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

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/runtime/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/runtime/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 nondeterminism_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

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

[1] We consider runners for signatures, but the work generalises to alg. theories.
[2] In the literature also known as **comodels** for Σ (or an alg. theory).

A natural model of top-level runtime

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

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

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

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

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

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

  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 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**[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

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

  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.,

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

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.,

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

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

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

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

Effectful runners for modular top-levels ctd.

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

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

- 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 of the form

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

  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 { op_1 x_1 → k_1 , ... , op_n x_n → k_n } @ 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

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

we can easily accommodate them in a programming language as

```
let R = runner { op_1 x_1 → k_1 , ... , op_n x_n → k_n } @ C
```

where `k_i` are **kernel computations**, modelled using $\mathbf{K}_C^{\Sigma' ! E_{\mathrm{op}_i} \lightning S}$

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

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

where

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

$$\mathrm{WriteSizeLimitExceeded} \in E_{\mathrm{op}} \qquad S = \mathbb{0}$$

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**:

```
using R @ m1
run m2
finally { return x @ c → m3 , raise e @ c → m4 , kill s → m5 }
```

where

  - `m1` is an **initialiser** user computation producing the initial state
  - `m2` is the user computation being run using the runner `R`
  - `m3`, `m4`, and `m5` are **finaliser** user computations

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

  - `m1` is an **initialiser** user computation producing the initial state
  - `m2` is the user computation being run using the runner `R`
  - `m3`, `m4`, and `m5` are **finaliser** user computations

- `m3` and `m4` **depend on the final state** `c`, but `m5` **does not**

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      → match s with {} }
```

- Importantly,
  - here the file handle is hidden from `m`
  - and `m` can only use `write` of type String $\rightsquigarrow \mathbb{1}$
  - and `fopen` and `fclose` are limited to initialisation-finalisation

Controlled initialisation and finalisation ctd.

- Semantically, in

```
using R @ m1                                                    (* (a) *)
run m2                                                          (* (b) *)
finally { return x @ c → m3 , raise e @ c → m4 , kill s → m5 }  (* (c) *)
```

  - `m1` denotes an element of $\mathbf{U}^{\Sigma'!E'}\, C \stackrel{\text{def}}{=} \mathbf{Free}_{\Sigma'}(C + E')$
    (a **user monad**)
  - `m2` denotes an element of $\mathbf{U}^{\Sigma!E}\, A$
  - `m3` denotes an element of $A \times C \Rightarrow \mathbf{U}^{\Sigma'!E'}\, B$
  - `m4` denotes an element of $E \times C \Rightarrow \mathbf{U}^{\Sigma'!E'}\, B$
  - `m5` denotes an element of $S \Rightarrow \mathbf{U}^{\Sigma'!E'}\, B$

Controlled initialisation and finalisation ctd.

- Semantically, in

```
using R @ m1                                                  (* (a) *)
run m2                                                        (* (b) *)
finally { return x @ c → m3 , raise e @ c → m4 , kill s → m5 } (* (c) *)
```

  - `m1` denotes an element of $\mathbf{U}^{\Sigma'!E'}\, C \stackrel{\text{def}}{=} \mathbf{Free}_{\Sigma'}(C + E')$
    (a **user monad**)
  - `m2` denotes an element of $\mathbf{U}^{\Sigma!E}\, A$
  - `m3` denotes an element of $A \times C \Rightarrow \mathbf{U}^{\Sigma'!E'}\, B$
  - `m4` denotes an element of $E \times C \Rightarrow \mathbf{U}^{\Sigma'!E'}\, B$
  - `m5` denotes an element of $S \Rightarrow \mathbf{U}^{\Sigma'!E'}\, B$

- allowing us to interpret (b) and (c) as the **composite**

$$\mathbf{U}^{\Sigma!E}A \xrightarrow[\text{(b)}]{\mathrm{r}_{A+E}} \mathbf{K}_C^{\Sigma'!E\lightning S}A \xrightarrow[\text{(c)}]{\boxed{\mathrm{m3}}^{\ddagger}} C \Rightarrow \mathbf{U}^{\Sigma'!E'}B$$

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)

- **Values**

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

- **User computations**

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

- **Kernel computations**

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

Core calculus (very briefly)

- **Values**

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

- **User computations**

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

- **Kernel computations**

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

Core calculus (very briefly) ctd.

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

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

Fig. 1. Syntax of user and kernel computations

Core calculus (very briefly) ctd.

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

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

Core calculus (very briefly) ctd.

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

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

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

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

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 WriteSizeLimitExceeded)
                 else  (setenv (s^s'))
  } @ String
  ```
  ```

Runners can be horizontally paired

Runners can be horizontally paired

- Given a runner for Σ

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

and a runner for Σ'

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

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

```
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
```

Ver. nesting for monitoring/instrumentation

Ver. nesting for monitoring/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 **monitoring runner** is defined as

  ```
  let R_Sniffer = runner {
     ... ,
    op a → let c = getenv () in
           op a;                                    (* forwards op outwards *)
           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
  ```

Ver. nesting for monitoring/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 **monitoring runner** is defined as

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

- 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
- This is a passive example, but can easily also do active monitors
  ```

Some other examples

- Various forms of **(ML-style) state**
  - 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 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
  - Implements the core calculus with few extras
  - The interpreter is directly based on the denotational semantics
  - Top-level containers for running external (OCaml) code

Experimenting with the theory in practice

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

Experimenting with the **theory in practice**

- A **small experimental language** COOP
  - 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

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
- Various **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.