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

Computational effects in PL

- Using **monads** (as in 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)

```
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)

```
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)

```
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 ())
```

- 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., 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.,

```
(* 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))))
  | ...
```

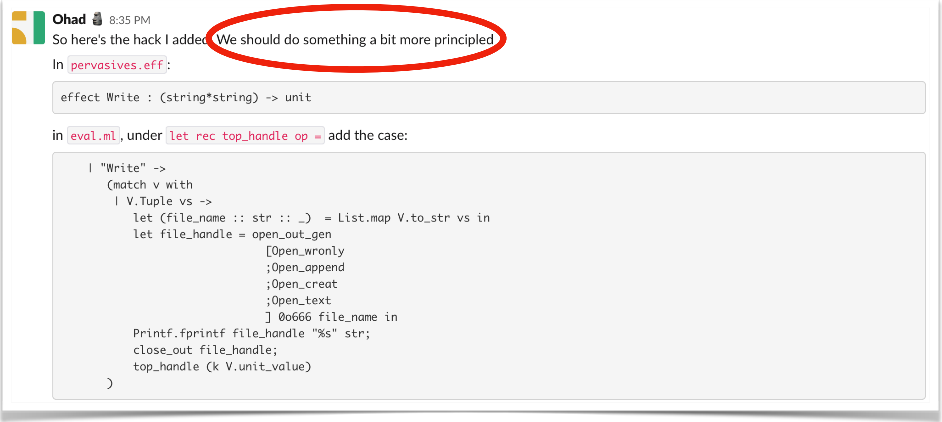
but there are **some issues** with that approach . . .

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

In `pervasives.eff`:

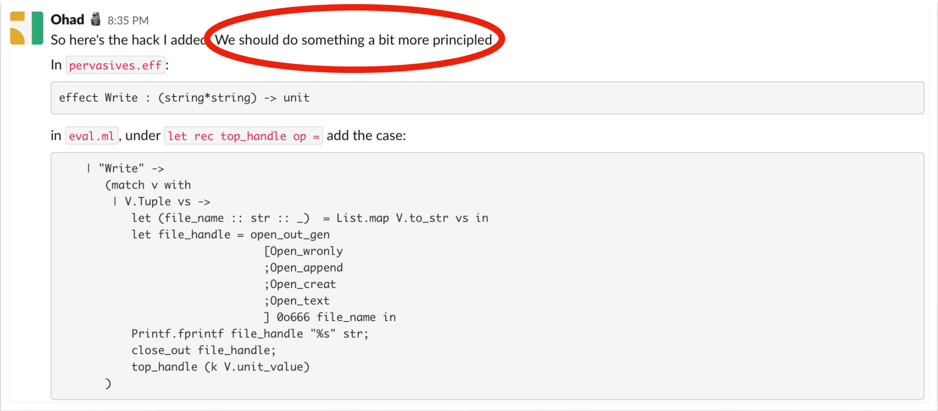
```
effect Write : (string*string) -> unit
```

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 work — a principled modular (co)algebraic approach!

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 any more ! *)
```

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 any more ! *)
```

- 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

```
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

```
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**

```
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

```
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**[1] Σ of operation symbols ($A_{\mathsf{op}}, B_{\mathsf{op}}$ are sets)

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

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

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

  where we think of $|\mathcal{R}|$ 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 Σ (or for an algebraic theory).

A natural model of top-level runtime

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

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

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

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

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

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

  $$\Big\{ \quad \mathsf{get} : \mathbb{1} \rightsquigarrow S \quad , \quad \mathsf{set} : S \rightsquigarrow \mathbb{1} \quad \Big\}$$

  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]
  - **top-level implementation of algebraic effects** in 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 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] $\mathsf{r} : \mathbf{Free}_{\Sigma}(-) \longrightarrow \mathbf{St}_{|\mathcal{R}|}$

---

[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.

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

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
    - ...

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**

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**

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

simply amounts to the **universal property of free models**, i.e.,

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

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 $\mathbf{St}_{|\mathcal{R}|}$, 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**

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

  simply amounts to the **universal property of free models**, i.e.,

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

- 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** **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)** $\mathsf{getenv} : \mathbb{1} \rightsquigarrow C \quad \& \quad \mathsf{setenv} : C \rightsquigarrow \mathbb{1}$
  - **(ii)** $\mathsf{op} : A_{\mathsf{op}} \rightsquigarrow B_{\mathsf{op}}$ (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)** $\mathsf{getenv} : \mathbb{1} \rightsquigarrow C \quad \& \quad \mathsf{setenv} : C \rightsquigarrow \mathbb{1}$
  - **(ii)** $\mathsf{op} : A_{\mathsf{op}} \rightsquigarrow B_{\mathsf{op}}$ (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

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

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

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

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

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

- 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** (the sum of **T** and excs.)

$$\mathbf{K}_C^{\Sigma' ! E_{\mathsf{op}} \lightning S}\, B_{\mathsf{op}} \quad \stackrel{\text{def}}{=} \quad C \Rightarrow \mathbf{Free}_{\Sigma'}\Big(\big((B_{\mathsf{op}} + E_{\mathsf{op}}) \times C\big) + S\Big)$$

T-runners as a programming construct

(towards a core calculus for runners)

T-runners as a **programming construct**

- First, we include **T-runners** for Σ

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

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

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

T-runners as a **programming construct**

- First, we include **T-runners** for Σ

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

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

$$\mathbf{let}\ \mathsf{R} = \mathbf{runner}\ \{\ \mathsf{op}_1\ \mathsf{x}_1 \to \mathsf{K}_1\ \ ,\ \ \ldots\ \ ,\ \ \mathsf{op}_n\ \mathsf{x}_n \to \mathsf{K}_n\ \ \}\ \mathbf{@}\ \mathsf{C}$$

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

```
let R_FH = runner {
  write s → if (length s > maxSize)
            then (raise WriteSizeExceeded)
            else  (let fh = getenv () in
                      if (isValid fh) then (fwrite (fh,s)) else (kill IOError))
} @ FileHandle
```

where

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

$$\left(\mathsf{fwrite} : \mathsf{FileHandle} \times \mathsf{String} \rightsquigarrow 1\ !\ E\right) \in \Sigma' \qquad S = \{\ \mathsf{IOError}\ \}$$

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 **T**-runner $\mathcal{R}$ are all **tail-recursive**

Controlled initialisation and finalisation

- Recall that the components r_X of the monad morphism

$$\xrightarrow{\text{initialisation}} \qquad \text{“}\circ\text{”} \qquad \mathsf{r} : \mathbf{Free}_{\Sigma}(-) \longrightarrow \mathbf{T} \qquad \text{“}\circ\text{”} \qquad \xrightarrow{\text{finalisation}}$$

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

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

using R **@** $\mathsf{M}_{\mathsf{init}}$
run M
finally $\{$**return** x **@** c $\to$ $\mathsf{M}_{\mathsf{ret}}$, ... **raise** e **@** c $\to$ M_{e} ... , ... **kill** s $\to$ M_{s} ...$\}$

  where

  (a **user monad**)

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

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

- Recall that the components r_X of the monad morphism

$$\xrightarrow{\text{initialisation}} \qquad \text{“}\circ\text{”} \qquad \mathsf{r} : \mathbf{Free}_{\Sigma}(-) \longrightarrow \mathbf{T} \qquad \text{“}\circ\text{”} \qquad \xrightarrow{\text{finalisation}}$$

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

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

using R **@** $\mathsf{M}_{\mathsf{init}}$
run M
finally $\{$**return** x **@** $\mathsf{c} \to \mathsf{M}_{\mathsf{ret}}$, ... **raise** e **@** $\mathsf{c} \to \mathsf{M}_{\mathsf{e}}$ ... , ... **kill** $\mathsf{s} \to \mathsf{M}_{\mathsf{s}}$ ...$\}$

  where

  (a **user monad**)

  - Ms are **user code**, modelled using $\mathbf{U}^{\Sigma ! E}\, X \overset{\text{def}}{=} \mathbf{Free}_{\Sigma}(X + E)$
  - $\mathsf{M}_{\mathsf{init}}$ produces the **initial kernel state**
  - M is the user code being **run using the runner** R
  - $\mathsf{M}_{\mathsf{ret}}$, 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

$$\xrightarrow{\text{initialisation}} \quad \text{“}\circ\text{”} \quad \mathsf{r} : \mathbf{Free}_{\Sigma}(-) \longrightarrow \mathbf{T} \quad \text{“}\circ\text{”} \quad \xrightarrow{\text{finalisation}}$$

  induced by a $\mathbf{T}$-runner $\mathcal{R}$ 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 → M_ret , ... raise e @ c → M_e ... , ... kill s → M_s ...}
```

  where

  (a **user monad**)

  - Ms are **user code**, modelled using $\mathbf{U}^{\Sigma ! E}\, X \overset{\text{def}}{=} \mathbf{Free}_{\Sigma}(X + E)$
  - $\mathsf{M}_{\mathsf{init}}$ produces the **initial kernel state**
  - M is the user code being **run using the runner** R
  - $\mathsf{M}_{\mathsf{ret}}$, M_{e}, M_{s} **finalise** for return values, exceptions, and signals
  - $\mathsf{M}_{\mathsf{ret}}$ 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**

```
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**

```
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** $\mathsf{write} : \mathsf{String} \rightsquigarrow 1 \; ! \; E \cup \{\mathsf{WriteSizeExceeded}\}$ 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)

Core calculus (syntax)

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

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

- **Types**

$$\begin{array}{rcl} X, Y & ::= & \mathsf{B} \mid 1 \mid 0 \mid X \times Y \mid X + Y \\ & \mid & X \xrightarrow{\Sigma} Y \mathrel{!} E \\ & \mid & X \xrightarrow{\Sigma} Y \mathrel{!} E \lightning S \mathrel{@} C \\ & \mid & \Sigma \Rightarrow \Sigma' \lightning S \mathrel{@} C \end{array}$$

- **Values**

$$\Gamma \vdash V : X$$

- **User computations**

$$\Gamma \models^{\Sigma} M : X \mathrel{!} E$$

- **Kernel computations**

$$\Gamma \models^{\Sigma} K : X \mathrel{!} E \lightning S \mathrel{@} C$$

Core calculus (user computations)

$$
\begin{array}{rcll}
M, N & ::= & \mathsf{return}\ V & \text{value} \\
 & \mid & \mathsf{try}\ M\ \mathsf{with}\ \{\mathsf{return}\ x \mapsto N, (\mathsf{raise}\ e \mapsto N_e)_{e \in E}\} & \text{exception handler} \\
 & \mid & V\ W & \text{application} \\
 & \mid & \mathsf{match}\ V\ \mathsf{with}\ \{\langle x, y \rangle \mapsto M\} & \text{product elimination} \\
 & \mid & \mathsf{match}\ V\ \mathsf{with}\ \{\}_X & \text{empty elimination} \\
 & \mid & \mathsf{match}\ V\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto M, \mathsf{inr}\ y \mapsto N\} & \text{sum elimination} \\
 & \mid & \mathsf{op}_X(V, (x \,.\, M), (N_e)_{e \in E_{\mathsf{op}}}) & \text{operation call} \\
 & \mid & \mathsf{raise}_X\ e & \text{raise exception} \\
 & \mid & \mathsf{using}\ V \mathbin{@} W\ \mathsf{run}\ M\ \mathsf{finally}\ \{ & \text{run} \\
 & & \quad \mathsf{return}\ x \mathbin{@} c \mapsto N, & \\
 & & \quad (\mathsf{raise}\ e \mathbin{@} c \mapsto N_e)_{e \in E}, & \\
 & & \quad (\mathsf{kill}\ s \mapsto N_s)_{s \in S}\} & \\
 & \mid & \mathsf{kernel}\ K \mathbin{@} V\ \mathsf{finally}\ \{ & \text{switch to kernel mode} \\
 & & \quad \mathsf{return}\ x \mathbin{@} c \mapsto N, & \\
 & & \quad (\mathsf{raise}\ e \mathbin{@} c \mapsto N_e)_{e \in E}, & \\
 & & \quad (\mathsf{kill}\ s \mapsto N_s)_{s \in S}\} &
\end{array}
$$

Core calculus (kernel computations)

$$
\begin{array}{rcll}
K, L & ::= & \mathsf{return}_C\, V & \text{value} \\
 & \mid & \mathsf{try}\ K\ \mathsf{with}\ \{\mathsf{return}\ x \mapsto L, (\mathsf{raise}\ e \mapsto L_e)_{e \in E}\} & \text{exception handler} \\
 & \mid & V\, W & \text{application} \\
 & \mid & \mathsf{match}\ V\ \mathsf{with}\ \{\langle x, y \rangle \mapsto K\} & \text{product elimination} \\
 & \mid & \mathsf{match}\ V\ \mathsf{with}\ \{\}_{X @ C} & \text{empty elimination} \\
 & \mid & \mathsf{match}\ V\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto K, \mathsf{inr}\ y \mapsto L\} & \text{sum elimination} \\
 & \mid & \mathsf{op}_{X @ C}(V, (x\,.\,K), (L_e)_{e \in E_{\mathsf{op}}}) & \text{operation call} \\
 & \mid & \mathsf{raise}_{X @ C}\, e & \text{raise exception} \\
 & \mid & \mathsf{kill}_{X @ C}\, s & \text{send signal} \\
 & \mid & \mathsf{getenv}_C(c\,.\,K) & \text{get state} \\
 & \mid & \mathsf{setenv}(V, K) & \text{set state} \\
 & \mid & \mathsf{user}\ M\ \mathsf{with}\ \{\mathsf{return}\ x \mapsto K, (\mathsf{raise}\ e \mapsto L_e)_{e \in E}\} & \text{switch to user mode}
\end{array}
$$

Core calculus (type system and eq. theory)

Core calculus (type system and eq. theory)

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

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

Core calculus (type system and eq. theory)

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

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

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

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

Core calculus (type system and eq. theory)

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

$$\frac{\Gamma \vdash V : A \qquad A <: B}{\Gamma \vdash V : B}$$

$$\frac{\Gamma \vDash^{\Sigma} M : A \,!\, E \qquad \Sigma \subseteq \Sigma' \qquad A <: B \qquad E \subseteq E'}{\Gamma \vDash^{\Sigma'} M : B \,!\, E'}$$

$$\frac{\begin{array}{c}\Gamma \vDash^{\Sigma} K : A \,!\, E \lightning S \mathbin{\texttt{@}} C \qquad \Sigma \subseteq \Sigma' \\ A <: B \qquad E \subseteq E' \qquad S \subseteq S' \qquad C = C'\end{array}}{\Gamma \vDash^{\Sigma'} K : B \,!\, E' \lightning S' \mathbin{\texttt{@}} C'}$$

Core calculus (type system and eq. theory)

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

$$\frac{\Gamma \vdash V : A \qquad A <: B}{\Gamma \vdash V : B}$$

$$\frac{\Gamma \models^{\Sigma} M : A \mathrel{!} E \qquad \Sigma \subseteq \Sigma' \qquad A <: B \qquad E \subseteq E'}{\Gamma \models^{\Sigma'} M : B \mathrel{!} E'}$$

$$\frac{\begin{array}{c}\Gamma \models^{\Sigma} K : A \mathrel{!} E \mathbin{\lightning} S \mathbin{@} C \qquad \Sigma \subseteq \Sigma' \\ A <: B \qquad E \subseteq E' \qquad S \subseteq S' \qquad C = C'\end{array}}{\Gamma \models^{\Sigma'} K : B \mathrel{!} E' \mathbin{\lightning} S' \mathbin{@} C'}$$

- 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 **Set**, using
  - **user monads** $\mathbf{U}^{\Sigma ! E}\, X \overset{\text{def}}{=} \mathbf{Free}_{\Sigma}(X + E)$
  - **kernel monads** $\mathbf{K}_{C}^{\Sigma ! E \lightning S}\, X \overset{\text{def}}{=} C \Rightarrow \mathbf{Free}_{\Sigma}\big(((X + E) \times C) + S\big)$

Core calculus (semantics)

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

- (At a high level) the **judgements are interpreted** as

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

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

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

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

$$\begin{tikzcd}
\llbracket \Gamma \rrbracket \arrow[r, "\llbracket \Gamma \vDash^{\Sigma} K \,:\, X \,!\, E \,\lightning\, S \,@\, C \rrbracket"] \arrow[d, "\subseteq"'] & \mathbf{K}^{\Sigma ! E \lightning S}_{\llbracket C \rrbracket} \llbracket X \rrbracket \arrow[d, "\subseteq"] \\
{[\![\![} \Gamma^{s} {]\!]\!]} \arrow[r, "{[\![\![} \Gamma^{s} \vdash K \,:\, X^{s} \,@\, C {]\!]\!]}"'] & \mathbf{K}^{\mathcal{O} ! \mathcal{E} \lightning \mathcal{S} + \{\dagger\}}_{[\![\![C]\!]\!]} {[\![\![} X^{s} {]\!]\!]}
\end{tikzcd}$$

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

$$\begin{tikzcd}
\llbracket \Gamma \rrbracket \arrow[rr, "\llbracket \Gamma \models^{\Sigma} K \,:\, X \,!\, E \lightning S \,@\, C \rrbracket"] \arrow[d, "\subseteq"'] & & \mathbf{K}^{\Sigma ! E \lightning S}_{\llbracket C \rrbracket} \llbracket X \rrbracket \arrow[d, "\subseteq"] \\
[\![\![\Gamma^{s}]\!]\!] \arrow[rr, "{[\![\![\Gamma^{s} \vdash K \,:\, X^{s} \,@\, C]\!]\!]}"'] & & \mathbf{K}^{\mathcal{O} ! \mathcal{E} \lightning \mathcal{S} + \{\pm\}}_{[\![\![C]\!]\!]} [\![\![X^{s}]\!]\!]
\end{tikzcd}$$

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

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

Core calculus (semantics ctd.)

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

- For instance, **kernel computations** are interpreted as

$$\begin{tikzcd}
\llbracket \Gamma \rrbracket \arrow[r, "\llbracket \Gamma \vdash^{\Sigma} K \,:\, X \,!\, E \lightning S \,@\, C \rrbracket"] \arrow[d, "\subseteq"'] & \mathbf{K}^{\Sigma ! E \lightning S}_{\llbracket C \rrbracket} \llbracket X \rrbracket \arrow[d, "\subseteq"] \\
[\![\![\Gamma^s]\!]\!] \arrow[r, "{[\![\![\Gamma^s \vdash K \,:\, X^s \,@\, C]\!]\!]}"'] & \mathbf{K}^{\mathcal{O} ! \mathcal{E} \lightning \mathcal{S} + \{\dagger\}}_{[\![\![C]\!]\!]} [\![\![X^s]\!]\!]
\end{tikzcd}$$

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

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

- **Ground type restriction** on C needed to stay within **Sub**$(\ldots)$
  - Otherwise, analogously to subtyping, we'd need **lenses** instead

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

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 = 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

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

Vertical nesting for **active monitoring**

Vertical nesting for **active monitoring**

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

```
type IntHeap = (Nat → (Int + 1)) × Nat                          type Ref = Nat
```

```
let R_IntState = runner {
  alloc x → let h = getenv () in                      (∗ alloc : Int ⇝ Ref ! {} ∗)
            let (r,h') = heapAlloc h x in
            setenv h';
            return r ,

  deref r → let h = getenv () in                      (∗ deref : Ref ⇝ Int ! {} ∗)
            match (heapSel h r) with
            | inl x → return x
            | inr () → kill ReferenceDoesNotExist ,

  assign r y → let h = getenv () in             (∗ assign : Ref × Int ⇝ 1 ! {} ∗)
               match (heapUpd h r y) with
               | inl h' → setenv h'
               | inr () → kill ReferenceDoesNotExist
} @ IntHeap
```

Vertical nesting for **active monitoring** ctd.

- Next we define a runner for **monotonicity layer** on top of $\mathsf{R}_{\mathsf{IntState}}$

Vertical nesting for **active monitoring** ctd.

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

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

```
let R_MonState = runner {
  mAlloc x rel → let r = alloc x in                        (∗ : Int × Ord ⇝ Ref ! {} ∗)
                  let m = getenv () in
                  setenv (memAdd m r rel);
                  return r,

  mDeref r → deref r ,                                   (∗ monDeref : Ref ⇝ Int ! {} ∗)

  mAssign r y → let x = deref r in                        (∗ : Ref × Int ⇝ 1 ! {MV} ∗)
                 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

```
using R_IntState @ ((fun _ → inr ()) , 0)        (∗ init. empty ML−style heap ∗)
run (

  using R_MonState @ (fun _ → inr ())        (∗ init. empty preorders memory ∗)
  run (

    let r = mAlloc 0 (⩽) in
    mAssign r 1;
    mAssign r 0;        (∗ R_MonState raises MonotonicityViolation exception ∗)
    mAssign r 2

  )
  finally { ... , raise MonotonicityViolation @ m → ... , ... }

)
finally { ... }
```

Runners can also be horizontally paired

Runners can also be **horizontally paired**

- Given runners for Σ and Σ'

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

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

```
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'')),       (∗ e ∈ E_op1i ∗)
                       kill s → return (inr s) }                       (∗ s ∈ S1 ∗)
             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 → ...,          (∗ analogously to above, just on 2nd comp. of state ∗)
  ... } @ C1 × C2
```

Runners can also be **horizontally paired**

- Given runners for Σ and Σ'

```
let R_1 = runner { ... ,  op_1i x → K_1i  ,  ... } @ C_1
let R_2 = runner { ... ,  op_2j x → K_2j  ,  ... } @ C_2
```

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

```
let R = runner { ... ,
  op_1i x → let (c,c') = getenv () in
            user (kernel (K_1i x) @ c finally {
                     return y @ c'' → return (inl (inl y,c'')),
                     raise e @ c'' → return (inl (inr e,c'')),     (* e ∈ E_op1i *)
                     kill s → return (inr s) }                     (* s ∈ S_1 *)
            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 },
  ... ,
  op_2j x → ...,         (* analogously to above, just on 2nd comp. of state *)
  ... } @ C_1 × C_2
```

- For instance, this way we can build a runner for **IO and state**

Other examples (in Haskell)

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 sig AmbHeap 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] sig AmbHeap
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.)

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

- $\llbracket V \rrbracket_\gamma \; = \; \mathcal{R} \; = \; \left(\overline{\mathsf{op}}_{\mathcal{R}} : \llbracket A_{\mathsf{op}} \rrbracket \longrightarrow \mathbf{K}^{\Sigma' ! E_{\mathsf{op}} \lightning S}_{\llbracket C \rrbracket} \llbracket B_{\mathsf{op}} \rrbracket \right)_{\mathsf{op} \in \Sigma}$
- $\llbracket W \rrbracket_\gamma \; \in \; \llbracket C \rrbracket$
- $\llbracket M \rrbracket_\gamma \; \in \; \mathbf{U}^{\Sigma ! E} \llbracket A \rrbracket$
- $\llbracket \textbf{return } \mathsf{x} \mathbin{@} \mathsf{c} \to N_{ret} \rrbracket_\gamma \; \in \; \llbracket A \rrbracket \times \llbracket C \rrbracket \longrightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$
- $\llbracket (\textbf{raise } \mathsf{e} \mathbin{@} \mathsf{c} \to N_e)_{e \in E} \rrbracket_\gamma \; \in \; E \times \llbracket C \rrbracket \longrightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$
- $\llbracket (\textbf{kill } \mathsf{s} \to N_s)_{s \in S} \rrbracket_\gamma \; \in \; S \longrightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$
- allowing us to use the **free model property** to get

$$\mathbf{U}^{\Sigma ! E} \llbracket A \rrbracket \xrightarrow{\mathsf{r}_{\llbracket A \rrbracket + E}} \mathbf{K}^{\Sigma' ! E \lightning S}_{\llbracket C \rrbracket} \llbracket A \rrbracket \xrightarrow{(\lambda \llbracket N_{ret} \rrbracket_\gamma)^{\ddagger}} \llbracket C \rrbracket \Rightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$$

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