

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 ()) (* : int → a * int *)
```

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 ()) (* : int → a * int *)
```

- 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**¹ Σ of operation symbols ($A_{\text{op}}, B_{\text{op}}$ countable)

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

a **runner**² \mathcal{R} for Σ is given by a carrier $|\mathcal{R}|$ and co-operations

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

¹We consider runners for signatures, but the work generalises to alg. theories.

²In the literature also known as **comodels** for Σ (or for an algebraic theory).

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

- Given a **signature**¹ Σ of operation symbols ($A_{\text{op}}, B_{\text{op}}$ countable)

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

a **runner**² \mathcal{R} for Σ 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

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

¹We consider runners for signatures, but the work generalises to alg. theories.

²In the literature also known as **comodels** for Σ (or for an algebraic 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}
 - **monad morphisms**³ $r : \mathbf{Free}_\Sigma(-) \longrightarrow \mathbf{St}_{|\mathcal{R}|}$

where

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

³ $\mathbf{Free}_\Sigma(X)$ 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 Σ 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 Σ 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.,

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

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 Σ 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.,

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

- Observe that κ 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

Effective 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 Σ')
 - (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 Σ')
 - (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 Σ 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 Σ 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 Σ 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 Σ 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 Σ 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 Σ with operation symbols

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

- With this, our **T-runners** \mathcal{R} for Σ are (with “primitive” excs.)

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

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

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

T-runners as a programming construct

T-runners as a programming construct

- As our **T-runners** for Σ 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 code**, modelled using $\mathbf{K}_C^{\Sigma'!E_{\text{op}_i} \not\leq S}$

T-runners as a **programming construct**

- As our **T-runners** for Σ 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 code**, 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 WriteSizeExceeded)  
    else (let fh = getenv () in  
      if (valid fh) then (fwrite (fh,s)) else (kill IOError))  
} @ FileHandle
```

where

$$(\text{fwrite} : \text{FileHandle} \times \text{String} \rightsquigarrow 1 + E) \in \Sigma'$$

$$\Sigma \stackrel{\text{def}}{=} \{ \text{write} : \text{String} \rightsquigarrow 1 + E \cup \{\text{WriteSizeExceeded}\} \} \quad \text{IOError} \in S$$

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

```
using R @ Minit  
run M  
finally {return x @ c → Mret , ... raise e @ c → Me ... , ... kill s → Ms ...}
```

where

- M_s are **user code**, modelled using $\mathbf{U}^{\Sigma!E} X \stackrel{\text{def}}{=} \mathbf{Free}_\Sigma(X + E)$ (a **user monad**)

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

```
using R @ Minit  
run M  
finally {return x @ c → Mret , ... raise e @ c → Me ... , ... kill s → Ms ...}
```

where

(a **user monad**)

- M_s are **user code**, modelled using $\mathbf{U}^{\Sigma!E} X \stackrel{\text{def}}{=} \mathbf{Free}_\Sigma(X + E)$
- M_{init} produces the **initial kernel state**
- M is the user code being **run using the runner** R
- M_{ret} , M_e , M_s **finalise** for return values, exceptions, and 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 make use of it, to accommodate **running user code**:

```
using R @ Minit  
run M  
finally {return x @ c → Mret , ... raise e @ c → Me ... , ... kill s → Ms ...}
```

where

(a **user monad**)

- M_s are **user code**, modelled using $\mathbf{U}^{\Sigma!E} X \stackrel{\text{def}}{=} \mathbf{Free}_\Sigma(X + E)$
- M_{init} produces the **initial kernel state**
- M is the user code being **run using the runner** R
- M_{ret} , M_e , M_s **finalise** for return values, exceptions, and signals
- M_{ret} and M_e **depend on the final state** c , but M_s **does not**

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

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

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

- Importantly, here
 - the **file handle is hidden** from M
 - M can only use **write** but not **fopen** and **fclose**
 - fopen** and **fclose** are limited to **initialisation-finalisation**

**A core calculus for
programming with runners**

Core calculus (syntax)

Core calculus (syntax)

- **Ground types** (types of ops. and kernel state)

$$A, B, C ::= B \mid 1 \mid 0 \mid A \times B \mid A + B$$

- **Types**

$$\begin{aligned} X, Y &::= B \mid 1 \mid 0 \mid X \times Y \mid X + Y \\ &\mid X \xrightarrow{\Sigma} Y ! E \\ &\mid X \xrightarrow{\Sigma} Y ! E \Downarrow S @ C \\ &\mid \Sigma \Rightarrow \Sigma' \Downarrow S @ C \end{aligned}$$

- **Values**

$$\Gamma \vdash V : X$$

- **User computations**

$$\Gamma \Vdash M : X ! E$$

- **Kernel computations**

$$\Gamma \Vdash K : X ! E \Downarrow S @ C$$

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

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

Fig. 1. Syntax of user and kernel computations

Core calculus (type system and eq. theory)

Core calculus (type system and eq. theory)

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

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

Core calculus (type system and eq. theory)

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

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

- and the **main β -equation for running user comps.** is

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

Core calculus (semantics)

Core calculus (semantics)

- Monadic semantics, for simplicity in **Set**, using
 - **user monads** $\mathbf{U}^{\Sigma!E} X \stackrel{\text{def}}{=} \mathbf{Free}_{\Sigma}(X + E)$
 - **kernel monads** $\mathbf{K}_C^{\Sigma!E \not\vdash S} X \stackrel{\text{def}}{=} C \Rightarrow \mathbf{Free}_{\Sigma}(((X + E) \times C) + S)$

Core calculus (semantics)

- Monadic semantics, for simplicity in **Set**, using
 - **user monads** $\mathbf{U}^{\Sigma!E} X \stackrel{\text{def}}{=} \mathbf{Free}_{\Sigma}(X + E)$
 - **kernel monads** $\mathbf{K}_C^{\Sigma!E \not\downarrow S} X \stackrel{\text{def}}{=} C \Rightarrow \mathbf{Free}_{\Sigma}(((X + E) \times C) + S)$
- (At a high level) the **judgements are interpreted** as

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

$$\llbracket \Gamma \Vdash M : X ! E \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \mathbf{U}^{\Sigma!E} \llbracket X \rrbracket$$

$$\llbracket \Gamma \Vdash K : X ! E \not\downarrow S @ C \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \mathbf{K}_{[C]}^{\Sigma!E \not\downarrow S} \llbracket X \rrbracket$$

Core calculus (semantics ctd.)

- However, to prove **coherence** of the semantics (**subtyping!**), we actually give the semantics in the **subset fibration**

Core calculus (semantics ctd.)

- However, to prove **coherence** of the semantics (**subtyping!**), we actually give the semantics in the **subset fibration**
- For instance, **kernel computations** are interpreted as

$$\begin{array}{ccc} \llbracket \Gamma \rrbracket & \xrightarrow{\llbracket \Gamma \vdash K : X ! E \downarrow S @ C \rrbracket} & \mathbf{K}_{\llbracket C \rrbracket}^{\Sigma ! E \downarrow S} \llbracket X \rrbracket \\ \downarrow \subseteq & & \downarrow \subseteq \\ \llbracket \Gamma^s \rrbracket & \xrightarrow{\llbracket \Gamma^s \vdash K : X^s @ C \rrbracket} & \mathbf{K}_{\llbracket C \rrbracket}^{\emptyset ! E \downarrow S} \llbracket X^s \rrbracket \end{array}$$

where $\Gamma^s \vdash K : X^s @ C$ is a **skeletal kernel typing judgement**

Core calculus (semantics ctd.)

- However, to prove **coherence** of the semantics (**subtyping!**), we actually give the semantics in the **subset fibration**
- For instance, **kernel computations** are interpreted as

$$\begin{array}{ccc} \llbracket \Gamma \rrbracket & \xrightarrow{\llbracket \Gamma \models K : X ! E \downarrow S @ C \rrbracket} & \mathbf{K}_{\llbracket C \rrbracket}^{\Sigma ! E \downarrow S} \llbracket X \rrbracket \\ \downarrow \subseteq & & \downarrow \subseteq \\ \llbracket \Gamma^s \rrbracket & \xrightarrow{\llbracket \Gamma^s \vdash K : X^s @ C \rrbracket} & \mathbf{K}_{\llbracket C \rrbracket}^{\emptyset ! E \downarrow S} \llbracket X^s \rrbracket \end{array}$$

where $\Gamma^s \vdash K : X^s @ C$ is a **skeletal kernel typing judgement**

- No essential obstacles to extending to **Sub(Cpo)** and beyond

Core calculus (semantics ctd.)

$$\begin{aligned} \llbracket \Gamma \stackrel{\Sigma'}{\vdash} \text{using } V @ W \text{ run } M \text{ finally } \{ & \text{return } x @ c \mapsto N_{ret} , \\ & (\text{raise } e @ c \mapsto N_e)_{e \in E} , \\ & (\text{kill } s \mapsto N_s)_{s \in S} \} : Y ! E' \rrbracket_\gamma \stackrel{\text{def}}{=} \dots \end{aligned}$$

Core calculus (semantics ctd.)

$$\llbracket \Gamma \stackrel{\Sigma'}{\vdash} \text{using } V @ W \text{ run } M \text{ finally } \{ \text{return } x @ c \mapsto N_{ret} , \\ (\text{raise } e @ c \mapsto N_e)_{e \in E} , \\ (\text{kill } s \mapsto N_s)_{s \in S} \} : Y ! E' \rrbracket_{\gamma} \stackrel{\text{def}}{=} \dots$$

- $\llbracket V \rrbracket_{\gamma} = \mathcal{R} = \left(\overline{\text{op}}_{\mathcal{R}} : \llbracket A_{\text{op}} \rrbracket \longrightarrow \mathbf{K}_{[C]}^{\Sigma' ! E_{\text{op}} \not\vdash S} \llbracket B_{\text{op}} \rrbracket \right)_{\text{op} \in \Sigma}$
- $\llbracket W \rrbracket_{\gamma} \in \llbracket C \rrbracket$
- $\llbracket M \rrbracket_{\gamma} \in \mathbf{U}^{\Sigma ! E} \llbracket A \rrbracket$
- $\llbracket \text{return } x @ c \mapsto N_{ret} \rrbracket_{\gamma} \in \llbracket A \rrbracket \times \llbracket C \rrbracket \Rightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$
- $\llbracket (\text{raise } e @ c \mapsto N_e)_{e \in E} \rrbracket_{\gamma} \in E \times \llbracket C \rrbracket \Rightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$
- $\llbracket (\text{kill } s \mapsto N_s)_{s \in S} \rrbracket_{\gamma} \in S \Rightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$

Core calculus (semantics ctd.)

$$\llbracket \Gamma \stackrel{\Sigma'}{\vdash} \text{using } V @ W \text{ run } M \text{ finally } \{ \text{return } x @ c \mapsto N_{ret} , \\ (\text{raise } e @ c \mapsto N_e)_{e \in E} , \\ (\text{kill } s \mapsto N_s)_{s \in S} \} : Y ! E' \rrbracket_{\gamma} \stackrel{\text{def}}{=} \dots$$

- $\llbracket V \rrbracket_{\gamma} = \mathcal{R} = \left(\overline{\text{op}}_{\mathcal{R}} : \llbracket A_{\text{op}} \rrbracket \longrightarrow \mathbf{K}_{[C]}^{\Sigma' ! E_{\text{op}} \not\vdash S} \llbracket B_{\text{op}} \rrbracket \right)_{\text{op} \in \Sigma}$
- $\llbracket W \rrbracket_{\gamma} \in \llbracket C \rrbracket$
- $\llbracket M \rrbracket_{\gamma} \in \mathbf{U}^{\Sigma ! E} \llbracket A \rrbracket$
- $\llbracket \text{return } x @ c \mapsto N_{ret} \rrbracket_{\gamma} \in \llbracket A \rrbracket \times \llbracket C \rrbracket \Rightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$
- $\llbracket (\text{raise } e @ c \mapsto N_e)_{e \in E} \rrbracket_{\gamma} \in E \times \llbracket C \rrbracket \Rightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$
- $\llbracket (\text{kill } s \mapsto N_s)_{s \in S} \rrbracket_{\gamma} \in S \Rightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$
- allowing us to use the **free model property** to get

$$\mathbf{U}^{\Sigma ! E} \llbracket A \rrbracket \xrightarrow{r_{[A] + E}} \mathbf{K}_{[C]}^{\Sigma' ! E \not\vdash S} \llbracket A \rrbracket \xrightarrow{(\lambda \llbracket M_3 \rrbracket_{\gamma})^{\dagger}} \llbracket C \rrbracket \Rightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$$

and then apply the resulting composite to $\llbracket M \rrbracket_{\gamma}$ and $\llbracket W \rrbracket_{\gamma}$

Runners in action

Runners can be **vertically nested**

Runners can be **vertically nested**

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

where the **file contents runner** (with  $\Sigma' = \emptyset$ ) is defined as

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

Runners can be horizontally paired

# Runners can be horizontally paired

- Given a runner for  $\Sigma$

```
let R1 = runner { ... , op1i x → k1i , ... } @ C1
```

and a runner for  $\Sigma'$

```
let R2 = runner { ... , op2j x → k2j , ... } @ C2
```

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

```
let R⊗ = runner {
 ... ,
 op1i x → let (c,c') = getenv () in
 let (x,c'') = k1i x in
 setenv (c'',c');
 return x ,
 ... ,
 op2j x → ... (* analogously to above *) ,
 ...
} @ C1 * C2
```

# Runners can be horizontally paired

- Given a runner for  $\Sigma$

```
let R1 = runner { ... , op1i x → k1i , ... } @ C1
```

and a runner for  $\Sigma'$

```
let R2 = runner { ... , op2j x → k2j , ... } @ C2
```

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

```
let R⊗ = runner {
 ... ,
 op1i x → let (c,c') = getenv () in
 let (x,c'') = k1i x in
 setenv (c'',c');
 return x ,

 ... ,
 op2j x → ... (* analogously to above *) ,
 ...
} @ C1 * C2
```

- For instance, this way we can build a **runner for multiple files**

# 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, nat_to_str c); fclose fh }
```

where the **instrumenting runner** is defined as

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

(* forwards op outwards *)

- The runner R_{Sniffer} implements the same sig. Σ that M is using
- As a result, the runner R_{Sniffer} is **invisible** from M 's viewpoint

Integer state **with** active monitoring

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 (heapSel h r) with  
    | Some x  $\rightarrow$  return x  
    | None  $\rightarrow$  kill ReferenceDoesNotExistSignal ,  
  
  assign r y  $\rightarrow$  let h = getenv () in  
    match (heapUpd h r y) with  
    | Some h'  $\rightarrow$  if (rel x y)  
      then (setenv h')  
      else (raise MonotonicityException)  
    | None  $\rightarrow$  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 (heapSel h r) with  
    | Some x  $\rightarrow$  return x  
    | None  $\rightarrow$  kill ReferenceDoesNotExistSignal ,  
  
  assign r y  $\rightarrow$  let h = getenv () in  
    match (heapUpd 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 (heapSel h r) with  
    | Some x  $\rightarrow$  return x  
    | None  $\rightarrow$  kill ReferenceDoesNotExistSignal ,  
  
  assign r y  $\rightarrow$  let h = getenv () in  
    match (heapUpd 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** COOP⁴
 - Implements the core calculus with few extras
 - The interpreter is directly based on the denotational semantics
 - Top-level containers for running external (OCaml) code

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

Experimenting with the **theory in practice**

- A **small experimental language** COOP⁴
 - 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, ...)

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

Experimenting with the **theory in practice**

- A **small experimental language** COOP⁴
 - 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

⁴coop [/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 propose **T-runners** to also model **non-top-level runtimes**
- We have turned **T-runners** into a **(practical ?) programming construct**, that supports controlled initialisation and finalisation
- I showed you some **combinators** and **programming examples**
- Two **implementations** in the works, COOP & HASKELL-COOP
- **Future:** handlers, case studies, more expressive ref. typing, ...

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!

