Interacting with **external resources**
using **runners** (aka **comodels**)  

Danel Ahman  
(joint work with Andrej Bauer)  

University of Ljubljana, Slovenia  

CHoCoLa meeting, Lyon, 17.10.2019
Today’s plan

- **Computational effects** and **external resources** in PL

- **Issues with standard approaches** to **external resources**

- **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**

- Demonstrate the use of runners through **programming examples**
Computational effects
and
external resources
Computational effects in PL

Using monads (as in Haskell)

```haskell
type St a = String -> (a, String)

instance St Monad where

f :: St a -> St (a, a)
f c = c "x" -> ("y", return ("x", "y"))
```

Using alg. effects and handlers (as in Eff, Frank, Koka)

- `effect Get : unit -> int`
- `effect Put : int -> unit`

```haskell
let g (c:unit -> a! {Get, Put}) : int -> a * int ! {} =
  with st 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?
Computational effects in PL

- Using **monads** (as in **HASKELL**)

  ```haskell
  type St a = String → (a,String)
  instance St Monad where
    ...
  
  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**)

  ```haskell
  effect Get : unit → int
  effect Put : int → unit

  let g (c:unit → a!{Get,Put}) : int → a * int ! {} =
    with st_handler handle (perform (Put 42); c ())
  ```
Computational effects in PL

- Using **monads** (as in **Haskell**)

  \[
  \text{type } \text{St } a = \text{String} \to (a, \text{String}) \\
  \text{instance } \text{St Monad where} \\
  \ldots \\
  \\
  f :: \text{St } a \to \text{St } (a,a) \\
  f c = c >>> (\ x \to c >>> (\ y \to \text{return } (x,y)))
  \]

- Using **alg. effects** and **handlers** (as in **Eff, Frank, Koka**)

  \[
  \text{effect } \text{Get} : \text{unit} \to \text{int} \\
  \text{effect } \text{Put} : \text{int} \to \text{unit} \\
  \\
  \text{let } g (c:\text{unit} \to a!\{\text{Get,Put}\}) : \text{int} \to a \ast \text{int} ! \{} = \\
  \quad \text{with st-handler handle (perform (Put 42); c ())}
  \]

- 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

Declare a signature of monads or algebraic effects, e.g.,

```plaintext
(pervasives.eff)
type IO a
openFile :: FilePath Ñ IOMode Ñ IO Handle

(effect)
RandomInt : int Ñ int
RandomFloat : float Ñ float
```

And then treat them specially in the compiler, e.g., in

```
(eff/src/backends/runtime/eval.ml)
let rec top handle op =
match op with
| Value v Ñ v
| Call (RandomInt, v, k) Ñ top handle (k (Const.of integer (Random.int (Value.to.int v))))
| ...
```

but there are some issues . . .
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., in EFF

```
(* eff/src/backends/runtime/eval.ml *)
let rec top_handle op =
  match op with
  | Value v → v
  | Call (RandomInt, v, k) →
    top_handle (k (Const.of_integer (Random.int (Value.to_int v)))
  | ...
```
External resources in PL

- Declare a **signature of monads or algebraic effects**, e.g.,

```nohighlight
(* 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., in **EFF**

```nohighlight
(* eff/src/backends/runtime/eval.ml *)
let rec top_handle op =
  match op with
  | Value v → v
  | Call (RandomInt, v, k) →
    top_handle (k (Const.of_integer (Random.int (Value.to_int v))))
  | ...
```

but there are **some issues** with that approach ...
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.

```effect
In pervasives.eff:

```

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.Printf 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*:

```ocaml
effect Write : (string*string) -> unit
```

In *eval.ml*, under `let rec top_handle op =` add the case:

```ocaml
| "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\n" str;
|     |     close_out file_handle;
|     |     top_handle (k V.unit_value)
|     )
```
Second issue

- **Lack of linearity** for external resources

```ocaml
define f (s : string) =
  define fh = fopen "foo.txt" in
  fwrite (fh, s^s);
  fclose fh;
  return fh

define g s =
  define fh = f s in fread fh
```

We shall address these kinds of issues indirectly (!), by not introducing a linear typing discipline but instead we make it convenient to hide external resources (addressing stronger typing disciplines in the future).
Second issue

- **Lack of linearity** for external resources

```plaintext
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
```

- We shall address these kinds of issues **indirectly** (!),
  - by **not** introducing a linear typing discipline
  - but instead we make it convenient to **hide external resources**
    (addressing stronger typing disciplines in the future)
Third issue

- **Excessive generality** of effect handlers

```ocaml
let f (s:string) =
  let fh = fopen "foo.txt" in
  fwrite (fh,s^s);
  fclose fh

let h = handler { fwrite (fh,s) k → return () }
```
Third issue

- **Excessive generality** of effect handlers

```ocaml
let f (s:string) =
  let fh = fopen "foo.txt" in
  fwrite (fh,s^s);
  fclose fh

let h = handler { fwrite (fh,s) k → return () }
```

- But misuse of external resources can also be **purely accidental**

```ocaml
let nd_handler =
  handler { choose () k → return (k true ++ k false) }

let g (s1 s2:string) =
  let fh = fopen "foo.txt" in
  let b = choose () in
  if b then (fwrite (fh,s1^s2)) else (fwrite (fh,s2^s1));
  fclose fh
```
Third issue

- **Excessive generality** of effect handlers

```ocaml
let f (s:string) =
  let fh = fopen "foo.txt" in
  fwrite (fh,s^s);
  fclose fh

let h = handler { fwrite (fh,s) k -> return () }
```

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

- Given a signature $\Sigma$ of operation symbols ($A_{op}, B_{op}$ are sets)

  \[\text{op} : A_{op} \rightsquigarrow B_{op}\]

  a runner $\mathcal{R}$ for $\Sigma$ is given by a carrier $\mid \mathcal{R}\mid$ and co-operations

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

  where we think of $\mid \mathcal{R}\mid$ as a set of runtime configurations

---

1. We consider runners for signatures, but the work generalises to alg. theories.
2. In the literature also known as comodels for $\Sigma$ (or for an algebraic theory).
A natural model of top-level runtime

- Given a signature $\Sigma$ of operation symbols ($A_{\text{op}}, B_{\text{op}}$ are sets)

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

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

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

where we think of $|\mathcal{R}|$ as a set of runtime configurations

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

$$\begin{cases} \text{get} : 1 \rightsquigarrow S, & \text{set} : S \rightsquigarrow 1 \end{cases}$$

is given by

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

---

1. We consider runners for signatures, but the work generalises to alg. theories.
2. In the literature also known as comodels for $\Sigma$ (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]
  - **top-level implementation of algebraic effects** in $\mathbf{EFF}$ [Bauer and Pretnar '15]
  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]
  - **top-level implementation of algebraic effects** in \( \text{EFF} \) [Bauer and Pretnar '15] 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**\(^3\) \( r : \text{Free}_\Sigma(-) \rightarrow \text{St}_{|\mathcal{R}|} \)

\(^3\text{Free}_\Sigma(X)\) is the free monad ind. defined with leaves val \( x \) and nodes op(\( a, \kappa \)).
So, runners $\mathcal{R}$ are a natural model of top-level runtime. But what if this runtime is not the runtime? hardware vs OSs OSs vs VMs VMs vs sandboxes browsers vs web pages.

Unfortunately, runners, as defined above, are not readily able to use external resources to signal failure caused by unavoidable circumstances. But is there a useful generalisation that would achieve this?
A natural model of **top-level runtime** ctd.

- So, runners $\mathcal{R}$ are a natural model of **top-level runtime**

- But what if this runtime is not **the** runtime?
  - hardware vs OSs
  - OSs vs VMs
  - VMs vs sandboxes
  - browsers vs web pages
  - ...
A natural model of **top-level runtime** ctd.

- So, runners $\mathcal{R}$ are a natural model of **top-level runtime**
- But what if this runtime is not **the** runtime?
  - hardware vs OSs
  - OSs vs VMs
  - VMs vs sandboxes
  but also
  - browsers vs web pages
  - ...
- 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

Møgelberg and Staton usefully observed that a runner $R$ is equivalently simply a family of generic effects for $\text{St} \mid R$, i.e.,

```
\text{op}_R : A \text{op}_{\text{ÝÑ}} \text{St} \mid R \mid B \text{op}_{\bar{\text{P}} \Sigma}
```

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

```
\text{op}_R : A \text{op}_{\text{ÝÑ}} T B \text{op}_{\bar{\text{P}} \Sigma}
```

The one-to-one correspondence with monad morphisms $r : \text{Free} \Sigma p \text{ÝÑ} T$ simply amounts to the universal property of free models, i.e.,

$q \approx \eta X x r X p \text{val}_a, \kappa q q p r X \circ q : p \text{op}_R a q$

Observe that $\kappa$ appears in a tail call position on the right!
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 $\text{St}_{|\mathcal{R}|}$, i.e.,

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

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

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

The one-to-one correspondence with monad morphisms $r : \text{Free} \Sigma p \overline{\text{op}} q \longrightarrow T$ simply amounts to the universal property of free models, i.e.,

$$r X p \text{val} x q \overset{\eta X x}{\cong} r X p \overline{\text{op}} p \kappa q q \overset{p \overline{\text{op}} R a q}{\cong}$$

Observe that $\kappa$ appears in a tail call position on the right!
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 $\text{St}_{\mathcal{R}}$, i.e.,

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

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

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

Observe that $\kappa$ appears in a tail call position on the right!
Effectful runners for modular top-levels

- Møgelberg and Staton usefully observed that a **runner** $\mathcal{R}$ is equivalently simply a family of **generic effects** for $\mathbf{St}_{|\mathcal{R}|}$, i.e.,
  \[
  \left(\overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{St}_{|\mathcal{R}|} B_{\text{op}}\right)_{\text{op} \in \Sigma}
  \]

- Building on this, we define a **T-runner** $\mathcal{R}$ for $\Sigma$ to be given by
  \[
  \left(\overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{T} B_{\text{op}}\right)_{\text{op} \in \Sigma}
  \]

- The one-to-one correspondence with **monad morphisms** $r : \text{Free}_\Sigma(-) \longrightarrow \mathbf{T}$ simply amounts to the **universal property of free models**, i.e.,
  \[
  r_X (\text{val} x) = \eta_X x \quad \quad r_X (\text{op}(a, \kappa)) = \left(r_X \circ \kappa\right)^\dagger (\overline{\text{op}}_{\mathcal{R}} a)_{\text{op}_{\mathcal{M}}(a, r_X \circ \kappa)}
  \]
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 $\text{St}_{|\mathcal{R}|}$, i.e.,
  \[
  \left(\overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \text{St}_{|\mathcal{R}|} B_{\text{op}}\right)_{\text{op} \in \Sigma}
  \]

- Building on this, we define a $\mathbf{T}$-runner $\mathcal{R}$ for $\Sigma$ to be given by
  \[
  \left(\overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow \mathbf{T} B_{\text{op}}\right)_{\text{op} \in \Sigma}
  \]

- The one-to-one correspondence with monad morphisms $r : \text{Free}_\Sigma(-) \longrightarrow \mathbf{T}$ simply amounts to the universal property of free models, i.e.,
  \[
  r_X (\text{val} x) = \eta_X x \quad r_X (\text{op}(a, \kappa)) = \underbrace{(r_X \circ \kappa)}_{\text{op}_{\mathcal{M}}(a, r_X \circ \kappa)} + \overline{\text{op}}_{\mathcal{R}} a
  \]

- Observe that $\kappa$ appears in a tail call position on the right!
Effectful runners for modular top-levels ctd.

- What would be a useful class of monads \( T \) to use?
Effectful runners for modular top-levels ctd.

- What would be a **useful class of monads** $T$ to use?

- We want a runner to be a bit like a **kernel of an OS**, i.e., to
  1. provide management of **(internal) resources**
  2. use further **external resources**
  3. **signal failure** caused by unavoidable circumstances

Algebraically (and pragmatically), this amounts to taking

(i) $\text{getenv} : \mathbb{1} \Rightarrow C$ & $\text{setenv} : C \Rightarrow \mathbb{1}$

(ii) $\text{op} : A \text{op} \Rightarrow B \text{op} (\text{op} P \Sigma_1, \text{for some external } \Sigma_1)$

(iii) $\text{kill} : S \Rightarrow 0$

s.t., (i) satisfy state equations; and (ii) and (iii)

The induced monad is then isomorphic to $T X = \text{Free } \Sigma_1 p X \hat{C} q S \hat{C}$
Effectful runners for modular top-levels ctd.

- What would be a **useful class of monads** $T$ 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) $\text{getenv} : 1 \rightsquigarrow C$ & $\text{setenv} : C \rightsquigarrow 1$
  - (ii) $\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$ (op $\in \Sigma'$, for some external $\Sigma'$)
  - (iii) $\text{kill} : S \rightsquigarrow 0$

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 $T$ 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 : 1 \rightsquigarrow C` & `setenv : C \rightsquigarrow 1`
  - (ii) `op : A_{op} \rightsquigarrow B_{op}` \hspace{1cm} (\text{\textit{op} \in \Sigma', for some external \Sigma'})
  - (iii) `kill : S \rightsquigarrow 0`
  
  s.t., (i) satisfy state equations; and (i) commute with (ii) and (iii)

- The induced monad is then isomorphic to

\[ T X \overset{\text{def}}{=} C \Rightarrow \text{Free}_{\Sigma'}((X \times C) + S) \]
Effectful runners for modular top-levels ctd.

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

$$\bigg(\overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \rightarrow C \Rightarrow \text{Free}_{\Sigma'}((B_{\text{op}} \times C) + S)\bigg)_{\text{op} \in \Sigma}$$
Effectful runners for modular top-levels ctd.

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

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

- 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** \( R \) for \( \Sigma \) are then of the form

\[
\left( \overline{\text{op}}_R : A_{\text{op}} \rightarrow C \Rightarrow \text{Free}_\Sigma \left( (B_{\text{op}} \times C) + S \right) \right)_{\text{op} \in \Sigma}
\]

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

- **Our solution:** consider signatures \( \Sigma \) with operation symbols

\[
\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}} + E_{\text{op}}
\]
Effectful runners for modular top-levels ctd.

- The corresponding **T-runners** $\mathcal{R}$ for $\Sigma$ are then of the form
  \[
  \left( \overline{\text{op}}_{\mathcal{R}} : A_{\text{op}} \longrightarrow C \Rightarrow \text{Free}_{\Sigma'}((B_{\text{op}} \times C) + S) \right)_{\text{op} \in \Sigma}
  \]

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

- **Our solution:** consider signatures $\Sigma$ with operation symbols
  \[
  \text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}} + E_{\text{op}} \quad \text{(which we write as} \quad \text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}} ! E_{\text{op}})\]
Effectful runners for modular top-levels ctd.

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

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

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

- **Our solution:** consider signatures $\Sigma$ with operation symbols

$$\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}} + E_{\text{op}} \quad \text{(which we write as} \quad \text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}} ! E_{\text{op}})$$

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

$$\left(\overline{\text{op}_{\mathcal{R}}} : A_{\text{op}} \longrightarrow K_{C}^{\Sigma' ! E_{\text{op}} \hat{\downarrow} S} B_{\text{op}}\right)_{\text{op} \in \Sigma}$$

where we call $K_{C}^{\Sigma' ! E_{\text{op}} \hat{\downarrow} S}$ a **kernel monad** (the sum of $T$ and excs.)

$$K_{C}^{\Sigma' ! E_{\text{op}} \hat{\downarrow} S} B_{\text{op}} \overset{\text{def}}{=} C \Rightarrow \text{Free}_{\Sigma'}(((B_{\text{op}} + E_{\text{op}}) \times C) + S)$$
T-runners as a **programming construct**
(towards a core calculus for runners)
T-runners as a programming construct

- First, we include **T-runners** for $\Sigma$

$$
\left( \overline{\text{op}} \mathcal{R} : A_{\text{op}} \rightarrow K^\Sigma_{\text{C}} \end{array} \right)_{\text{op} \in \Sigma}$$

in our language **as values**, and **co-ops. as kernel code**, i.e.,

```
let R = runner \{ op_1 \ x_1 \rightarrow K_1 \; , \; \ldots \; , \; op_n \ x_n \rightarrow K_n \} @ C
```
T-runners as a programming construct

• First, we include T-runners for $\Sigma$

\[
\left( \overline{\text{op}} \stackrel{\text{R}}{\rightarrow} A_{\text{op}} \longrightarrow K_{\Sigma'}^{E_{\text{op}} S} B_{\text{op}} \right)_{\text{op} \in \Sigma}
\]

in our language as values, and co-ops. as kernel code, i.e.,

\[
\text{let R = runner} \{ \ \text{op}_1 \ x_1 \rightarrow K_1, \ldots, \text{op}_n \ x_n \rightarrow K_n \} \ @ C
\]

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

\[
\text{let R_{FH} = runner} \{
\text{write s} \rightarrow \text{if} \ (\text{length s} > \text{maxSize})
\text{then} (\text{raise WriteSizeExceeded})
\text{else} \ (\text{let fh = getenv () in}
\text{if} \ (\text{isValid fh}) \text{then} (\text{fwrite (fh,s)}) \text{else} (\text{kill IOError}))
\} \ @ \text{FileHandle}
\]

where

\[
\Sigma \overset{\text{def}}{=} \{ \text{write : String } \rightsquigarrow 1 ! E \cup \{\text{WriteSizeExceeded}\} \}
\]

\[
(\text{fwrite : FileHandle } \times \text{String } \rightsquigarrow 1 ! E) \in \Sigma' \quad S = \{ \text{IOError} \} 
\]
Controlled initialisation and finalisation

Recall that the components $r^X$ of the monad morphism initialisation and finalisation induced by a $T$-runner $R$ are all tail-recursive.

We make use of it to enable programmers to run user code:

using $R$ ~ Ms init run M finally {
  return x ~ Ms ret, ... ...
  raise e ~ Ms e ... ..., ...
  kill s ~ Ms s ... ...
}

where (a user monad) $Ms$ are user code, modelled using $U \Sigma E X \def \Free \Sigma p^X \E q^M$.

$M$ 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*

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

$$r : \text{Free}_\Sigma(-) \rightarrow T$$

induced by a $T$-runner $\mathcal{R}$ are all *tail-recursive*
Controlled *initialisation and finalisation*

- Recall that the components $r_X$ of the monad morphism $r: \text{Free}_\Sigma(-) \rightarrow T$ induced by a $T$-runner $R$ are all *tail-recursive*.

- We make use of it to enable programmers to run user code:

```
using R @ M_init
run M
finally {return x @ c \rightarrow M_{ret}, \ldots raise e @ c \rightarrow M_e \ldots, \ldots kill s \rightarrow M_s \ldots}
```

where

- Ms are *user code*, modelled using $U^{\Sigma!E} X \overset{\text{def}}{=} \text{Free}_\Sigma(X + E)$
- (a user monad)
Controlled initialisation and finalisation

- Recall that the components $r_X$ of the monad morphism induced by a $T$-runner $R$ are all tail-recursive

- We make use of it to enable programmers to run user code:

```plaintext
using R @ M_init
run M
finally {return x @ c → M_ret, ... raise e @ c → M_e, ... , ... kill s → M_s, ...}
```

where

- Ms are user code, modelled using $U^E X ≝ Free_Σ(X + E)$
- $M_{init}$ 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

\[
\text{initialisation} \quad \xrightarrow{\text{"o"}} \quad r : \text{Free}_\Sigma(-) \longrightarrow T \quad \xleftarrow{\text{"o"}} \quad \text{finalisation}
\]

induced by a $T$-runner $R$ are all **tail-recursive**

- We make use of it to enable programmers to **run user code**:

```plaintext
using R @ M_init
run M
finally {return x @ c \rightarrow M_{ret} , ... raise e @ c \rightarrow M_e ... , ... kill s \rightarrow M_s ...}
```

where

(a **user monad**)

- Ms are **user code**, modelled using $U^{\Sigma ! E} X \defeq \text{Free}_\Sigma(X + E)$
- $M_{\text{init}}$ produces the **initial kernel state**
- $M$ is the user code being **run using the runner** $R$
- $M_{\text{ret}}, M_e, M_s$ **finalise** for return values, exceptions, and signals
- $M_{\text{ret}}$ 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-esque `with` construct

```haskell
with fileName do M =
  using R@FH (fopen fileName)
run M
finally {
  return x @ fh -> fclose fh; return x ,
  raise WriteSizeExceeded @ fh -> fclose fh; return () ,
  raise e @ fh -> fclose fh; raise e , (* other exceptions in E are re-raised *)
  kill IOError -> ... }
```
Controlled initialisation and finalisation ctd.

- For instance, we can define a **Python**-esque **with construct**

```plaintext
with fileName do M

= using R_{FH} @ (fopen fileName)
run M
finally {
    return x @ fh → fclose fh; return x,
    raise WriteSizeExceeded @ fh → fclose fh; return () ,
    raise e @ fh → fclose fh; raise e , (∗ other exceptions in E are re-raised ∗)
    kill IOError → ... }
```

- the **file handle is hidden** from **M**
- **M** **can only call** write : String ↦ 1 ! E ∪ {WriteSizeExceeded}
  but **not** (the external operations) fopen, fclose, and fwrite
- fopen and fclose are **limited to initialisation-finalisation**
- **M** can itself also catch WriteSizeExceeded to **re-try writing**
A core calculus for programming with runners
Core calculus (syntax)

Ground types (types of operations and kernel state)

\[ A, B, C :: B | 1 | 0 | A \ ^ B | A \ ` B \]

Types

\[ X, Y :: B | 1 | 0 | X \ ^ Y | X \ ` Y | X \ \Sigma \ \Pi \ Y ! E | X \ \Sigma \ \Pi \ Y ! E \ \mathbin{\bowtie} S \ \mathbin{\otimes} C \]

Values

\[ \Gamma \$ V : X \]

User computations

\[ \Gamma \$ \Sigma M : X ! E \]

Kernel computations

\[ \Gamma \$ \Sigma K : X ! E \ \mathbin{\bowtie} S \ \mathbin{\otimes} C \]


Core calculus (syntax)

- **Ground types** (types of operations and kernel state)

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

- **Types**

\[ X, Y ::= B \mid 1 \mid 0 \mid X \times Y \mid X + Y \]
\[ \mid X \xrightarrow{\Sigma} Y ! E \]
\[ \mid X \xrightarrow{\Sigma} Y ! E \Downarrow S @ C \]
\[ \mid \Sigma \Rightarrow \Sigma' \Downarrow S @ C \]

- **Values**

\[ \Gamma \vdash V : X \]

- **User computations**

\[ \Gamma \triangleright M : X ! E \]

- **Kernel computations**

\[ \Gamma \triangleright K : X ! E \Downarrow S @ C \]
Core calculus (user computations)

\[ M, N ::= \text{return } V \]

\[ \text{try } M \text{ with } \{ \text{return } x \mapsto N, (\text{raise } e \mapsto N_e)_{e \in E} \} \]

\[ VW \]

\[ \text{match } V \text{ with } \{ \langle x, y \rangle \mapsto M \} \]

\[ \text{match } V \text{ with } \{ \} \_X \]

\[ \text{match } V \text{ with } \{ \text{inl } x \mapsto M, \text{inr } y \mapsto N \} \]

\[ \text{op}_X (V, (x . M), (N_e)_{e \in E_{op}}) \]

\[ \text{raise}_X e \]

\[ \text{using } V @ W \text{ run } M \text{ finally } \{ \]

\[ \text{return } x @ c \mapsto N, \]

\[ (\text{raise } e @ c \mapsto N_e)_{e \in E}, \]

\[ (\text{kill } s \mapsto N_s)_{s \in S} \} \]

\[ \text{kernel } K @ V \text{ finally } \{ \]

\[ \text{return } x @ c \mapsto N, \]

\[ (\text{raise } e @ c \mapsto N_e)_{e \in E}, \]

\[ (\text{kill } s \mapsto N_s)_{s \in S} \} \]

value
exception handler
application
product elimination
empty elimination
sum elimination
operation call
raise exception
run

switch to kernel mode

Fig. 2. Values, user and kernel computations of coop
Core calculus (kernel computations)

\[ K, L ::= \text{return}_C V \]

\[ \text{try } K \text{ with } \{ \text{return } x \mapsto L, (\text{raise } e \mapsto L_e)_{e \in E} \} \]

\[ V W \]

\[ \text{match } V \text{ with } \{ \langle x, y \rangle \mapsto K \} \]

\[ \text{match } V \text{ with } \{ \}_{X@C} \]

\[ \text{match } V \text{ with } \{ \text{inl } x \mapsto K, \text{inr } y \mapsto L \} \]

\[ \text{op}_{X@C}(V, (x \cdot K), (L_e)_{e \in E_{\text{op}}}) \]

\[ \text{raise}_{X@C} e \]

\[ \text{kill}_{X@C} s \]

\[ \text{getenv}_C(c \cdot K) \]

\[ \text{setenv}(V, K) \]

\[ \text{user } M \text{ with } \{ \text{return } x \mapsto K, (\text{raise } e \mapsto L_e)_{e \in E} \} \]

value

exception handler

application

product elimination

empty elimination

sum elimination

operation call

raise exception

send signal

get state

set state

switch to user mode

Fig. 2. Values, user and kernel computations of coop
Core calculus (type system and eq. theory)

For example, the typing rule for running user comps.

\[ \Gamma, V : \Sigma \vdash \Gamma, x : X, c : C \vdash \text{return } x @ c : \Sigma \]

and the main \( \beta \)-equation for running user comps.

\[ \Gamma, V \vdash \Gamma, x : X, c : C \vdash \text{return } x @ c : \Sigma \]
Core calculus (type system and eq. theory)

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

\[
\Gamma \vdash V : \Sigma \Rightarrow \Sigma' \downarrow S \@ C \quad \Gamma \vdash W : C \\
\Gamma \not\vdash M : X ! E \quad \Gamma, x : X, c : C \not\vdash N_{\text{ret}} : Y ! E' \\
(\Gamma, c : C \not\vdash N_e : Y ! E')_{e \in E} \quad (\Gamma \not\vdash N_s : Y ! E')_{s \in S} \\
\frac{\Gamma \not\vdash using \ V \@ W \ run \ M \ finally \ \{ \ \text{return} \ x \@ c \mapsto N_{\text{ret}} \ , \ \\
(\text{raise} \ e \@ c \mapsto N_e)_{e \in E} \ , \ \\
(\text{kill} \ s \mapsto N_s)_{s \in S} \ \} : Y ! E'}{\Gamma \not\vdash using \ V \@ W \ run \ M \ finally \ \{ \ \text{return} \ x \@ c \mapsto N_{\text{ret}} \ , \ \\
(\text{raise} \ e \@ c \mapsto N_e)_{e \in E} \ , \ \\
(\text{kill} \ s \mapsto N_s)_{s \in S} \ \} : Y ! E'}
\]
Core calculus (type system and eq. theory)

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

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

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

\[
\Gamma \not\vdash \text{using } R @ W \text{ run } (\text{op}_X(V, (y.M), (M_e)_{e \in E_{\text{op}}))) \text{ finally } F
\]
\[
\equiv \text{kernel } K_{\text{op}}[V/x_{\text{op}}] @ W \text{ finally } \{
\]
\[
\text{return } y @ c' \mapsto \text{using } R @ c' \text{ run } M \text{ finally } F , \quad (\text{raise } e @ c' \mapsto \text{using } R @ c' \text{ run } M_e \text{ finally } F)_{e \in E_{\text{op}}} , \quad
\]
\[
(\text{kill } s \mapsto N_s)_{s \in S} \} : Y ! E'
\]
Core calculus (type system and eq. theory)

- The calculus also includes **subtyping**, and **subsumption rules**

\[
\Gamma \vdash V : A \quad A <: B \\
\Gamma \vdash V : B
\]

\[
\Gamma \models M : A ! E \quad \Sigma \subseteq \Sigma' \quad A <: B \quad E \subseteq E' \\
\Gamma \models' M : B ! E'
\]

\[
\Gamma \models K : A ! E \nonumber \downarrow \quad S \bowtie C \quad \Sigma \subseteq \Sigma' \quad A <: B \quad E \subseteq E' \quad S \subseteq S' \quad C = C' \\
\Gamma \models' K : B ! E' \nonumber \downarrow \quad S' \bowtie C'
\]

We use \(C \bowtie C\) to have (standard) proof-irrelevant subtyping. Otherwise, instead of just \(C <: C\), we would need a lens \(C \bowtie \emptyset C\).
Core calculus (type system and eq. theory)

- The calculus also includes subtyping, and subsumption rules

\[
\frac{\Gamma \vdash V : A \quad A <: B}{\Gamma \vdash V : B}
\]

\[
\frac{\Gamma \not\vdash M : A ! E \quad \Sigma \subseteq \Sigma' \quad A <: B \quad E \subseteq E'}{\Gamma \not\vdash M : B ! E'}
\]

\[
\frac{\Gamma \not\vdash K : A ! E \not\not S \circ C \quad \Sigma \subseteq \Sigma'}{\frac{A <: B \quad E \subseteq E' \quad S \subseteq S' \quad C = C'}{\Gamma \not\vdash K : B ! E' \not\not S' \circ C'}}
\]

- We use \( C = C' \) to have (standard) proof-irrelevant subtyping

- Otherwise, instead of just \( C <: C' \), we would need a lens \( C' \leftrightarrow C \)
Core calculus (semantics)
Core calculus (semantics)

- **Monadic semantics**, for concreteness in $\textbf{Set}$, using
  
  - **user monads** $U^{\Sigma!E} X \overset{\text{def}}{=} \text{Free}_\Sigma(X + E)$
  
  - **kernel monads** $K^{\Sigma!E \downarrow S} C X \overset{\text{def}}{=} C \Rightarrow \text{Free}_\Sigma(((X + E) \times C) + S)$
Core calculus (semantics)

- **Monadic semantics**, for concreteness in *Set*, using
  
  - user monads $U^{Σ!E} X \overset{\text{def}}{=} \text{Free}_Σ(X + E)$
  
  - kernel monads $K^{Σ!E \downarrow S} C X \overset{\text{def}}{=} C \Rightarrow \text{Free}_Σ(((X + E) \times C) + S)$

- (At a high level) the judgments are interpreted as

\[
\llbracket \Gamma \vdash V : X \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket X \rrbracket \\
\llbracket \Gamma \vdash M : X ! E \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow U^{Σ!E} \llbracket X \rrbracket \\
\llbracket \Gamma \vdash K : X ! E \downarrow S \otimes C \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow K^{Σ!E \downarrow S} \llbracket C \rrbracket \llbracket X \rrbracket
\]
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:

  \[
  \begin{array}{c}
  \llbracket \Gamma \rrbracket \quad \llbracket \Gamma \vdash K : X ! E \dashv S @ C \rrbracket \quad \llbracket \kappa \Sigma ! E \dashv S \rrbracket \llbracket X \rrbracket \\
  \subseteq \\
  \llbracket \Gamma^s \rrbracket \\
  \llbracket \Gamma^s \dashv K : X^s @ C \rrbracket \\
  \end{array}
  \]

  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

\[
\begin{align*}
\text{if } \Gamma \vdash K : X \land E \Downarrow S \land C \text{ then } & \\
\Gamma \vdash K : X \land E \Downarrow S \land C & \Rightarrow K^{\Sigma ! E \Downarrow S} [X] \\
\Gamma^s \vdash K : X^s \land C & \Rightarrow K^{\Sigma ! E \Downarrow S + \{x\}} [X^s]
\end{align*}
\]

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

- No essential obstacles to extending to $\text{Sub(Cpo)}$ and beyond
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

\[
\begin{align*}
\llbracket \Gamma \rrbracket & \quad \llbracket \Sigma \vdash K : X ! E \Downarrow S @ C \rrbracket \\
& \quad \xrightarrow{\llbracket \Gamma \vdash K : X @ C \rrbracket} \\
& \quad \llbracket K \Sigma ! E \Downarrow S \rrbracket \llbracket X \rrbracket \\
\end{align*}
\]

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

- No essential obstacles to extending to \( \text{Sub} (\text{Cpo}) \) and beyond

- **Ground type restriction** on \( C \) needed to stay within \( \text{Sub} (\ldots) \)
  - Otherwise, analogously to subtyping, we’d need **lenses** instead
Implementing runners
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

- 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 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, ...)

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

coop [ˈkuːp/] – a cage where small animals are kept, especially chickens
Runners in action
Runners can be *vertically nested*
Runners can be **vertically nested**

- **using** $R_{FH}$ (@ (fopen fileName))
  
  run (  
    **using** $R_{FC}$ (@ (return ""))
    run M
    finally {
      return x @ str → write str; return x ,
      **raise** WriteSizeExceeded @ str → write str; **raise** WriteSizeExceeded 
    }
  )
  
  finally {
    return x @ fh → ... , **raise** e @ fh → ... , **kill** IOError → ...
  }

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

**let** $R_{FC} = \text{runner}$ {
  write str' → let str = getenv () in
    if (length (str^str')) > max) **then** (**raise** WriteSizeExceeded)
    else (**setenv** (str^str'))
} @ String
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,toStr c); fclose fh; return x }

where the **instrumenting runner** is defined as

```plaintext
let $R_{Sniffer} = runner \{
  ... ,
  op a → let c = getenv () in
  setenv (c + 1);
  op a ,
  ... \}
```

- The runner $R_{Sniffer}$ implements the same sig. $\Sigma$ that $M$ is using
- As a result, the runner $R_{Sniffer}$ is invisible from $M$ ’s viewpoint
Vertical nesting for active monitoring
Vertical nesting for **active monitoring**

- First, we define a runner for **integer-valued ML-style state** as

  ```haskell
  type IntHeap = (Nat \to (Int + 1)) \times Nat
  type Ref = Nat
  
  let R_{IntState} = runner {
    alloc x \to let h = getenv () in
    let (r,h') = heapAlloc h \times in
    setenv h';
    return r ,
  }
  
  deref r \to let h = getenv () in
  match (heapSel h r) with
  | inl x \to return x
  | inr () \to kill ReferenceDoesNotExist ,
  
  assign r y \to let h = getenv () in
  match (heapUpd h r y) with
  | inl h' \to setenv h'
  | inr () \to kill ReferenceDoesNotExist
  }
  @ IntHeap
  ```
Vertical nesting for **active monitoring** ctd.

- Next we define a runner for **monotonicity layer** on top of $R_{\text{IntState}}$.
Vertical nesting for active monitoring ctd.

- Next we define a runner for **monotonicity layer** on top of $R_{\text{IntState}}$

```ocaml
type MonMemory = Ref → ((Int → Int → Bool) + 1)

let $R_{\text{MonState}}$ = runner {
  mAlloc x rel → let r = alloc x
                  let m = getenv () in
                  setenv (memAdd m r rel);
                  return r,

  mDeref r → deref r ,

  mAssign r y → let x = deref r
                 let m = getenv () in
                 match (memSel m r) with
                 | inl rel → if (rel x y)
                       then (assign r y)
                       else (raise MonotonicityViolation)
                 | inr → kill PreorderDoesNotExist
}

@ MonMemory
```
Vertical nesting for active monitoring ctd.

- We can then perform **runtime monotonicity verification** as
Vertical nesting for active monitoring ctd.

- We can then perform runtime monotonicity verification as

```plaintext
using \( R_{\text{IntState}} \) ((fun _ \rightarrow \text{inr} ()) , 0) (* init. empty ML–style heap *)
run (

using \( R_{\text{MonState}} \) (fun _ \rightarrow \text{inr} ()) (* init. empty preorders memory *)
run (

let r = mAlloc 0 (\( \leq \)) in
mAssign r 1;
mAssign r 0; (* \( R_{\text{MonState}} \) raises MonotonicityViolation exception *)
mAssign r 2
)
finally { ... , raise MonotonicityViolation @ m \rightarrow ... , ... }
)
finally { ... }
```
Runners can also be horizontally paired.
Runners can also be **horizontally paired**

- Given runners for $\Sigma$ and $\Sigma'$

```ocaml
let R₁ = runner { ... , op₁ᵢ x → K₁ᵢ , ... } @ C₁
let R₂ = runner { ... , op₂ⱼ x → K₂ⱼ , ... } @ C₂
```

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

```ocaml
let R = runner { ... ,
  op₁ᵢ x → let (c,c') = getenv () in
    user (kernel (K₁ᵢ x) @ c finally {
      return y @ c'' → return (inl (inl y,c''))),
      raise e @ c'' → return (inl (inr e,c''))),
      kill s → return (inr s) }
    finally {
      return (inl (inl y,c'')) → setenv (c'',c'); return y,
      return (inl (inr e,c'')) → setenv (c'',c'); raise e,
      return (inr s) → kill s },
  ... ,
  op₂ⱼ x → ...,
  ... } @ C₁ × C₂
```

For instance, this way we can build a runner for IO and state
Runners can also be horizontally paired

- Given runners for \( \Sigma \) and \( \Sigma' \)

\[
\text{let } R_1 = \text{runner } \{ \ldots, \text{op}_{1i} \times \to K_{1i}, \ldots \} \circ C_1
\]
\[
\text{let } R_2 = \text{runner } \{ \ldots, \text{op}_{2j} \times \to K_{2j}, \ldots \} \circ C_2
\]

we can pair them to get a runner for \( \Sigma + \Sigma' \)

\[
\text{let } R = \text{runner } \{ \ldots, \]
\[
\quad \text{op}_{1i} \times \to \text{let } (c,c') = \text{getenv } () \text{ in}
\quad \text{user } (\text{kernel } (K_{1i} \times) @ c \text{ finally } \{
\quad \quad \text{return } y @ c'' \to \text{return } (\text{inl } (\text{inl } y,c'')),
\quad \quad \text{raise } e @ c'' \to \text{return } (\text{inl } (\text{inr } e,c'')),
\quad \quad \text{kill } s \to \text{return } (\text{inr } s) \}
\quad \}\text{ finally } \{
\quad \quad \text{return } (\text{inl } (\text{inl } y,c'')) \to \text{setenv } (c'',c'); \text{return } y,
\quad \quad \text{return } (\text{inl } (\text{inr } e,c'')) \to \text{setenv } (c'',c'); \text{raise } e,
\quad \quad \text{return } (\text{inr } s) \to \text{kill } s \},
\]
\[
\quad \text{op}_{2j} \times \to \ldots, \quad (* \text{ analogously to above, just on 2nd comp. of state } *)
\]
\[
\quad \ldots \} \circ C_1 \times C_2
\]

- For instance, this way we can build a runner for IO and state
Other examples (in Haskell)

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 applied in their lexical context

A runner that treats amb. fun. application as a co-operation

Amb. funs. are stored in a context-depth-sensitive heap

The appl. co-operation restores the heap to the lexical context

If the host language allows it, we use GADTs, etc for safety

Some examples extract a footprint from a larger memory
Other examples (in Haskell)

- 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 applied in their lexical context
  - a runner that treats amb. fun. application as a co-operation
  - amb. funs. are stored in a context-depth-sensitive heap
  - the appl. co-operation restores the heap to the lexical context
Other examples (ambient functions)

module Control.Runner.Ambients

...  

ambCoOps :: Amb a -> Kernel sigAmbHeap a
ambCoOps (Bind f) =
  do h <- getEnv;
     (f,h') <- return (ambHeapAlloc h f);
     setEnv h';
  return f
ambCoOps (Apply f x) =
  do h <- getEnv;
     (f,d) <- return (ambHeapSel h f (depth h));
     user
     (run
         ambRunner
         (return (h {depth = d}))
         (f x)
         ambFinaliser)
  return
ambCoOps (Rebind f g) =
  do h <- getEnv;
     setEnv (ambHeapUpd h f g)

ambRunner :: Runner '[Amb] sigAmbHeap
ambRunner = mkRunner ambCoOps

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**
- **Ongoing** and **future**: lenses in subtyping and semantics, cat. of runners, handlers, case studies, refinement typing, compilation, . . .

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.
Core calculus (semantics ctd.)

\[
\begin{align*}
\llbracket \Gamma \vdash_{\Sigma'} \text{using } V \& W \text{ run } M \text{ finally } \{ & \text{return } x \oplus c \mapsto N_{\text{ret}} , \\
& (\text{raise } e \oplus c \mapsto N_e)_{e \in E} , \\
& (\text{kill } s \mapsto N_s)_{s \in S} \} : Y ! E' \rrbracket_{\gamma} \overset{\text{def}}{=} \ldots 
\end{align*}
\]

- \(\llbracket V \rrbracket_{\gamma} = \mathcal{R} = \left( \overline{\text{op}_R : \llbracket A_{\text{op}} \rrbracket \rightarrow K^{\Sigma' ! E_{\text{op}} \gamma S_{\text{op}} \llbracket B_{\text{op}} \rrbracket}_{\text{op}} \right)_{\text{op} \in \Sigma} \)
- \(\llbracket W \rrbracket_{\gamma} \in \llbracket C \rrbracket\)
- \(\llbracket M \rrbracket_{\gamma} \in U^{\Sigma!E} \llbracket A \rrbracket\)
- \(\llbracket \text{return } x \oplus c \mapsto N_{\text{ret}} \rrbracket_{\gamma} \in \llbracket A \rrbracket \times \llbracket C \rrbracket \rightarrow U^{\Sigma' ! E'} \llbracket B \rrbracket\)
- \(\llbracket (\text{raise } e \oplus c \mapsto N_e)_{e \in E} \rrbracket_{\gamma} \in E \times \llbracket C \rrbracket \rightarrow U^{\Sigma' ! E'} \llbracket B \rrbracket\)
- \(\llbracket (\text{kill } s \mapsto N_s)_{s \in S} \rrbracket_{\gamma} \in S \rightarrow U^{\Sigma' ! E'} \llbracket B \rrbracket\)

- allowing us to use the free model property to get

\[
U^{\Sigma!E} \llbracket A \rrbracket \xrightarrow{r^{\llbracket A \rrbracket} + E} K^{\Sigma' ! E_S S_{\llbracket A \rrbracket}} \llbracket C \rrbracket \xrightarrow{(\lambda N_{\text{ret}}\rrbracket_{\gamma})^\dagger} \llbracket C \rrbracket \Rightarrow U^{\Sigma' ! E'} \llbracket B \rrbracket
\]

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