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 ($A_{\mathsf{op}}, B_{\mathsf{op}}$ countable)

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

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

$$\Big(\overline{\mathsf{op}}_{\mathcal{R}} : A_{\mathsf{op}} \times |\mathcal{R}| \longrightarrow B_{\mathsf{op}} \times |\mathcal{R}|\Big)_{\mathsf{op} \in \Sigma}$$

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

[2] In 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 ($A_{\mathsf{op}}, B_{\mathsf{op}}$ countable)

$$\mathsf{op} : A_{\mathsf{op}} \leadsto B_{\mathsf{op}}$$

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

$$\Big(\overline{\mathsf{op}}_{\mathcal{R}} : A_{\mathsf{op}} \times |\mathcal{R}| \longrightarrow B_{\mathsf{op}} \times |\mathcal{R}|\Big)_{\mathsf{op} \in \Sigma}$$

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

$$\mathsf{get} : \mathbb{1} \leadsto S \qquad \mathsf{set} : S \leadsto \mathbb{1}$$

  is given by

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

[1] We consider runners for signatures, but the work generalises to alg. theories.
[2] In 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] $\mathsf{r} : \mathbf{Free}_{\Sigma}(-) \longrightarrow \mathbf{St}_{|\mathcal{R}|}$

  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 $\mathsf{val}\, x$ and nodes $\mathsf{op}(a, \kappa)$.

A natural model of top-level runtime ctd.

- For our purposes, we see runners

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

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

A natural model of top-level runtime ctd.

- For our purposes, we see runners

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

  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

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

  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

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

  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.,

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

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.,

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

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

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

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.,

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

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

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

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

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

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

$$\mathsf{r}_X\,(\mathsf{val}\, x) = \eta_X\, x \qquad\qquad \mathsf{r}_X\,(\mathsf{op}(a, \kappa)) = (\mathsf{r}_X \circ \kappa)^{\dagger}(\overline{\mathsf{op}}_{\mathcal{R}}\; a)$$

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.,

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

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

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

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

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

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

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

- 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

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)** $\mathsf{getenv} : \mathbb{1} \rightsquigarrow C$, $\mathsf{setenv} : C \rightsquigarrow \mathbb{1}$
  - **(ii)** $\mathsf{op} : A_{\mathsf{op}} \rightsquigarrow B_{\mathsf{op}}$ $\qquad$ ($\mathsf{op} \in \Sigma'$, for some external Σ')
  - **(iii)** $\mathsf{kill} : S \rightsquigarrow \mathbb{0}$

  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)** $\mathsf{getenv} : \mathbb{1} \rightsquigarrow C$, $\mathsf{setenv} : C \rightsquigarrow \mathbb{1}$
  - **(ii)** $\mathsf{op} : A_{\mathsf{op}} \rightsquigarrow B_{\mathsf{op}}$ ($\mathsf{op} \in \Sigma'$, for some external Σ')
  - **(iii)** $\mathsf{kill} : S \rightsquigarrow \mathbb{0}$

  s.t., **(i)** satisfy state equations; and **(i)** commute with **(ii)** and **(iii)**
- The **induced monad** is then isomorphic to

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

Effectful runners for modular top-levels ctd.

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

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

Effectful runners for **modular top-levels** ctd.

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

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

- 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

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

- 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

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

Effectful runners for modular top-levels ctd.

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

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

- 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

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

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

$$\Big(\overline{\mathsf{op}}_{\mathcal{R}} : A_{\mathsf{op}} \longrightarrow \mathbf{K}_C^{\Sigma' ! E_{\mathsf{op}} \lightning S}\, B_{\mathsf{op}}\Big)_{\mathsf{op}\in\Sigma}$$

  where we call $\mathbf{K}_C^{\Sigma ! E \lightning S}$ a **kernel monad**, given by

$$\mathbf{K}_C^{\Sigma ! E \lightning S}\, X \quad \stackrel{\text{def}}{=} \quad C \Rightarrow \mathbf{Free}_{\Sigma}\big(((X + E) \times C) + S\big)$$

T-runners as a programming construct

T-runners as a programming construct

- As our **T-runners** for Σ are of the form

$$\left(\overline{\mathsf{op}}_{\mathcal{R}} : A_{\mathsf{op}} \longrightarrow \mathbf{K}_{C}^{\Sigma' ! E_{\mathsf{op}} \lightning S}\, B_{\mathsf{op}}\right)_{\mathsf{op} \in \Sigma}$$

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 $\mathbf{K}_{C}^{\Sigma' ! E_{\mathsf{op}_i} \lightning S}$

T-runners as a **programming construct**

- As our **T-runners** for Σ are of the form

$$\left(\overline{\mathsf{op}}_{\mathcal{R}} : A_{\mathsf{op}} \longrightarrow \mathbf{K}_C^{\Sigma' ! E_{\mathsf{op}} \lightning S}\, B_{\mathsf{op}}\right)_{\mathsf{op} \in \Sigma}$$

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 $\mathbf{K}_C^{\Sigma' ! E_{\mathsf{op}_i} \lightning S}$

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

```
let R_FH = 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

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

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

Controlled initialisation and finalisation

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

- Recall that the components r_X of the monad morphism

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

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

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

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

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

$$\begin{array}{l} \textbf{using}\ \mathsf{R}\ \textbf{@}\ \mathsf{M}_1 \\ \textbf{run}\ \mathsf{M}_2 \\ \textbf{finally}\ \{\ \textbf{return}\ \mathsf{x}\ \textbf{@}\ \mathsf{c} \to \mathsf{M}_3\ ,\ \textbf{raise}\ \mathsf{e}\ \textbf{@}\ \mathsf{c} \to \mathsf{M}_4\ ,\ \textbf{kill}\ \mathsf{s} \to \mathsf{M}_5\ \} \end{array}$$

  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, M_5 are **finalisers** for return values, exceptions, signals

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

- Recall that the components r_X of the monad morphism

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

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, M_5 are **finalisers** for return values, exceptions, signals
  - M_3 and M_4 **depend on the final state** c, but M_5 **does not**

Controlled initialisation and finalisation

- Recall that the components r_X of the monad morphism

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

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, M_5 are **finalisers** for return values, exceptions, signals
  - M_3 and M_4 **depend on the final state** c, but M_5 **does not**

- User code is modelled using **user monads** $\mathbf{U}^{\Sigma!E}\,X \stackrel{\text{def}}{=} \mathbf{Free}_\Sigma(X + E)$

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

- Recall that the components r_X of the monad morphism

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

  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 (* : Σ ⇒ Σ' ↯ S @ C *) @ M₁ (* : U^{Σ' ! E'} C *)
run M₂ (* : U^{Σ ! E} X *)
finally { return x @ c → M₃ (* : U^{Σ' ! E'} Y *) , ... , 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, M_5 are **finalisers** for return values, exceptions, signals
  - M_3 and M_4 **depend on the final state** c, but M_5 **does not**

- User code is modelled using **user monads** $\mathbf{U}^{\Sigma ! E}\, X \stackrel{\text{def}}{=} \mathbf{Free}_\Sigma(X + E)$

Controlled initialisation and finalisation ctd.

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

```
with file_name do M
=
using R_FH @ (fopen file_name)
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

Controlled initialisation and finalisation ctd.

- **Semantically** (say, in the category of sets), in

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

  - R denotes $\left(\overline{\mathsf{op}}_{\mathcal{R}} : A_{\mathsf{op}} \longrightarrow \mathbf{K}_{C}^{\Sigma' ! E_{\mathsf{op}} \lightning S}\, B_{\mathsf{op}}\right)_{\mathsf{op} \in \Sigma}$
  - M_1 denotes an element of $\mathbf{U}^{\Sigma' ! E'}\, C$
  - M_2 denotes an element of $\mathbf{U}^{\Sigma ! E}\, A$
  - M_3 denotes an element of $A \times C \Rightarrow \mathbf{U}^{\Sigma' ! E'}\, B$
  - M_4 denotes an element of $E \times C \Rightarrow \mathbf{U}^{\Sigma' ! E'}\, B$
  - M_5 denotes an element of $S \Rightarrow \mathbf{U}^{\Sigma' ! E'}\, B$

Controlled initialisation and finalisation ctd.

- **Semantically** (say, in the category of sets), in

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

  - R denotes $\left(\overline{\mathsf{op}}_{\mathcal{R}} : A_{\mathsf{op}} \longrightarrow \mathbf{K}_{C}^{\Sigma'!E_{\mathsf{op}} \lightning S} B_{\mathsf{op}}\right)_{\mathsf{op} \in \Sigma}$
  - M_1 denotes an element of $\mathbf{U}^{\Sigma'!E'}\, C$
  - M_2 denotes an element of $\mathbf{U}^{\Sigma!E}\, A$
  - M_3 denotes an element of $A \times C \Rightarrow \mathbf{U}^{\Sigma'!E'}\, B$
  - M_4 denotes an element of $E \times C \Rightarrow \mathbf{U}^{\Sigma'!E'}\, B$
  - M_5 denotes an element of $S \Rightarrow \mathbf{U}^{\Sigma'!E'}\, B$

- allowing us to interpret (b) and (c) using **free model properties**

$$\mathbf{U}^{\Sigma!E} A \xrightarrow[\text{(b)}]{\mathsf{r}_{A+E}} \mathbf{K}_{C}^{\Sigma'!E \lightning S} A \xrightarrow[\text{(c)}]{(\lambda M_3)^{\ddagger}} C \Rightarrow \mathbf{U}^{\Sigma'!E'} B$$

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)

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

$$A, B, C \quad ::= \quad \mathsf{B} \mid 1 \mid 0 \mid A \times B \mid A + B$$

- **Types**

$$X, Y \;\; ::= \;\; \ldots \mid X \xrightarrow{\Sigma} Y \,!\, E \mid X \xrightarrow{\Sigma} Y \,!\, E \lightning S \,@\, C \mid \Sigma \Rightarrow \Sigma' \lightning S \,@\, C$$

- **Values**

$$[\![\Gamma \vdash V : X]\!] : [\![\Gamma]\!] \longrightarrow [\![X]\!]$$

- **User computations**

$$[\![\Gamma \overset{\Sigma}{\models} M : X \,!\, E]\!] : [\![\Gamma]\!] \longrightarrow \mathbf{U}^{\Sigma ! E}[\![X]\!]$$

- **Kernel computations**

$$[\![\Gamma \overset{\Sigma}{\models} K : X \,!\, E \lightning S \,@\, C]\!] : [\![\Gamma]\!] \longrightarrow \mathbf{K}^{\Sigma ! E \lightning S}_{[\![C]\!]}[\![X]\!]$$

Core calculus (very briefly)

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

$$A, B, C \quad ::= \quad \mathsf{B} \mid 1 \mid 0 \mid A \times B \mid A + B$$

- **Types**

$$X, Y \ ::= \ \ldots \mid X \xrightarrow{\Sigma} Y \,!\, E \mid X \xrightarrow{\Sigma} Y \,!\, E \not\downarrow S \,@\, C \mid \Sigma \Rightarrow \Sigma' \not\downarrow S \,@\, C$$

- **Values**

$$[\![\Gamma \vdash V : X]\!] : [\![\Gamma]\!] \longrightarrow [\![X]\!]$$

- **User computations**

$$[\![\Gamma \overset{\Sigma}{\models} M : X \,!\, E]\!] : [\![\Gamma]\!] \longrightarrow \mathbf{U}^{\Sigma ! E}[\![X]\!]$$

- **Kernel computations**

$$[\![\Gamma \overset{\Sigma}{\models} K : X \,!\, E \not\downarrow S \,@\, C]\!] : [\![\Gamma]\!] \longrightarrow \mathbf{K}^{\Sigma ! E \not\downarrow S}_{[\![C]\!]}[\![X]\!]$$

- To address coherence, actual semantics in subobject fibrations

Core calculus (very briefly) ctd.

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

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

Fig. 1. Syntax of user and kernel computations

Core calculus (very briefly) ctd.

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

$$\frac{\begin{array}{c}\Gamma \vdash V : \Sigma \Rightarrow \Sigma' \lightning S \mathbin{@} C \qquad \Gamma \vdash W : C \\ \Gamma \models^{\Sigma} M : X \mathbin{!} E \qquad \Gamma, x \mathbin{:} X, c \mathbin{:} C \models^{\Sigma'} N_{ret} : Y \mathbin{!} E' \\ \big(\Gamma, c \mathbin{:} C \models^{\Sigma'} N_e : Y \mathbin{!} E'\big)_{e \in E} \qquad \big(\Gamma \models^{\Sigma'} N_s : Y \mathbin{!} E'\big)_{s \in S}\end{array}}{\begin{array}{l}\Gamma \models^{\Sigma'} \textbf{using } V \mathbin{@} W \textbf{ run } M \textbf{ finally } \{ \textbf{return } x \mathbin{@} c \mapsto N_{ret} \,, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \big(\textbf{raise } e \mathbin{@} c \mapsto N_e\big)_{e \in E} \,, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \big(\textbf{kill } s \mapsto N_s\big)_{s \in S} \, \} : Y \mathbin{!} E'\end{array}}$$

Core calculus (very briefly) ctd.

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

$$\frac{\begin{array}{c}\Gamma \vdash V : \Sigma \Rightarrow \Sigma' \lightning S \mathbin{@} C \qquad \Gamma \vdash W : C \\ \Gamma \mathrel{\models}^{\Sigma} M : X \mathbin{!} E \qquad \Gamma, x\!:\!X, c\!:\!C \mathrel{\models}^{\Sigma'} N_{ret} : Y \mathbin{!} E' \\ \left(\Gamma, c\!:\!C \mathrel{\models}^{\Sigma'} N_e : Y \mathbin{!} E'\right)_{e \in E} \qquad \left(\Gamma \mathrel{\models}^{\Sigma'} N_s : Y \mathbin{!} E'\right)_{s \in S}\end{array}}{\begin{array}{l}\Gamma \mathrel{\models}^{\Sigma'} \mathbf{using}\ V \mathbin{@} W\ \mathbf{run}\ M\ \mathbf{finally}\ \{\ \mathbf{return}\ x \mathbin{@} c \mapsto N_{ret}\ , \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \left(\mathbf{raise}\ e \mathbin{@} c \mapsto N_e\right)_{e \in E}\ , \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \left(\mathbf{kill}\ s \mapsto N_s\right)_{s \in S}\ \} : Y \mathbin{!} E'\end{array}}$$

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

$$\begin{array}{l}\Gamma \mathrel{\models}^{\Sigma'} \mathbf{using}\ {\color{red}R_C} \mathbin{@} W\ \mathbf{run}\ (\mathrm{op}_X\, V\, (x.M)\, (M_e)_{e \in E_{\mathrm{op}}})\ \mathbf{finally}\ F \\ \quad \equiv \mathbf{exec}\ {\color{red}R_{op}[V]} \mathbin{@} W\ \mathbf{finally}\ \{ \\ \qquad\qquad \mathbf{return}\ x \mathbin{@} c' \mapsto \mathbf{using}\ {\color{red}R_C} \mathbin{@} c'\ \mathbf{run}\ M\ \mathbf{finally}\ F\ , \\ \qquad\qquad \left(\mathbf{raise}\ e \mathbin{@} c' \mapsto \mathbf{using}\ {\color{red}R_C} \mathbin{@} c'\ \mathbf{run}\ M_e\ \mathbf{finally}\ F\right)_{e \in E_{\mathrm{op}}}\ , \\ \qquad\qquad \left(\mathbf{kill}\ s \mapsto N_s\right)_{s \in S}\ \} : Y \mathbin{!} E'\end{array}$$

Runners in action

Runners can be vertically nested

Runners can be **vertically nested**

- ```
  using R_FH @ (fopen file_name)
  run (
    using R_FC @ (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 $\Sigma' = \mathbb{0}$) is defined as

  ```
  let R_FC = 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 Σ

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

and a runner for Σ'

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

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

```
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
```

Vertical nesting for instrumentation

Vertical nesting for **instrumentation**

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

- The runner $\mathsf{R}_{\mathsf{Sniffer}}$ implements the same sig. Σ that `m` is using
- As a result, the runner $\mathsf{R}_{\mathsf{Sniffer}}$ 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 → 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
  ```

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

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
  ```
- This is **runtime verification** for `rel`-**monotonic integer state**
- Also possible with **vertical nesting**: MLState ↔ 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[4]
  - Implements the core calculus with few extras
  - The interpreter is directly based on the denotational semantics
  - Top-level containers for running external (OCaml) code

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

Experimenting with the **theory in practice**

- A **small experimental language** COOP[4]
    - 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, ...)

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

Experimenting with the theory in practice

- A **small experimental language** COOP[4]
    - 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

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

Experimenting with the theory in practice

```haskell
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 Sklodowska-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!