be given by

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{T} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

- The one-to-one correspondence with **monad morphisms**

$$r : \mathbf{Free}_{\Sigma}(-) \longrightarrow \mathbf{T}$$

now simply amounts to the **univ. property of free models**, e.g.,

$$r_X(\text{val } x) = \eta x \qquad r_X(\text{op}(a, \kappa)) = (r_X \circ \kappa)^{\dagger}(\overline{\text{op}}_{\mathcal{R}} a)$$

- Observe that  $\kappa$  appears in a **tail call position** on the right!

## Effectful runners for modular top-levels ctd.

- What would be a **useful class of monads  $\mathbf{T}$**  to use?

# Effectful runners for modular top-levels ctd.

- What would be a **useful class of monads  $\mathbf{T}$**  to use?
- We want a runner to be a bit like a **kernel** of an OS, i.e., to
  - (i) provide management of **(internal) resources**
  - (ii) use further **external resources**
  - (iii) **signal failure** caused by unavoidable circumstances

# Effectful runners for modular top-levels ctd.

- What would be a **useful class of monads**  $\mathbf{T}$  to use?
- We want a runner to be a bit like a **kernel** of an OS, i.e., to
  - (i) provide management of **(internal) resources**
  - (ii) use further **external resources**
  - (iii) **signal failure** caused by unavoidable circumstances
- **Algebraically** (and pragmatically), this amounts to taking
  - (i)  $\text{getenv} : \mathbb{1} \rightsquigarrow C$ ,  $\text{setenv} : C \rightsquigarrow \mathbb{1}$
  - (ii)  $\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$  ( $\text{op} \in \Sigma'$ , for some external  $\Sigma'$ )
  - (iii)  $\text{kill} : S \rightsquigarrow \mathbb{0}$s.t., (i) satisfy state equations; and (i) commute with (ii) and (iii)



# Effectful runners for modular top-levels ctd.

- What would be a **useful class of monads**  $\mathbf{T}$  to use?
- We want a runner to be a bit like a **kernel** of an OS, i.e., to
  - (i) provide management of **(internal) resources**
  - (ii) use further **external resources**
  - (iii) **signal failure** caused by unavoidable circumstances
- **Algebraically** (and pragmatically), this amounts to taking
  - (i)  $\text{getenv} : \mathbb{1} \rightsquigarrow C$ ,  $\text{setenv} : C \rightsquigarrow \mathbb{1}$
  - (ii)  $\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$  ( $\text{op} \in \Sigma'$ , for some external  $\Sigma'$ )
  - (iii)  $\text{kill} : S \rightsquigarrow \mathbb{0}$s.t., (i) satisfy state equations; and (i) commute with (ii) and (iii)
- The **induced monad** is then isomorphic to

$$\mathbf{T} X \stackrel{\text{def}}{=} C \Rightarrow \mathbf{Free}_{\Sigma'}((X \times C) + S)$$

## Effectful runners for modular top-levels ctd.

- The corresponding **T-runners**  $\mathcal{R}$  for  $\Sigma$  are then of the form

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow C \Rightarrow \mathbf{Free}_{\Sigma'}((B_{\text{op}} \times C) + S) \right)_{\text{op} \in \Sigma}$$

## Effectful runners for modular top-levels ctd.

- The corresponding **T-runners**  $\mathcal{R}$  for  $\Sigma$  are then of the form

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow C \Rightarrow \mathbf{Free}_{\Sigma'}((B_{\text{op}} \times C) + S) \right)_{\text{op} \in \Sigma}$$

- Observe that raising signals in  $S$  **discards the state**,  
but **not all problems are terminal**—they can be recovered from

# Effectful runners for modular top-levels ctd.

- The corresponding **T-runners**  $\mathcal{R}$  for  $\Sigma$  are then of the form

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow C \Rightarrow \mathbf{Free}_{\Sigma'}((B_{\text{op}} \times C) + S) \right)_{\text{op} \in \Sigma}$$

- Observe that raising signals in  $S$  **discards the state**,  
but **not all problems are terminal**—they can be recovered from
- **Our solution:** consider signatures  $\Sigma$  with operation symbols

$$\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}} + E_{\text{op}}$$

# Effectful runners for modular top-levels ctd.

- The corresponding **T-runners**  $\mathcal{R}$  for  $\Sigma$  are then of the form

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow C \Rightarrow \mathbf{Free}_{\Sigma'}((B_{\text{op}} \times C) + S) \right)_{\text{op} \in \Sigma}$$

- Observe that raising signals in  $S$  **discards the state**,  
but **not all problems are terminal**—they can be recovered from
- Our solution:** consider signatures  $\Sigma$  with operation symbols

$$\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}} + E_{\text{op}}$$

- With this, our **T-runners**  $\mathcal{R}$  for  $\Sigma$  are of the form

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{K}_C^{\Sigma'!E_{\text{op}} \not\vdash S} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

where we call  $\mathbf{K}_C^{\Sigma'!E \not\vdash S}$  a **kernel monad**, given by

$$\mathbf{K}_C^{\Sigma'!E \not\vdash S} X \stackrel{\text{def}}{=} C \Rightarrow \mathbf{Free}_{\Sigma}(((X + E) \times C) + S)$$

**T-runners as a programming construct**

# T-runners as a programming construct

- As our **T-runners** for  $\Sigma$  are of the form

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{K}_C^{\Sigma'!E_{\text{op}} \not\leq S} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

we can easily accommodate them in a programming language as

```
let R = runner { op1 x1 → K1 , ... , opn xn → Kn } @ C
```

where  $K_i$  are **kernel computations**, modelled using  $\mathbf{K}_C^{\Sigma'!E_{\text{op}_i} \not\leq S}$

# T-runners as a programming construct

- As our **T-runners** for  $\Sigma$  are of the form

$$\left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{K}_C^{\Sigma'!E_{\text{op}} \not\leq S} B_{\text{op}} \right)_{\text{op} \in \Sigma}$$

we can easily accommodate them in a programming language as

```
let R = runner { op1 x1 → K1 , ... , opn xn → Kn } @ C
```

where  $K_i$  are **kernel computations**, modelled using  $\mathbf{K}_C^{\Sigma'!E_{\text{op}_i} \not\leq S}$

- For instance, we can implement a **write-only file handle** as

```
let RFH = runner {  
  write s → if (length s > max)  
    then (raise WriteSizeLimitExceeded)  
    else (let fh = getenv () in fwrite (fh,s))  
} @ FileHandle
```

where

$$\Sigma \stackrel{\text{def}}{=} \{ \text{write} : \text{String} \rightsquigarrow \mathbb{1} \} \quad \text{fwrite} : \text{FileHandle} \times \text{String} \rightsquigarrow \mathbb{1} \in \Sigma'$$

$$\text{WriteSizeLimitExceeded} \in E_{\text{op}} \quad S = \emptyset$$



# Controlled **initialisation** and **finalisation**

# Controlled **initialisation** and **finalisation**

- Recall that the components  $r_X$  of the monad morphism

$$r : \mathbf{Free}_\Sigma(-) \longrightarrow \mathbf{T}$$

induced by a  $\mathbf{T}$ -runner  $\mathcal{R}$  are all **tail-recursive**

# Controlled **initialisation** and **finalisation**

- Recall that the components  $r_X$  of the monad morphism

$$r : \mathbf{Free}_\Sigma(-) \longrightarrow \mathbf{T}$$

induced by a  $\mathbf{T}$ -runner  $\mathcal{R}$  are all **tail-recursive**

- We can make use of it, to accommodate **running user code**:

```
using R @ M1  
run M2  
finally { return x @ c → M3 , raise e @ c → M4 , kill s → M5 }
```

where

- $M_1$  is an **initialiser** producing the initial kernel state
- $M_2$  is the user computation being run using the runner  $R$
- $M_3$ ,  $M_4$ , and  $M_5$  are **finalisers** for return value, exceptions, signals

# Controlled **initialisation** and **finalisation**

- Recall that the components  $r_X$  of the monad morphism

$$r : \mathbf{Free}_\Sigma(-) \longrightarrow \mathbf{T}$$

induced by a  $\mathbf{T}$ -runner  $\mathcal{R}$  are all **tail-recursive**

- We can make use of it, to accommodate **running user code**:

```
using R @ M1  
run M2  
finally { return x @ c → M3 , raise e @ c → M4 , kill s → M5 }
```

where

- $M_1$  is an **initialiser** producing the initial kernel state
- $M_2$  is the user computation being run using the runner  $R$
- $M_3$ ,  $M_4$ , and  $M_5$  are **finalisers** for return value, exceptions, signals
- $M_3$  and  $M_4$  **depend on the final state**  $c$ , but  $M_5$  **does not**

# Controlled **initialisation** and **finalisation** ctd.

- For instance, we can define a PYTHON-like **with-file construct**

```
with file_name do M
=
using RFH @ (fopen file_name)
run M
finally {
  return x @ fh → fclose fh; return x ,
  raise e @ fh → fclose fh; raise e ,
  kill s → match s with {} }
```

- Importantly,
  - here the file handle is hidden from `M`
  - and `M` can only use `write` of type `String ↪ 1`
  - and `fopen` and `fclose` are limited to initialisation-finalisation

# Controlled **initialisation** and **finalisation** ctd.

- Semantically, in

<b>using</b> R @ M <sub>1</sub>	( * (a) * )
<b>run</b> M <sub>2</sub>	( * (b) * )
<b>finally</b> { <b>return</b> x @ c → M <sub>3</sub> , <b>raise</b> e @ c → M <sub>4</sub> , <b>kill</b> s → M <sub>5</sub> }	( * (c) * )

- M<sub>1</sub> denotes an element of  $\mathbf{U}^{\Sigma'!E'} C \stackrel{\text{def}}{=} \mathbf{Free}_{\Sigma'}(C + E')$   
(a **user monad**)
- M<sub>2</sub> denotes an element of  $\mathbf{U}^{\Sigma!E} A$
- M<sub>3</sub> denotes an element of  $A \times C \Rightarrow \mathbf{U}^{\Sigma'!E'} B$
- M<sub>4</sub> denotes an element of  $E \times C \Rightarrow \mathbf{U}^{\Sigma'!E'} B$
- M<sub>5</sub> denotes an element of  $S \Rightarrow \mathbf{U}^{\Sigma'!E'} B$

# Controlled **initialisation** and **finalisation** ctd.

- Semantically, in

<b>using</b> R @ M <sub>1</sub>	( * (a) * )
<b>run</b> M <sub>2</sub>	( * (b) * )
<b>finally</b> { <b>return</b> x @ c → M <sub>3</sub> , <b>raise</b> e @ c → M <sub>4</sub> , <b>kill</b> s → M <sub>5</sub> }	( * (c) * )

- M<sub>1</sub> denotes an element of  $\mathbf{U}^{\Sigma'!E'} C \stackrel{\text{def}}{=} \mathbf{Free}_{\Sigma'}(C + E')$   
(a **user monad**)
- M<sub>2</sub> denotes an element of  $\mathbf{U}^{\Sigma!E} A$
- M<sub>3</sub> denotes an element of  $A \times C \Rightarrow \mathbf{U}^{\Sigma'!E'} B$
- M<sub>4</sub> denotes an element of  $E \times C \Rightarrow \mathbf{U}^{\Sigma'!E'} B$
- M<sub>5</sub> denotes an element of  $S \Rightarrow \mathbf{U}^{\Sigma'!E'} B$
- allowing us to interpret (b) and (c) as the **composite**

$$\mathbf{U}^{\Sigma!E} A \xrightarrow[(b)]{r_{A+E}} \mathbf{K}_C^{\Sigma'!E \downarrow S} A \xrightarrow[(c)]{(\lambda M_3)^\dagger} C \Rightarrow \mathbf{U}^{\Sigma'!E'} B$$

and (a) using the **Kleisli extension** of  $\mathbf{U}^{\Sigma'!E'}$

**A core calculus for  
programming with runners**



# Core calculus (very briefly)

# Core calculus (very briefly)

- **Values**

$$\llbracket \Gamma \vdash V : A \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket$$

- **User computations**

$$\llbracket \Gamma \stackrel{\Sigma}{\vdash} M : A ! E \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \mathbf{U}^{\Sigma ! E} \llbracket A \rrbracket$$

- **Kernel computations**

$$\llbracket \Gamma \stackrel{\Sigma}{\vdash} K : A ! E \downarrow S @ C \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \mathbf{K}_{[C]}^{\Sigma ! E \downarrow S} \llbracket A \rrbracket$$

# Core calculus (very briefly) ctd.

$$\begin{aligned} M ::= & \text{ return } V \mid \text{ try } M \text{ with } \{ \text{ return } x \mapsto N_{val} , (\text{ raise } e \mapsto N_e)_{e \in E} \} \\ & \mid V W \mid \text{ match } V \text{ with } \{ \langle x_1, x_2 \rangle \mapsto N \} \\ & \mid \text{ match } V \text{ with } \{ \}_X \mid \text{ match } V \text{ with } \{ \text{ inl } x_1 \mapsto N_1 , \text{ inr } x_2 \mapsto N_2 \} \\ & \mid \text{ op}_X V (x.M) (N_e)_{e \in E_{\text{op}}} \mid \text{ raise}_X e \\ & \mid \text{ using } V @ W \text{ run } M \text{ finally } \{ \text{ return } x @ c \mapsto N_{val} , \\ & \qquad \qquad \qquad (\text{ raise } e @ c \mapsto N_e)_{e \in E} , \\ & \qquad \qquad \qquad (\text{ kill } s \mapsto N_s)_{s \in S} \} \\ & \mid \text{ exec } K @ W \text{ finally } \{ \text{ return } x @ c \mapsto N_{val} , \\ & \qquad \qquad \qquad (\text{ raise } e @ c \mapsto N_e)_{e \in E} , \\ & \qquad \qquad \qquad (\text{ kill } s \mapsto N_s)_{s \in S} \} \\ \\ K ::= & \text{ return}_C V \mid \text{ try } K \text{ with } \{ \text{ return } x \mapsto L_{val} , (\text{ raise } e \mapsto L_e)_{e \in E} \} \\ & \mid V W \mid \text{ match } V \text{ with } \{ \langle x_1, x_2 \rangle \mapsto L \} \\ & \mid \text{ match } V \text{ with } \{ \}_X @ C \mid \text{ match } V \text{ with } \{ \text{ inl } x_1 \mapsto L_1 , \text{ inr } x_2 \mapsto L_2 \} \\ & \mid \text{ op}_{X @ C} V (x.K) (L_e)_{e \in E_{\text{op}}} \mid \text{ raise}_{X @ C} e \mid \text{ kill}_{X @ C} s \\ & \mid \text{ getenv}_C (c.K) \mid \text{ setenv } V K \\ & \mid \text{ exec } M \text{ finally } \{ \text{ return } x \mapsto L_{val} , (\text{ raise } e \mapsto L_e)_{e \in E} \} \end{aligned}$$

Fig. 1. Syntax of user and kernel computations

# Core calculus (very briefly) ctd.

- For example, the **typing rule for running user comps.** is

$$\begin{array}{c}
 \Gamma \vdash V : \Sigma \Rightarrow \Sigma' \not\downarrow S @ C \quad \Gamma \vdash W : C \\
 \Gamma \Vdash M : A ! E \quad \Gamma, x:A, c:C \Vdash' N_{ret} : B ! E' \\
 (\Gamma, c:C \Vdash' N_e : B ! E')_{e \in E} \quad (\Gamma \Vdash' N_s : B ! E')_{s \in S} \\
 \hline
 \Gamma \Vdash' \textbf{using } V @ W \textbf{ run } M \textbf{ finally } \{ \textbf{return } x @ c \mapsto N_{ret} , \\
 \textbf{(raise } e @ c \mapsto N_e)_{e \in E} , \\
 \textbf{(kill } s \mapsto N_s)_{s \in S} \} : B ! E'
 \end{array}$$

# Core calculus (very briefly) ctd.

- For example, the **typing rule for running user comps.** is

$$\begin{array}{c}
 \Gamma \vdash V : \Sigma \Rightarrow \Sigma' \not\downarrow S @ C \quad \Gamma \vdash W : C \\
 \Gamma \Vdash M : A ! E \quad \Gamma, x:A, c:C \Vdash N_{ret} : B ! E' \\
 (\Gamma, c:C \Vdash N_e : B ! E')_{e \in E} \quad (\Gamma \Vdash N_s : B ! E')_{s \in S} \\
 \hline
 \Gamma \Vdash \text{using } V @ W \text{ run } M \text{ finally } \{ \text{return } x @ c \mapsto N_{ret} , \\
 \text{(raise } e @ c \mapsto N_e)_{e \in E} , \\
 \text{(kill } s \mapsto N_s)_{s \in S} \} : B ! E'
 \end{array}$$

- and the **main  $\beta$ -equation for running user comps.** is

$$\begin{aligned}
 &\Gamma \Vdash \text{using } R_C @ W \text{ run } (\text{op}_X V (x.M) (M_e)_{e \in E_{\text{op}}}) \text{ finally } F \\
 &\equiv \text{exec } R_{op}[V] @ W \text{ finally } \{ \\
 &\quad \text{return } x @ c' \mapsto \text{using } R_C @ c' \text{ run } M \text{ finally } F , \\
 &\quad (\text{raise } e @ c' \mapsto \text{using } R_C @ c' \text{ run } M_e \text{ finally } F)_{e \in E_{\text{op}}} , \\
 &\quad (\text{kill } s \mapsto N_s)_{s \in S} \} : Y ! E'
 \end{aligned}$$

**Runners in action**

Runners can be **vertically nested**

# Runners can be **vertically nested**

- ```
using RFH @ (fopen file_name)
run (
  using RFC @ (return "")
  run m
  finally {
    return x @ s → write s; return x ,
    raise e @ s → write s; raise e }
)
finally {
  return x @ fh → fclose fh; return x ,
  raise e @ fh → fclose fh; raise e }
```

where the **file contents runner** (with  $\Sigma' = \mathbb{O}$ ) is defined as

```
let RFC = runner {
  write s → let s' = getenv () in
    if (length (s^s') > max) then (raise WriteSizeExceeded)
    else (setenv (s^s'))
} @ String
```



Runners can be horizontally paired

# Runners can be horizontally paired

- Given a runner for  $\Sigma$

```
let R1 = runner { ... , op1_i x → k1_i , ... } @ C1
```

and a runner for  $\Sigma'$

```
let R2 = runner { ... , op2_i x → k2_i , ... } @ C2
```

we can **pair them** to get a runner for  $\Sigma \cup \Sigma'$

```
let R = runner {  
  ... ,  
  op1_i x → let (c,c') = getenv () in  
             let (x,c'') = k1_i x in  
             setenv (c'', c');  
             return x,  
  
  ... ,  
  op2_i x → ... (* analogously to above *) ,  
  ...  
} @ C1 * C2
```

# Vertical nesting for instrumentation

# Vertical nesting for instrumentation

- ```
using RSniffer @ (return 0)
run m
finally {
  return x @ c →
    let fh = fopen "nsa.txt" in fwrite (fh, to_str c); fclose fh }
```

where the **instrumenting runner** is defined as

```
let RSniffer = runner {
  ... ,
  op a → op a;                                (* forwards op outwards *)
  let c = getenv () in
  setenv (c + 1) ,
  ...
} @ Nat
```

- The runner  $R_{\text{Sniffer}}$  implements the same sig.  $\Sigma$  that `m` is using
- As a result, the runner  $R_{\text{Sniffer}}$  is **invisible** from `m`'s viewpoint

# Integer state **with** active monitoring

# Integer state with active monitoring

- `type` IntHeap = { memory : Nat → Option Int ; next : Nat }

```
let R_IntState = runner {  
  alloc x → ... ,  
  
  deref r → let h = getenv () in  
             match (heap_sel h r) with  
             | Some x → return x  
             | None → kill "ReferenceDoesNotExistSignal" ,  
  
  assign r y → let h = getenv () in  
               match (heap_upd h r y) with  
               | Some h' → if (rel x y)  
                           then (setenv h')  
                           else (raise "MonotonicityException")  
               | None → kill "ReferenceDoesNotExistSignal"  
}  
@ IntHeap
```

# Integer state with active monitoring

- **type** IntHeap = { memory : Nat  $\rightarrow$  Option Int ; next : Nat }

```
let RIntState = runner {  
  alloc x  $\rightarrow$  ... ,  
  
  deref r  $\rightarrow$  let h = getenv () in  
    match (heap_sel h r) with  
    | Some x  $\rightarrow$  return x  
    | None  $\rightarrow$  kill "ReferenceDoesNotExistSignal" ,  
  
  assign r y  $\rightarrow$  let h = getenv () in  
    match (heap_upd h r y) with  
    | Some h'  $\rightarrow$  if (rel x y)  
      then (setenv h')  
      else (raise "MonotonicityException")  
    | None  $\rightarrow$  kill "ReferenceDoesNotExistSignal"  
}
```

@ IntHeap

- This is **runtime verification** for **rel-monotonic integer state**

# Integer state with active monitoring

- **type** IntHeap = { memory : Nat  $\rightarrow$  Option Int ; next : Nat }

```
let RIntState = runner {  
  alloc x  $\rightarrow$  ... ,  
  
  deref r  $\rightarrow$  let h = getenv () in  
    match (heap_sel h r) with  
    | Some x  $\rightarrow$  return x  
    | None  $\rightarrow$  kill "ReferenceDoesNotExistSignal" ,  
  
  assign r y  $\rightarrow$  let h = getenv () in  
    match (heap_upd h r y) with  
    | Some h'  $\rightarrow$  if (rel x y)  
      then (setenv h')  
      else (raise "MonotonicityException")  
    | None  $\rightarrow$  kill "ReferenceDoesNotExistSignal"  
}
```

@ IntHeap

- This is **runtime verification** for **rel-monotonic integer state**
- Also possible with **vertical nesting**: MLState  $\leftrightarrow$  Monotonicity



# Other examples

- More general forms of **(ML-style) state** (for general Ref A)
  - if the host language allows it, we use GADTs, etc for safety
  - some examples extract a footprint from a larger memory
- **Combinations** of different effects and runners
  - in particular the combination of IO and state
  - good use case for both vertical and horizontal composition
- KOKA-style **ambient values** and **ambient functions**
  - ambient values are essentially mutable variables/parameters
  - ambient functions are executed in their lexical context
  - a runner for amb. funs. treats fun. application as a co-operation
  - amb. funs. are stored in a context-sensitive heap
  - the appl. co-operation restores the heap to the lexical context

## Implementing runners

Experimenting with the **theory in practice**

# Experimenting with the **theory in practice**

- A **small experimental language** COOP<sup>4</sup>
  - Implements the core calculus with few extras
  - The interpreter is directly based on the denotational semantics
  - Top-level containers for running external (OCaml) code

---

<sup>4</sup>coop [/ku:p/] – a cage where small animals are kept, especially chickens

# Experimenting with the **theory in practice**

- A **small experimental language** COOP<sup>4</sup>
  - Implements the core calculus with few extras
  - The interpreter is directly based on the denotational semantics
  - Top-level containers for running external (OCaml) code
- A **HASKELL library** HASKELL-COOP
  - A shallow-embedding of the core calculus in HASKELL
  - Uses one of the Freer monad implementations underneath
  - Again, the operational aspects implement the denot. semantics
  - Top-level containers for arbitrary HASKELL monads
  - Examples make use of HASKELL's features (GADTs, ...)

---

<sup>4</sup>coop [/ku:p/] – a cage where small animals are kept, especially chickens

# Experimenting with the **theory in practice**

- A **small experimental language** COOP<sup>4</sup>
  - Implements the core calculus with few extras
  - The interpreter is directly based on the denotational semantics
  - Top-level containers for running external (OCaml) code
- A **HASKELL library** HASKELL-COOP
  - A shallow-embedding of the core calculus in HASKELL
  - Uses one of the Freer monad implementations underneath
  - Again, the operational aspects implement the denot. semantics
  - Top-level containers for arbitrary HASKELL monads
  - Examples make use of HASKELL's features (GADTs, ...)
- Both still need some finishing touches, but will be public soon

---

<sup>4</sup>coop [/ku:p/] – a cage where small animals are kept, especially chickens

# Experimenting with the theory in practice

```
module AmbientsTests where

import Control.Runner
import Control.Runner.Ambients

ambFun :: AmbVal Int -> Int -> AmbEff Int
ambFun x y =
  do x <- getVal x;
  return (x + y)

test1 :: AmbEff Int
test1 =
  withAmbVal
    (4 :: Int)
    (\ x ->
      withAmbFun
        (ambFun x)
        (\ f ->
          do rebindVal x 2;
          applyFun f 1))

test2 = ambToplevel test1
```

# Wrapping up

- **Runners** are a natural model of **top-level runtime**
- We proposed **T-runners** to also model **non-top-level runtimes**
- We turned **T-runners** into a **practical programming construct**, that supports controlled initialisation and finalisation
- We showed some **combinators** and **programming examples**
- Two **implementations** in the works, COOP and HASKELL-COOP

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 834146.



This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.



**Thank you!**

