Interacting with the external world using comodels (aka runners)

Danel Ahman

(joint work with Andrej Bauer)

University of Ljubljana, Slovenia

Gallinette seminar, Nantes, 14.10.2019

The plan

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

- **Runners** – a natural model for **top-level runtime**

- **T-runners** – for also modelling **non-top-level runtimes**

- Turning **T**-runners into a **useful programming construct**

- Some **programming examples**

- Some **implementation details**

Computational effects

and

external resources

Computational effects in PL

Computational effects in PL

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

```
type St a = String → (a,String)

f :: St a → St (a,a)
f c = c  >>=  (\x → c  >>=  (\y → return (x,y)))
```

Computational effects in PL

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

```
type St a = String → (a,String)

f :: St a → St (a,a)
f c = c  >>=  (\x → c  >>=  (\y → return (x,y)))
```

- Using **alg. effects** and **handlers** (as in EFF, FRANK, KOKA)

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

let g (c:Unit → a!{Get,Put}) =
  with st_handler handle (perform (Put 42); c ())  (∗ : int → a ∗ int ∗)
```

Computational effects in PL

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

```
type St a = String → (a,String)

f :: St a → St (a,a)
f c = c  >>=  (\x → c  >>=  (\y → return (x,y)))
```

- Using **alg. effects** and **handlers** (as in EFF, FRANK, KOKA)

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

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

- Both are good for **faking comp. effects** in a pure language!

  But what about effects that need access to the **external world**?

External resources in PL

External resources **in PL**

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

```
(∗ System.IO ∗)
type IO a
openFile :: FilePath → IOMode → IO Handle
```

```
(∗ pervasives.eff ∗)
effect RandomInt : int → int
effect RandomFloat : float → float
```

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

```
(∗ eff/src/backends/eval.ml ∗)
let rec top_handle op =
  match op with
  | ...
```

External resources **in PL**

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

```
(* System.IO *)
type IO a
openFile :: FilePath → IOMode → IO Handle
```

```
(* pervasives.eff *)
effect RandomInt : int → int
effect RandomFloat : float → float
```

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

```
(* eff/src/backends/eval.ml *)
let rec top_handle op =
  match op with
  | ...
```

but there are some issues with that approach . . .

First issue

First issue

- Difficult to cover all possible use cases
  - **external resources hard-coded** into the top-level runtime
  - **non-trivial to change** what's available and how it's implemented

First issue

- Difficult to cover all possible use cases
    - **external resources hard-coded** into the top-level runtime
    - **non-trivial to change** what's available and how it's implemented

Ohad 8:35 PM
So here's the hack I added. We should do something a bit more principled

In `pervasives.eff`:

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

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

```
    | "Write" ->
        (match v with
         | V.Tuple vs ->
            let (file_name :: str :: _) = List.map V.to_str vs in
            let file_handle = open_out_gen
                                [Open_wronly
                                ;Open_append
                                ;Open_creat
                                ;Open_text
                                ] 0o666 file_name in
        Printf.fprintf file_handle "%s" str;
        close_out file_handle;
        top_handle (k V.unit_value)
        )
```

First issue

- Difficult to cover all possible use cases
    - **external resources hard-coded** into the top-level runtime
    - **non-trivial to change** what's available and how it's implemented

Ohad 8:35 PM
So here's the hack I added. We should do something a bit more principled

In `pervasives.eff`:

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

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

```
    | "Write" ->
        (match v with
        | V.Tuple vs ->
            let (file_name :: str :: _) = List.map V.to_str vs in
            let file_handle = open_out_gen
                            [Open_wronly
                            ;Open_append
                            ;Open_creat
                            ;Open_text
                            ] 0o666 file_name in
        Printf.fprintf file_handle "%s" str;
        close_out file_handle;
        top_handle (k V.unit_value)
        )
```

This talk — a principled modular (co)algebraic approach!

Second issue

Second issue

- **Lack of linearity** for external resources

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

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

Second issue

- **Lack of linearity** for external resources

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

let g s =
  let fh = f s in fread fh          (* fh not open ! *)
```

Second issue

- **Lack of linearity** for external resources

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

let g s =
  let fh = f s in fread fh              (* fh not open ! *)
```

- We shall address these kinds of issues **indirectly**,

  - by **not** introducing a linear typing discipline
  - but instead make it convenient to **hide external resources**
    (and address stronger typing disciplines in the future)

Third issue

Third issue

- **Excessive generality** of effect handlers

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

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

let f' s = handle (f "bar") with h
```

Third issue

- **Excessive generality** of effect handlers

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


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


let f' s = handle (f "bar") with h
```

where misuse of external resources can also be **purely accidental**

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


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

Third issue

- **Excessive generality** of effect handlers

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

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

let f' s = handle (f "bar") with h
```

- We shall address these kinds of issues **directly**,

  - by proposing a **restricted form of handlers** for resources

  - that support **controlled initialisation** and **finalisation**,

  - and **limit** how general handlers can be used

Runners enter the spotlight

A natural model of top-level runtime

A natural model of top-level runtime

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

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

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

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

---

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

A natural model of top-level runtime

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

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

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

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

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

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

  is given by

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

---

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

A natural model of top-level runtime ctd.

- Runners/comodels have been used for
  - **operational semantics** using tensors of models and comodels

    [Plotkin and Power '08]

    and
  - **stateful running** of algebraic effects   [Uustalu '15]
  - **linear-use state-passing translation**

    [Møgelberg and Staton '11, '14]

A natural model of top-level runtime ctd.

- Runners/comodels have been used for

    - **operational semantics** using tensors of models and comodels

        [Plotkin and Power '08]

        and

    - **stateful running** of algebraic effects [Uustalu '15]

    - **linear-use state-passing translation**

        [Møgelberg and Staton '11, '14]

- The latter explicitly rely on one-to-one correspondence between

    - **runners** $\mathcal{R}$

    - **monad morphisms**[3] $r : \mathbf{Free}_\Sigma(-) \longrightarrow \mathbf{St}_{|\mathcal{R}|}$

    where

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

---

[3]$\mathbf{Free}_\Sigma(X)$ is the free monad ind. defined with leaves val x and nodes op(a, κ).

A natural model of top-level runtime ctd.

- For our purposes, we see runners

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

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

A natural model of top-level runtime ctd.

- For our purposes, we see runners

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

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

- But what if this runtime is not **the** runtime?
  - hardware vs OS
  - OS vs VMs
  - VMs vs sandboxes

A natural model of top-level runtime ctd.

- For our purposes, we see runners

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

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

- But what if this runtime is not **the** runtime?
    - hardware vs OS
    - OS vs VMs
    - VMs vs sandboxes

- Unfortunately, runners, as defined above, are **not readily able to**
    - use **external resources**
    - **signal failure** caused by unavoidable circumstances

A natural model of top-level runtime ctd.

- For our purposes, we see runners

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

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

- But what if this runtime is not **the** runtime?
  - hardware vs OS
  - OS vs VMs
  - VMs vs sandboxes

- Unfortunately, runners, as defined above, are **not readily able to**
  - use **external resources**
  - **signal failure** caused by unavoidable circumstances

- But is there a **useful generalisation** that would achieve this?

Effectful runners for modular top-levels

Effectful runners for modular top-levels

- Møgelberg and Staton usefully observed that a **runner** $\mathcal{R}$
  is equivalently simply a family of **generic effects** for $\mathbf{St}_{|\mathcal{R}|}$, i.e.,

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

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.,

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

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

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

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.,

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

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

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

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

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

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

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

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**

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

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

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

- 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 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 T** to use?

- We want a runner to be a bit like a **kernel** of an OS, i.e., to

  (i) provide management of **(internal) resources**

  (ii) use further **external resources**

  (iii) **signal failure** caused by unavoidable circumstances

- **Algebraically** (and pragmatically), this amounts to taking

  (i) getenv : $\mathbb{1} \rightsquigarrow C$, setenv : $C \rightsquigarrow \mathbb{1}$

  (ii) op : $A_{\text{op}} \rightsquigarrow B_{\text{op}}$ (op $\in \Sigma'$, for some external Σ')

  (iii) kill : $S \rightsquigarrow \mathbb{0}$

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

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

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

- The **induced monad** is then isomorphic to

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

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

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

- Observe that raising signals in S **discards the state**,

  but **not all problems are terminal**—they can be recovered from

Effectful runners for modular top-levels ctd.

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

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

- 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, Σ' with operation symbols

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

Effectful runners for modular top-levels ctd.

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

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

- 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, Σ' with operation symbols

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

Effectful runners for modular top-levels ctd.

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

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

- 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, Σ' with operation symbols

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

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

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

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

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

T-runners as a programming construct

T-runners as a programming construct

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

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

  we accommodate **runners as values** and **co-ops. as kernel code**

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

T-runners as a programming construct

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

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

  we accommodate **runners as values** and **co-ops. as kernel code**

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

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

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

  where

  $$(\mathsf{fwrite} : \mathsf{FileHandle} \times \mathsf{String} \rightsquigarrow 1 \, ! \, E) \in \Sigma'$$

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

Controlled initialisation and finalisation

Controlled initialisation and finalisation

- Recall that the components r_X of the monad morphism

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

  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

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

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

- We make use of it to enable one 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 ...}
  ```

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

- Recall that the components r_X of the monad morphism

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

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

- We make use of it to enable one 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 $\mathbf{U}^{\Sigma!E} X \stackrel{\text{def}}{=} \mathbf{Free}_\Sigma(X + E)$

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

- Recall that the components r_X of the monad morphism

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

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

- We make use of it to enable one 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 $\mathbf{U}^{\Sigma ! E} X \stackrel{\text{def}}{=} \mathbf{Free}_\Sigma(X + E)$

  - M_{init} produces the **initial kernel state**

  - M is the user code being **run using the runner** R

  - M_{ret}, M_e, M_s **finalise** for return values, exceptions, and signals

Controlled initialisation and finalisation

- Recall that the components r_X of the monad morphism

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

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

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

Controlled initialisation and finalisation ctd.

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

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

- Importantly, here

  - the **file handle is hidden** from M

  - M can only use write but not fopen and fclose

    - write : String $\rightsquigarrow$ 1 ! $E \cup \{$WriteSizeExceeded$\}$

  - fopen and fclose are limited to **initialisation-finalisation**

A core calculus for programming with runners

Core calculus (syntax)

Core calculus (syntax)

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

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

- **Types**

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

- **Values**

$$\Gamma \vdash V : X$$

- **User computations**

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

- **Kernel computations**

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

Core calculus (user computations)

$$
\begin{aligned}
M, N ::= \ & \mathsf{return}\ V \\
| \ & \mathsf{try}\ M\ \mathsf{with}\ \{\mathsf{return}\ x \mapsto M_{\mathsf{return}}, \\
& \qquad\qquad (\mathsf{raise}\ e \mapsto N_e)_{e \in E}\} \\
| \ & V\ W \\
| \ & \mathsf{match}\ V\ \mathsf{with}\ \{\langle x, y \rangle \mapsto M\} \\
| \ & \mathsf{match}\ V\ \mathsf{with}\ \{\}_X \\
| \ & \mathsf{match}\ V\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto M, \mathsf{inr}\ y \mapsto N\} \\
| \ & \mathsf{op}_X(V, (x\,.\,M), (N_e)_{e \in E}) \\
| \ & \mathsf{raise}_X\ e \\
| \ & \mathsf{using}\ V\ @\ W\ \mathsf{run}\ M\ \mathsf{finally}\ \{\mathsf{return}\ x\ @\ c \mapsto M, \\
& \qquad\qquad\qquad\qquad\qquad (\mathsf{raise}\ e\ @\ c \mapsto N_e)_{e \in E} \\
& \qquad\qquad\qquad\qquad\qquad (\mathsf{kill}\ s \mapsto N_s)_{s \in S}\} \\
| \ & \mathsf{kernel}\ K\ @\ V\ \mathsf{finally}\ \{\mathsf{return}\ x\ @\ c \mapsto M, \\
& \qquad\qquad\qquad\qquad\qquad (\mathsf{raise}\ e\ @\ c \mapsto N_e)_{e \in E} \\
& \qquad\qquad\qquad\qquad\qquad (\mathsf{kill}\ s \mapsto N_s)_{s \in S}\}
\end{aligned}
$$

Core calculus (kernel computations)

$$
\begin{aligned}
K, L ::=\ & \mathsf{return}_C\, V \\
& \mid\ \mathsf{try}\ K\ \mathsf{with}\ \{\mathsf{return}\ x \mapsto K_{\mathsf{return}}, \\
& \qquad\qquad\qquad (\mathsf{raise}\ e \mapsto L_e)_{e \in E}\} \\
& \mid\ V\,W \\
& \mid\ \mathsf{match}\ V\ \mathsf{with}\ \{\langle x, y\rangle \mapsto K\} \\
& \mid\ \mathsf{match}\ V\ \mathsf{with}\ \{\}_{X @ C} \\
& \mid\ \mathsf{match}\ V\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto K, \mathsf{inr}\ y \mapsto L\} \\
& \mid\ \mathsf{op}_{X @ C}(V, (x\,.\,K), (L_e)_{e \in E}) \\
& \mid\ \mathsf{raise}_{X @ C}\, e \\
& \mid\ \mathsf{kill}_{X @ C}\, s \\
& \mid\ \mathsf{getenv}_C(c\,.\,K) \\
& \mid\ \mathsf{setenv}(V, K) \\
& \mid\ \mathsf{user}\ M\ \mathsf{with}\ \{\mathsf{return}\ x \mapsto K_{\mathsf{return}}, \\
& \qquad\qquad\qquad (\mathsf{raise}\ e \mapsto L_e)_{e \in E}\}
\end{aligned}
$$

Core calculus (type system and eq. theory)

Core calculus (type system and eq. theory)

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

$$\frac{\begin{array}{cc} \Gamma \vdash V : \Sigma \Rightarrow \Sigma' \not\downarrow S @ C & \Gamma \vdash W : C \\ \Gamma \vDash^\Sigma M : X \: ! \: E & \Gamma, x\!:\!X, c\!:\!C \vDash^{\Sigma'} N_{ret} : Y \: ! \: E' \\ \left(\Gamma, c\!:\!C \vDash^{\Sigma'} N_e : Y \: ! \: E'\right)_{e \in E} & \left(\Gamma \vDash^{\Sigma'} N_s : Y \: ! \: E'\right)_{s \in S} \end{array}}{\begin{array}{l} \Gamma \vDash^{\Sigma'} \textbf{using } V @ W \textbf{ run } M \textbf{ finally } \{ \textbf{ return } x @ c \mapsto N_{ret} \: , \\ \qquad\qquad\qquad\qquad\qquad \left(\textbf{raise } e @ c \mapsto N_e\right)_{e \in E} \: , \\ \qquad\qquad\qquad\qquad\qquad \left(\textbf{kill } s \mapsto N_s\right)_{s \in S} \: \} : Y \: ! \: E' \end{array}}$$

Core calculus (type system and eq. theory)

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

$$\Gamma \vdash V : \Sigma \Rightarrow \Sigma' \not\lightning S @ C \qquad \Gamma \vdash W : C$$

$$\Gamma \models^{\Sigma} M : X \mathbin{!} E \qquad \Gamma, x : X, c : C \models^{\Sigma'} N_{ret} : Y \mathbin{!} E'$$

$$\frac{\left(\Gamma, c : C \models^{\Sigma'} N_e : Y \mathbin{!} E'\right)_{e \in E} \qquad \left(\Gamma \models^{\Sigma'} N_s : Y \mathbin{!} E'\right)_{s \in S}}{\Gamma \models^{\Sigma'} \textbf{using } V @ W \textbf{ run } M \textbf{ finally } \{ \textbf{ return } x @ c \mapsto N_{ret} \, ,}$$

$$\left(\textbf{raise } e @ c \mapsto N_e\right)_{e \in E} \, ,$$

$$\left(\textbf{kill } s \mapsto N_s\right)_{s \in S} \, \} : Y \mathbin{!} E'$$

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

$$\Gamma \models^{\Sigma'} \textbf{using } R_C @ W \textbf{ run } \left(\text{op}_X \left(V, (x.M), (M_e)_{e \in E_{\text{op}}}\right)\right) \textbf{ finally } F$$

$$\equiv \textbf{kernel } R_{op}[V] @ W \textbf{ finally } \{$$

$$\textbf{return } x @ c' \mapsto \textbf{using } R_C @ c' \textbf{ run } M \textbf{ finally } F \, ,$$

$$\left(\textbf{raise } e @ c' \mapsto \textbf{using } R_C @ c' \textbf{ run } M_e \textbf{ finally } F\right)_{e \in E_{\text{op}}} \, ,$$

$$\left(\textbf{kill } s \mapsto N_s\right)_{s \in S} \, \} : Y \mathbin{!} E'$$

Core calculus (type system and eq. theory)

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

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

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

$$\frac{\begin{array}{ccc} \Gamma \overset{\Sigma}{\models} K : A \mathbin{!} E \mathbin{\natural} S \mathbin{@} C & \Sigma \subseteq \Sigma' \\ A <: B \qquad E \subseteq E' \qquad S \subseteq S' \qquad C = C' \end{array}}{\Gamma \overset{\Sigma'}{\models} K : B \mathbin{!} E' \mathbin{\natural} S' \mathbin{@} C'}$$

Core calculus (type system and eq. theory)

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

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

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

$$\frac{\Gamma \stackrel{\Sigma}{\vDash} K : A \mathbin{!} E \mathbin{\natural} S @ C \qquad \Sigma \subseteq \Sigma'}{A <: B \qquad E \subseteq E' \qquad S \subseteq S' \qquad C = C'}$$
$$\frac{}{\Gamma \stackrel{\Sigma'}{\vDash} K : B \mathbin{!} E' \mathbin{\natural} S' @ C'}$$

- 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 simplicity in **Set**, using

    - **user monads** $U^{\Sigma!E} X \overset{\text{def}}{=} \textbf{Free}_{\Sigma}(X + E)$

    - **kernel monads** $K_C^{\Sigma!E \natural S} X \overset{\text{def}}{=} C \Rightarrow \textbf{Free}_{\Sigma}\big(((X + E) \times C) + S\big)$

Core calculus (semantics)

- **Monadic semantics**, for simplicity in **Set**, using

    - **user monads** $\mathbf{U}^{\Sigma!E} X \overset{\text{def}}{=} \mathbf{Free}_\Sigma(X + E)$

    - **kernel monads** $\mathbf{K}_C^{\Sigma!E \natural S} X \overset{\text{def}}{=} C \Rightarrow \mathbf{Free}_\Sigma\big(((X + E) \times C) + S\big)$

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

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

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

$$\llbracket \Gamma \overset{\Sigma}{\vdash} K : X \mathbin{!} E \natural S @ C \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \mathbf{K}_{\llbracket C \rrbracket}^{\Sigma!E \natural S} \llbracket X \rrbracket$$

Core calculus (semantics ctd.)

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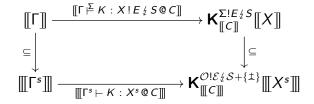
Core calculus (semantics ctd.)

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

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

$$
\begin{array}{ccc}
[\![\Gamma]\!] & \xrightarrow{\;[\![\Gamma \models^{\Sigma} K \,:\, X\,!\,E\,\natural\,S\,@\,C]\!]\;} & \mathsf{K}^{\Sigma!E\natural S}_{[\![C]\!]}[\![X]\!] \\
\scriptstyle{\subseteq}\Big\downarrow & & \Big\downarrow\scriptstyle{\subseteq} \\
[\![\![\Gamma^s]\!]\!] & \xrightarrow[\;[\![\![\Gamma^s \vdash K \,:\, X^s\,@\,C]\!]\!]\;]{} & \mathsf{K}^{\mathcal{O}!\mathcal{E}\natural\mathcal{S}+\{\bot\}}_{[\![\![C]\!]\!]}[\![\![X^s]\!]\!]
\end{array}
$$

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

Core calculus (semantics ctd.)

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

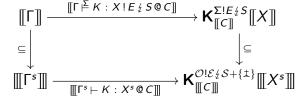- For instance, **kernel computations** are interpreted as

$$
\begin{array}{ccc}
[\![\Gamma]\!] & \xrightarrow{[\![\Gamma \overset{\Sigma}{\vdash} K \,:\, X\,!\,E \,\natural\, S \,@\, C]\!]} & \mathsf{K}^{\Sigma!E\natural S}_{[\![C]\!]}[\![X]\!] \\
\subseteq \Big\downarrow & & \Big\downarrow \subseteq \\
[\![\![\Gamma^s]\!]\!] & \xrightarrow[{[\![\![\Gamma^s \vdash K \,:\, X^s \,@\, C]\!]\!]}]{} & \mathsf{K}^{\mathcal{O}!\mathcal{E}\natural\mathcal{S}+\{\downarrow\}}_{[\![\![C]\!]\!]}[\![\![X^s]\!]\!]
\end{array}
$$

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

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

Core calculus (semantics ctd.)

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

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

$$
\begin{array}{ccc}
[\![\Gamma]\!] & \xrightarrow{[\![\Gamma \mid^{\Sigma}_{E} K \,:\, X \,!\, E \,\natural\, S \,@\, C]\!]} & \mathsf{K}^{\Sigma!E\natural S}_{[\![C]\!]}[\![X]\!] \\
{\scriptstyle \subseteq} \downarrow & & \downarrow {\scriptstyle \subseteq} \\
[\![\![\Gamma^s]\!]\!] & \xrightarrow[{[\![\![\Gamma^s \vdash K \,:\, X^s \,@\, C]\!]\!]}]{} & \mathsf{K}^{\mathcal{O}!\mathcal{E}\natural\mathcal{S}+\{\mathcal{I}\}}_{[\![\![C]\!]\!]}[\![\![X^s]\!]\!]
\end{array}
$$

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

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

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

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   e **@** str → **raise** e }
)
  finally {
    return x **@** fh → fclose fh; **return** x ,
    raise   e **@** fh → fclose fh; **raise** e , **kill** IOError → **return** ()}

where the **file contents runner** (with $\Sigma' = \mathbb{0}$) 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 }

  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 $R_{Sniffer}$ implements the same sig. Σ 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

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

```
let R_IntState = runner {
  alloc x → let h = getenv () in                    (∗ alloc : Int ⤳ Ref ! 𝟘 ∗)
            let (r,h') = heapAlloc h x in
            setenv h';
            return r ,

  deref r → let h = getenv () in                    (∗ deref : Ref ⤳ Int ! 𝟘 ∗)
            match (heapSel h r) with
            | inl x → return x
            | inr () → kill ReferenceDoesNotExist ,

  assign r y → let h = getenv () in          (∗ assign : Ref × Int ⤳ 1 ! 𝟘 ∗)
               match (heapUpd h r y) with
               | inl h' → setenv h'
               | inr () → kill ReferenceDoesNotExist
} @ IntHeap
```

- Next we define a runner for **monotonicity layer** on top of R_{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 {
  monAlloc x rel → let r = alloc x in          (∗ : Int × Ord ⤳ Ref ! 0 ∗)
                   let m = getenv () in
                   setenv (memAdd m r rel);
                   return r,

  monDeref r → deref r ,                       (∗ monDeref : Ref ⤳ Int ! 0 ∗)

  monAssign 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
} @ IntHeap
```

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 = monAlloc 0 (≤) in
    monAssign r 1;
    try (monAssign r 0) with {                      (* R_MonState raises exception *)
      return _ → monAssign r 3 ,
      raise MonotonicityViolation → return ()};
    monAssign r 2
  )
  finally {return x @ m → return x ,
           raise MonotonicityViolation @ m → ... ,
           kill PreorderDoesNotExist → ... }
)
finally {return x @ h → print h; return x ,
         kill ReferenceDoesNotExist → ... }
```

Runners can also be horizontally paired

Runners can also be horizontally paired

- Given a runner for Σ

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

  and a runner for Σ'

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

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

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

Runners can also be **horizontally paired**

- Given a runner for Σ

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

  and a runner for Σ'

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

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

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

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

Other examples

Other examples

- More general forms of **(ML-style) state** (for general `Ref A`)
  - if the host language allows it, we use GADTs, etc for safety
  - some examples extract a footprint from a larger memory

- **Combinations** of different effects and runners
  - in particular the combination of IO and state
  - good use case for both vertical and horizontal composition

- KOKA-style **ambient values** and **ambient functions**
  - ambient values are essentially **mutable variables/parameters**
  - ambient functions are **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

Implementing runners

Experimenting with the theory in practice

Experimenting with the theory in practice

- A **small experimental language** Coop^4
  - Implements the core calculus with few extras
  - The interpreter is directly based on the denotational semantics
  - Top-level containers for running external (OCaml) code

---

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

Experimenting with the **theory in practice**

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

  - Implements the core calculus with few extras
  - The interpreter is directly based on the denotational semantics
  - Top-level containers for running external (OCaml) code

- A HASKELL **library** HASKELL-COOP

  - A shallow-embedding of the core calculus in HASKELL
  - Uses one of the Freer monad implementations underneath
  - Again, the operational aspects implement the denot. semantics
  - Top-level containers for arbitrary HASKELL monads
  - Examples make use of HASKELL's features (GADTs, ...)

---

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

Experimenting with the theory in practice

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

  - Implements the core calculus with few extras
  - The interpreter is directly based on the denotational semantics
  - Top-level containers for running external (OCaml) code

- A HASKELL **library** HASKELL-COOP

  - A shallow-embedding of the core calculus in HASKELL
  - Uses one of the Freer monad implementations underneath
  - Again, the operational aspects implement the denot. semantics
  - Top-level containers for arbitrary HASKELL monads
  - Examples make use of HASKELL's features (GADTs, ...)

- Both still need some finishing touches, but will be public soon

---

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

Experimenting with the theory in practice

```haskell
module AmbientsTests where

import Control.Runner
import Control.Runner.Ambients

ambFun :: AmbVal Int -> Int -> AmbEff Int
ambFun x y =
  do x <- getVal x;
     return (x + y)

test1 :: AmbEff Int
test1 =
  withAmbVal
    (4 :: Int)
    (\ x ->
      withAmbFun
        (ambFun x)
        (\ f ->
          do rebindVal x 2;
             applyFun f 1))

test2 = ambTopLevel test1
```

Wrapping up

- **Runners** are a natural model of **top-level runtime**

- We propose **T-runners** to also model **non-top-level runtimes**

- We have turned **T**-runners into a **(practical ?) programming construct**, that supports controlled initialisation and finalisation

- I showed you some **combinators** and **programming examples**

- Two **implementations** in the works, COOP & HASKELL-COOP

- **Future:** lenses in subtyping and semantics, category of runners, handlers, larger case studies, refinement typing, compilation, . . .

Thank you!

Core calculus (semantics ctd.)

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

- $\llbracket V \rrbracket_\gamma \; = \; \mathcal{R} \; = \; \left(\overline{\text{op}}_{\mathcal{R}} : \llbracket A_{\text{op}} \rrbracket \longrightarrow \mathbf{K}^{\Sigma' ! E_{\text{op}} \, \natural \, S}_{\llbracket C \rrbracket} \llbracket B_{\text{op}} \rrbracket \right)_{\text{op} \in \Sigma}$

- $\llbracket W \rrbracket_\gamma \; \in \; \llbracket C \rrbracket$

- $\llbracket M \rrbracket_\gamma \; \in \; \mathbf{U}^{\Sigma ! E} \llbracket A \rrbracket$

- $\llbracket \textbf{return } x \text{ @ } c \to N_{ret} \rrbracket_\gamma \; \in \; \llbracket A \rrbracket \times \llbracket C \rrbracket \longrightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$

- $\llbracket (\textbf{raise } e \text{ @ } c \to N_e)_{e \in E} \rrbracket_\gamma \; \in \; E \times \llbracket C \rrbracket \longrightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$

- $\llbracket (\textbf{kill } s \to N_s)_{s \in S} \rrbracket_\gamma \; \in \; S \longrightarrow \mathbf{U}^{\Sigma' ! E'} \llbracket B \rrbracket$

- allowing us to use the **free model property** to get

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

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