

Runners in action

Danel Ahman

(joint work with Andrej Bauer)

University of Ljubljana, Slovenia

Tallinn, 18.11.2019

Comodels in action

Danel Ahman

(joint work with Andrej Bauer)

University of Ljubljana, Slovenia

Tallinn, 18.11.2019

Today's plan

- **Computational effects** and **external resources** in PL
- **Issues with standard approaches** to **external resources**
- **Comodels/Runners** – a natural model for **top-level runtime**
- **T-runners** – for also modelling **non-top-level runtimes**
- Turn **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 ())
```

- Good for **simulating 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)  
  )
```

This talk — 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
 - 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 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  
  
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 () }
```

- We shall address these kinds of issues **directly (!!)**,
 - by proposing a **restricted form of handlers** for resources
 - that supports **controlled initialisation** and **finalisation**,
 - (and in the future 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 ($A_{\text{op}}, B_{\text{op}}$ are sets)

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

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

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

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

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

²In 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 ($A_{\text{op}}, B_{\text{op}}$ are sets)

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

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

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

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

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

$$\left\{ \text{get} : \mathbb{1} \rightsquigarrow S \quad , \quad \text{set} : S \rightsquigarrow \mathbb{1} \right\}$$

is given by

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

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

²In 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
- **linear-use state-passing translation**
[Møgelberg and Staton '11, '14]
- **stateful running** of algebraic effects
[Uustalu '15]

A natural model of **top-level runtime** ctd.

- Runners/comodels have been used for
 - **operational semantics** using tensors of models and comodels [Plotkin and Power '08]
 - **top-level implementation of algebraic effects** in \mathbf{EFF} [Bauer and Pretnar '15]and
- **linear-use state-passing translation** [Møgelberg and Staton '11, '14]
- **stateful running** of algebraic effects [Uustalu '15]
- The latter explicitly rely on one-to-one correspondence between
 - **runners** \mathcal{R}
 - **monad morphisms**³ $r : \mathbf{Free}_{\Sigma}(-) \longrightarrow \mathbf{St}_{|\mathcal{R}|}$

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

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.,

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

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.,

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

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

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

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.,

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

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

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

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

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

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

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

Effectful runners for modular top-levels

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

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

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

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

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

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

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

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

- 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) $\text{getenv} : \mathbb{1} \rightsquigarrow C$ & $\text{setenv} : C \rightsquigarrow \mathbb{1}$
 - (ii) $\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$ ($\text{op} \in \Sigma'$, for some external Σ')
 - (iii) $\text{kill} : S \rightsquigarrow \mathbb{0}$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) $\text{getenv} : \mathbb{1} \rightsquigarrow C$ & $\text{setenv} : C \rightsquigarrow \mathbb{1}$
 - (ii) $\text{op} : A_{\text{op}} \rightsquigarrow B_{\text{op}}$ ($\text{op} \in \Sigma'$, for some external Σ')
 - (iii) $\text{kill} : S \rightsquigarrow \mathbb{0}$s.t., (i) satisfy state equations; and (i) commute with (ii) and (iii)
- The **induced monad** is then isomorphic to

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

Effectful runners for modular top-levels ctd.

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

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

Effectful runners for modular top-levels ctd.

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

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

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

Effectful runners for modular top-levels ctd.

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

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

- Observe that raising signals in S **discards the state**,
but **not all problems are terminal**—they can be recovered from
- Our solution:** consider signatures Σ with operation symbols

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

Effectful runners for modular top-levels ctd.

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

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

- Observe that raising signals in S **discards the state**,
but **not all problems are terminal**—they can be recovered from
- Our solution:** consider signatures Σ with operation symbols

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

Effectful runners for modular top-levels ctd.

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

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

- Observe that raising signals in S **discards the state**,
but **not all problems are terminal**—they can be recovered from
- Our solution:** consider signatures Σ with operation symbols

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

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

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

where we call $\mathbf{K}_{\Sigma, E, S, C}$ a **kernel monad** (the sum of **T** and excs.)

$$\mathbf{K}_{\Sigma', E_{\text{op}}, S, C} B_{\text{op}} \stackrel{\text{def}}{=} C \Rightarrow \mathbf{Free}_{\Sigma'}((B_{\text{op}} + E_{\text{op}}) \times C + S)$$

T-runners as a programming construct
(towards a core calculus for runners)

T-runners as a programming construct

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

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

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

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

T-runners as a programming construct

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

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

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

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

- For instance, we can provide **write-only file access** as

```
let RFH = {  
  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

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

$$(\text{fwrite} : \text{FileHandle} \times \text{String} \rightsquigarrow 1 ! E) \in \Sigma' \quad S = \{ \text{IOError} \}$$

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

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

- Recall that the components r_X of the monad morphism

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

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

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

```
using R @ Minit  
run M  
finally { return x @ c → Mret , ... raise e @ c → Me ... , ... kill s → Ms ... }
```

where

(**user monads**)

- M_s 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

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

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

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

```
using R @ Minit
run M
finally {return x @ c → Mret , ... raise e @ c → Me ... , ... kill s → Ms ...}
```

where

(**user monads**)

- M_s 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 RFH @ (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** `write : String → 1 ! E ∪ {WriteSizeExceeded}`
but **not** (the external operations) `fopen` , `fclose` , and `fwrite`
- `fopen` and `fclose` are **limited to initialisation-finalisation**
- M can itself also catch `WriteSizeExceeded` to **re-try writing**

**A core calculus for
programming with runners**

Core calculus (types and judgements)

Core calculus (types and judgements)

- **Values**

$$\Gamma \vdash V : X$$

$$\Gamma \vdash V \equiv W : X$$

- **User computations**

$$\Gamma \vdash M : X ! (\Sigma, E)$$

$$\Gamma \vdash M \equiv N : X ! (\Sigma, E)$$

- **Kernel computations**

$$\Gamma \vdash K : X \Downarrow (\Sigma, E, S, C)$$

$$\Gamma \vdash K \equiv L : X \Downarrow (\Sigma, E, S, C)$$

Core calculus (types and judgements)

- **Values**

$$\Gamma \vdash V : X$$

$$\Gamma \vdash V \equiv W : X$$

- **User computations**

$$\Gamma \vdash M : X ! (\Sigma, E)$$

$$\Gamma \vdash M \equiv N : X ! (\Sigma, E)$$

- **Kernel computations**

$$\Gamma \vdash K : X \Downarrow (\Sigma, E, S, C)$$

$$\Gamma \vdash K \equiv L : X \Downarrow (\Sigma, E, S, C)$$

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

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

- **Types**

$$X, Y ::= B \mid 1 \mid 0 \mid X \times Y \mid X + Y$$

$$\mid X \rightarrow Y ! (\Sigma, E)$$

$$\mid X \rightarrow Y \Downarrow (\Sigma, E, S, C)$$

$$\mid \Sigma \Rightarrow (\Sigma', S, C)$$

Core calculus (user computations)

$M, N ::= \text{return } V$

value

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

exception handler

| $V W$

application

| $\text{match } V \text{ with } \{\langle x, y \rangle \mapsto M\}$

product elimination

| $\text{match } V \text{ with } \{\} X$

empty elimination

| $\text{match } V \text{ with } \{\text{inl } x \mapsto M, \text{inr } y \mapsto N\}$

sum elimination

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

operation call

| $\text{raise}_X e$

raise exception

| $\text{using } V @ W \text{ run } M \text{ finally } \{$

run

$\text{return } x @ c \mapsto N,$

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

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

| $\text{kernel } K @ V \text{ finally } \{$

switch to kernel mode

$\text{return } x @ c \mapsto N,$

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

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

Core calculus (kernel computations)

$K, L ::=$	<code>return_C V</code>	value
	<code>try K with {return $x \mapsto L$, (raise $e \mapsto L_e$)_{$e \in E$}}</code>	exception handler
	<code>V W</code>	application
	<code>match V with {$\langle x, y \rangle \mapsto K$}</code>	product elimination
	<code>match V with {}_{$X @ C$}</code>	empty elimination
	<code>match V with {inl $x \mapsto K$, inr $y \mapsto L$}</code>	sum elimination
	<code>op_{$X @ C$}(V, ($x . K$), (L_e)_{$e \in E_{\text{op}}$})</code>	operation call
	<code>raise_{$X @ C$} e</code>	raise exception
	<code>kill_{$X @ C$} s</code>	send signal
	<code>getenv_C($c . K$)</code>	get state
	<code>setenv(V, K)</code>	set state
	<code>user M with {return $x \mapsto K$, (raise $e \mapsto L_e$)_{$e \in E$}}</code>	switch to user mode

Core calculus (type system)

Core calculus (type system)

- For example, the **typing rule for runners** is

$$\frac{\Sigma = \{ \text{op}_1, \dots, \text{op}_n \} \quad \left(\Gamma, x_i : A_{\text{op}_i} \vdash K_i : B_{\text{op}_i} \not\Downarrow (\Sigma', E_{\text{op}_i}, S, C) \right)_{1 \leq i \leq n}}{\Gamma \vdash \{ \text{op}_1 x_1 \mapsto K_1, \dots, \text{op}_n x_n \mapsto K_n \}_C : \Sigma \Rightarrow (\Sigma', S, C)}$$

Core calculus (type system)

- For example, the **typing rule for runners** is

$$\frac{\Sigma = \{ \text{op}_1, \dots, \text{op}_n \} \quad \left(\Gamma, x_i : A_{\text{op}_i} \vdash K_i : B_{\text{op}_i} \not\Downarrow (\Sigma', E_{\text{op}_i}, S, C) \right)_{1 \leq i \leq n}}{\Gamma \vdash \{ \text{op}_1 x_1 \mapsto K_1, \dots, \text{op}_n x_n \mapsto K_n \}_C : \Sigma \Rightarrow (\Sigma', S, C)}$$

- and the **typing rule for running user comps.** is

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

Core calculus (equational theory)

Core calculus (equational theory)

- For example, the β -equations for running user comps. are

$$\Gamma \vdash \text{using } V @ W \text{ run } (\text{return } V') \text{ finally } F \equiv N_{\text{ret}}[V'/x, W/c] : Y! (\Sigma', E')$$

Core calculus (equational theory)

- For example, the β -equations for running user comps. are

$$\Gamma \vdash \text{using } V @ W \text{ run } (\text{return } V') \text{ finally } F \equiv N_{ret}[V'/x, W/c] : Y! (\Sigma', E')$$

$$\Gamma \vdash \text{using } V @ W \text{ run } (\text{raise}_x e) \text{ finally } F \equiv N_e[W/c] : Y! (\Sigma', E')$$

Core calculus (equational theory)

- For example, the β -equations for running user comps. are

$$\Gamma \vdash \text{using } V @ W \text{ run } (\text{return } V') \text{ finally } F \equiv N_{ret}[V'/x, W/c] : Y! (\Sigma', E')$$

$$\Gamma \vdash \text{using } V @ W \text{ run } (\text{raise}_x e) \text{ finally } F \equiv N_e[W/c] : Y! (\Sigma', E')$$

$$\Gamma \vdash \text{using } R @ W \text{ run } (\text{op}_X (V, (y.M), (M_e)_{e \in E_{op}})) \text{ finally } F$$

$$\begin{aligned} &\equiv \text{kernel } K_{op}[V/x_{op}] @ W \text{ finally } \{ \\ &\quad \text{return } y @ c' \mapsto \text{using } R @ c' \text{ run } M \text{ finally } F, \\ &\quad (\text{raise } e @ c' \mapsto \text{using } R @ c' \text{ run } M_e \text{ finally } F)_{e \in E_{op}}, \\ &\quad (\text{kill } s \mapsto N_s)_{s \in S} \} : Y! (\Sigma', E') \end{aligned}$$

Core calculus (equational theory)

- For example, the β -equations for running user comps. are

$$\Gamma \vdash \text{using } V @ W \text{ run } (\text{return } V') \text{ finally } F \equiv N_{ret}[V'/x, W/c] : Y! (\Sigma', E')$$

$$\Gamma \vdash \text{using } V @ W \text{ run } (\text{raise}_x e) \text{ finally } F \equiv N_e[W/c] : Y! (\Sigma', E')$$

$$\begin{aligned} \Gamma \vdash & \text{using } R @ W \text{ run } (\text{op}_X (V, (y.M), (M_e)_{e \in E_{op}})) \text{ finally } F \\ & \equiv \text{kernel } K_{op}[V/x_{op}] @ W \text{ finally } \{ \\ & \quad \text{return } y @ c' \mapsto \text{using } R @ c' \text{ run } M \text{ finally } F, \\ & \quad (\text{raise } e @ c' \mapsto \text{using } R @ c' \text{ run } M_e \text{ finally } F)_{e \in E_{op}}, \\ & \quad (\text{kill } s \mapsto N_s)_{s \in S} \} : Y! (\Sigma', E') \end{aligned}$$

and the β -equation for signal handling is

$$\Gamma \vdash \text{kernel } (\text{kill}_{x@c} s) @ W \text{ finally } F \equiv N_s : Y! (\Sigma', E')$$

Core calculus (equational theory)

- For example, the β -equations for running user comps. are

$$\Gamma \vdash \text{using } V @ W \text{ run } (\text{return } V') \text{ finally } F \equiv N_{ret}[V'/x, W/c] : Y! (\Sigma', E')$$

$$\Gamma \vdash \text{using } V @ W \text{ run } (\text{raise}_x e) \text{ finally } F \equiv N_e[W/c] : Y! (\Sigma', E')$$

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

and the β -equation for signal handling is

$$\Gamma \vdash \text{kernel } (\text{kill}_{x@c} s) @ W \text{ finally } F \equiv N_s : Y! (\Sigma', E')$$

and kernel comp. equations include **kernel theory equations**

Core calculus (subtyping)

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

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

$$\frac{\Gamma \vdash M : A! (\Sigma, E) \quad \Sigma \subseteq \Sigma' \quad A <: B \quad E \subseteq E'}{\Gamma \vdash M : B! (\Sigma', E')}$$

$$\frac{\begin{array}{cccc} \Gamma \vdash K : A \downarrow (\Sigma, E, S, C) & \Sigma \subseteq \Sigma' & & \\ A <: B & E \subseteq E' & S \subseteq S' & C \equiv C' \end{array}}{\Gamma \vdash K : B \downarrow (\Sigma', E', S', C')}$$

Core calculus (subtyping)

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

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

$$\frac{\Gamma \vdash M : A!(\Sigma, E) \quad \Sigma \subseteq \Sigma' \quad A <: B \quad E \subseteq E'}{\Gamma \vdash M : B!(\Sigma', E')}$$

$$\frac{\begin{array}{cccc} \Gamma \vdash K : A \downarrow (\Sigma, E, S, C) & \Sigma \subseteq \Sigma' & & \\ A <: B & E \subseteq E' & S \subseteq S' & C \equiv C' \end{array}}{\Gamma \vdash K : B \downarrow (\Sigma', E', S', C')}$$

- We use $C \equiv C'$ 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** $\mathbf{U}_{\Sigma, E} X \stackrel{\text{def}}{=} \mathbf{Free}_{\Sigma}(X + E)$
 - **kernel monads** $\mathbf{K}_{\Sigma, E, S, C} X \stackrel{\text{def}}{=} C \Rightarrow \mathbf{Free}_{\Sigma}((X + E) \times C + S)$

Core calculus (semantics)

- **Monadic semantics**, for concreteness in **Set**, using
 - **user monads** $\mathbf{U}_{\Sigma, E} X \stackrel{\text{def}}{=} \mathbf{Free}_{\Sigma}(X + E)$
 - **kernel monads** $\mathbf{K}_{\Sigma, E, S, C} X \stackrel{\text{def}}{=} C \Rightarrow \mathbf{Free}_{\Sigma}((X + E) \times C + S)$
- (At a high level) the **judgements are interpreted** as maps

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

$$\llbracket \Gamma \vdash M : X ! (\Sigma, E) \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \mathbf{U}_{\Sigma, E} \llbracket X \rrbracket$$

$$\llbracket \Gamma \vdash K : X \downarrow (\Sigma, E, S, C) \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \mathbf{K}_{\Sigma, E, S, \llbracket C \rrbracket} \llbracket X \rrbracket$$

Core calculus (semantics)

- **Monadic semantics**, for concreteness in **Set**, using
 - **user monads** $\mathbf{U}_{\Sigma, E} X \stackrel{\text{def}}{=} \mathbf{Free}_{\Sigma}(X + E)$
 - **kernel monads** $\mathbf{K}_{\Sigma, E, S, C} X \stackrel{\text{def}}{=} C \Rightarrow \mathbf{Free}_{\Sigma}((X + E) \times C + S)$
- (At a high level) the **judgements are interpreted** as maps

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

$$\llbracket \Gamma \vdash M : X ! (\Sigma, E) \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \mathbf{U}_{\Sigma, E} \llbracket X \rrbracket$$

$$\llbracket \Gamma \vdash K : X \not\downarrow (\Sigma, E, S, C) \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \mathbf{K}_{\Sigma, E, S, \llbracket C \rrbracket} \llbracket X \rrbracket$$

- **Theorem:** The semantics is coherent (**subtyping!**) and sound.

Core calculus (semantics ctd.)

- In order to prove **coherence** of the semantics, we actually work in the (total category **Sub**(**Set**) of the) **subset fibration**

Core calculus (semantics ctd.)

- In order to prove **coherence** of the semantics, we actually work in the (total category **Sub(Set)** of the) **subset fibration**
- For instance, **kernel computations** are interpreted as

$$\begin{array}{ccc}
 \llbracket \Gamma \rrbracket & \xrightarrow{\llbracket \Gamma \vdash K : X \Downarrow (\Sigma, E, S, C) \rrbracket} & \mathbf{K}_{\Sigma, E, S, \llbracket C \rrbracket} \llbracket X \rrbracket \\
 \subseteq \downarrow & & \downarrow \subseteq \\
 \llbracket \Gamma^s \rrbracket & \xrightarrow{\llbracket \Gamma^s \vdash K : X^s \Downarrow C \rrbracket} & \mathbf{K}_{\mathcal{O} + \{\perp\}, \mathcal{E}, S, \llbracket C \rrbracket} \llbracket X^s \rrbracket
 \end{array}$$

where $\Gamma^s \vdash K : X^s \Downarrow C$ is a **skeletal kernel typing judgement** and use the extra op. $\perp : 1 \rightsquigarrow 0 ! \{ \}$ to model **runtime errors**

Core calculus (semantics ctd.)

- In order to prove **coherence** of the semantics, we actually work in the (total category **Sub(Set)** of the) **subset fibration**
- For instance, **kernel computations** are interpreted as

$$\begin{array}{ccc}
 \llbracket \Gamma \rrbracket & \xrightarrow{\llbracket \Gamma \vdash K : X \Downarrow (\Sigma, E, S, C) \rrbracket} & \mathbf{K}_{\Sigma, E, S, \llbracket C \rrbracket} \llbracket X \rrbracket \\
 \subseteq \downarrow & & \downarrow \subseteq \\
 \llbracket \Gamma^s \rrbracket & \xrightarrow{\llbracket \Gamma^s \vdash K : X^s \Downarrow C \rrbracket} & \mathbf{K}_{\mathcal{O} + \{\perp\}, \mathcal{E}, S, \llbracket C \rrbracket} \llbracket X^s \rrbracket
 \end{array}$$

where $\Gamma^s \vdash K : X^s \Downarrow C$ is a **skeletal kernel typing judgement** and use the extra op. $\perp : 1 \rightsquigarrow 0 ! \{ \}$ to model **runtime errors**

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

Core calculus (semantics ctd.)

- In order to prove **coherence** of the semantics, we actually work in the (total category **Sub(Set)** of the) **subset fibration**
- For instance, **kernel computations** are interpreted as

$$\begin{array}{ccc}
 \llbracket \Gamma \rrbracket & \xrightarrow{\llbracket \Gamma \vdash K : X \Downarrow (\Sigma, E, S, C) \rrbracket} & \mathbf{K}_{\Sigma, E, S, \llbracket C \rrbracket} \llbracket X \rrbracket \\
 \subseteq \downarrow & & \downarrow \subseteq \\
 \llbracket \Gamma^s \rrbracket & \xrightarrow{\llbracket \Gamma^s \vdash K : X^s \Downarrow C \rrbracket} & \mathbf{K}_{\mathcal{O} + \{\perp\}, \mathcal{E}, S, \llbracket C \rrbracket} \llbracket X^s \rrbracket
 \end{array}$$

where $\Gamma^s \vdash K : X^s \Downarrow C$ is a **skeletal kernel typing judgement** and use the extra op. $\perp : 1 \rightsquigarrow 0 ! \{ \}$ to model **runtime errors**

- No essential obstacles to extending to **Sub(Cpo)** and beyond
- Ground type restriction** on C simplifies the sem. ($\llbracket C \rrbracket = \llbracket C \rrbracket$)

Core calculus (semantics ctd.)

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

Core calculus (semantics ctd.)

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

- $\llbracket V \rrbracket_\gamma = \mathcal{R} = \left(\overline{\text{op}}_{\mathcal{R}} : \llbracket A_{\text{op}} \rrbracket \longrightarrow \mathbf{K}_{\Sigma', E_{\text{op}}, S, \llbracket C \rrbracket} \llbracket B_{\text{op}} \rrbracket \right)_{\text{op} \in \Sigma}$
- $\llbracket \text{return } x @ c \mapsto N_{ret} \rrbracket_\gamma \in \llbracket X \rrbracket \times \llbracket C \rrbracket \longrightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket$
- $\llbracket (\text{raise } e @ c \mapsto N_e)_{e \in E} \rrbracket_\gamma \in E \times \llbracket C \rrbracket \longrightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket$
- $\llbracket (\text{kill } s \mapsto N_s)_{s \in S} \rrbracket_\gamma \in S \longrightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket$

Core calculus (semantics ctd.)

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

- $\llbracket V \rrbracket_\gamma = \mathcal{R} = \left(\overline{\text{op}}_{\mathcal{R}} : \llbracket A_{\text{op}} \rrbracket \longrightarrow \mathbf{K}_{\Sigma', E_{\text{op}}, S, \llbracket C \rrbracket} \llbracket B_{\text{op}} \rrbracket \right)_{\text{op} \in \Sigma}$
- $\llbracket \text{return } x @ c \mapsto N_{ret} \rrbracket_\gamma \in \llbracket X \rrbracket \times \llbracket C \rrbracket \longrightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket$
- $\llbracket (\text{raise } e @ c \mapsto N_e)_{e \in E} \rrbracket_\gamma \in E \times \llbracket C \rrbracket \longrightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket$
- $\llbracket (\text{kill } s \mapsto N_s)_{s \in S} \rrbracket_\gamma \in S \longrightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket$
- allowing us to use the **free model property** to construct

$$\mathbf{U}_{\Sigma, E} \llbracket X \rrbracket \xrightarrow{r_{\llbracket X \rrbracket} + E} \mathbf{K}_{\Sigma', E, S, \llbracket C \rrbracket} \llbracket X \rrbracket \xrightarrow{(\lambda \llbracket N_{ret} \rrbracket_\gamma)^\dagger} \underbrace{\llbracket C \rrbracket \Rightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket}_{\text{carrier of ker. th. model}}$$

Core calculus (semantics ctd.)

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

- $\llbracket V \rrbracket_\gamma = \mathcal{R} = \left(\overline{\text{op}}_{\mathcal{R}} : \llbracket A_{\text{op}} \rrbracket \longrightarrow \mathbf{K}_{\Sigma', E_{\text{op}}, S, \llbracket C \rrbracket} \llbracket B_{\text{op}} \rrbracket \right)_{\text{op} \in \Sigma}$
- $\llbracket \text{return } x @ c \mapsto N_{ret} \rrbracket_\gamma \in \llbracket X \rrbracket \times \llbracket C \rrbracket \longrightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket$
- $\llbracket (\text{raise } e @ c \mapsto N_e)_{e \in E} \rrbracket_\gamma \in E \times \llbracket C \rrbracket \longrightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket$
- $\llbracket (\text{kill } s \mapsto N_s)_{s \in S} \rrbracket_\gamma \in S \longrightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket$
- allowing us to use the **free model property** to construct

$$\mathbf{U}_{\Sigma, E} \llbracket X \rrbracket \xrightarrow{r_{\llbracket X \rrbracket + E}} \mathbf{K}_{\Sigma', E, S, \llbracket C \rrbracket} \llbracket X \rrbracket \xrightarrow{(\lambda \llbracket N_{ret} \rrbracket_\gamma)^\dagger} \underbrace{\llbracket C \rrbracket \Rightarrow \mathbf{U}_{\Sigma', E'} \llbracket Y \rrbracket}_{\text{carrier of ker. th. model}}$$

- and then apply the resulting composite map to

$$\llbracket M \rrbracket_\gamma \in \mathbf{U}_{\Sigma, E} \llbracket X \rrbracket \quad \text{and} \quad \llbracket W \rrbracket_\gamma \in \llbracket C \rrbracket$$

Core calculus (finalisation)

$$\Gamma \vdash \text{using } V @ W \text{ run } M \text{ finally } \left\{ \begin{array}{l} \text{return } x @ c \mapsto N_{ret} , \\ (\text{raise } e @ c \mapsto N_e)_{e \in E} , \\ (\text{kill } s \mapsto N_s)_{s \in S} \end{array} \right\} : Y ! (\Sigma', E')$$

Core calculus (finalisation)

$$\Gamma \vdash \text{using } V @ W \text{ run } M \text{ finally } \{ \text{return } x @ c \mapsto N_{ret} , \\ (\text{raise } e @ c \mapsto N_e)_{e \in E} , \\ (\text{kill } s \mapsto N_s)_{s \in S} \} : Y ! (\Sigma', E')$$

- The **finally** block $(N_{ret}, N_e, \dots, N_s, \dots)$ determines **fin. maps**

$$\phi_\gamma : ([X] + E) \times [C] + S \longrightarrow \mathbf{U}_{\Sigma', E'} [Y] \quad (\gamma \in [\Gamma])$$

Core calculus (finalisation)

$$\Gamma \vdash \text{using } V @ W \text{ run } M \text{ finally } \{ \text{return } x @ c \mapsto N_{ret} , \\ (\text{raise } e @ c \mapsto N_e)_{e \in E} , \\ (\text{kill } s \mapsto N_s)_{s \in S} \} : Y ! (\Sigma', E')$$

- The **finally** block $(N_{ret}, N_e, \dots, N_s, \dots)$ determines **fin. maps**

$$\phi_\gamma : ([\![X]\!] + E) \times [\![C]\!] + S \longrightarrow \mathbf{U}_{\Sigma', E'} [\![Y]\!] \quad (\gamma \in [\![\Gamma]\!])$$

- Theorem (finalisation):**

- for every environment $\gamma \in [\![\Gamma]\!]$,
- there exists a comp. tree $t \in \mathbf{Free}_{\Sigma'}(([\![X]\!] + E) \times [\![C]\!] + S)$,
- such that **running factors through finalisation**, i.e.,

$$[\![\Gamma \vdash \text{using } V @ W \text{ run } M \text{ finally } F : Y ! (\Sigma', E')]\!]_\gamma = \phi_\gamma^\dagger t$$

Implementing runners

Experimenting with the **theory in practice**

Experimenting with the theory in practice

- A **small experimental language** COOP⁴
 - Implements the core calculus with few extras
 - The interpreter is directly based on the equational theory
 - Top-level containers for running external (OCaml) code
 - <https://github.com/andrejbauer/coop>

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

Experimenting with the **theory in practice**

- A **small experimental language** COOP⁴
 - Implements the core calculus with few extras
 - The interpreter is directly based on the equational theory
 - Top-level containers for running external (OCaml) code
 - <https://github.com/andrejbauer/coop>
- A **HASKELL library** HASKELL-COOP
 - A shallow-embedding of the core calculus in HASKELL
 - Uses one of the Freer monad implementations underneath
 - Operational aspects implement the denotational semantics
 - Top-level containers for arbitrary HASKELL monads
 - Examples make use of HASKELL's features (GADTs, ...)
 - <https://github.com/danelahman/haskell-coop>

⁴coop [/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 RFH @ (fopen fileName)
run (
 using RFC @ (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 RFC = {
 write strl → let str = getenv () in
 if (length (str^strl) > max) then (raise WriteSizeExceeded)
 else (setenv (str^strl))
}String
```



# Vertical nesting for instrumentation

# Vertical nesting for instrumentation

- ```
using RCost @ (return 0)
run M
finally {
  return x @ c → report_cost c; return x ,
  raise e @ c → report_cost c; raise e ,
  kill s → raise SignalHappenedException }
```

where the **cost model runner** is defined as

```
let RCost = {
  ... ,
  op a → let c = getenv () in
    setenv (c + 1);
    op a ,
  ...
}Nat
```

(* forwards op outwards *)

- The runner R_{Cost} implements the same sig. Σ that M is using
- As a result, the runner R_{Cost} 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 \rightarrow (Int + 1)) \times Nat

type Ref = Nat

```
let RIntState = {  
  alloc x  $\rightarrow$  let h = getenv () in (* alloc : Int  $\rightsquigarrow$  Ref ! {} *)  
    let (r, h') = heapAlloc h x in  
    setenv h';  
    return r ,  
  
  deref r  $\rightarrow$  let h = getenv () in (* deref : Ref  $\rightsquigarrow$  Int ! {} *)  
    match (heapSel h r) with  
    | inl x  $\rightarrow$  return x  
    | inr ()  $\rightarrow$  kill ReferenceDoesNotExist ,  
  
  assign r y  $\rightarrow$  let h = getenv () in (* assign : Ref  $\times$  Int  $\rightsquigarrow$  1 ! {} *)  
    match (heapUpd h r y) with  
    | inl h'  $\rightarrow$  setenv h'  
    | inr ()  $\rightarrow$  kill ReferenceDoesNotExist  
}
```

IntHeap

Vertical nesting for **active monitoring** ctd.

- Next, we define F^* -style **monotonic state** on top of R_{IntState}

Vertical nesting for active monitoring ctd.

- Next, we define F^* -style **monotonic state** on top of R_{IntState}

type MonMemory = Ref \rightarrow (Ord + 1) **type** Ord = Int \rightarrow Int \rightarrow Bool

```
let RMonState = {  
  mAlloc x rel  $\rightarrow$  let r = alloc x in      (* : Int  $\times$  Ord  $\rightsquigarrow$  Ref ! { } *)  
    let m = getenv () in  
    setenv (memAdd m r rel);  
    return r,  
  
  mDeref r  $\rightarrow$  deref r ,      (* : Ref  $\rightsquigarrow$  Int ! { } *)  
  
  mAssign r y  $\rightarrow$  let x = deref r in      (* : Ref  $\times$  Int  $\rightsquigarrow$  1 ! {MV} *)  
    let m = getenv () in  
    match (memSel m r) with  
    | inl rel  $\rightarrow$  if (rel x y)  
        then (assign r y)  
        else (raise MonotonicityViolation)  
    | inr  $\rightarrow$  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

let $R_1 = \{ \dots, \text{op}_{1i} \ x_{1i} \rightarrow K_{1i}, \dots \}_{C_1}$ $(* : \Sigma_1 \Rightarrow (\Sigma'_1, S_1, C_1) *)$
let $R_2 = \{ \dots, \text{op}_{2j} \ x_{2j} \rightarrow K_{2j}, \dots \}_{C_2}$ $(* : \Sigma_2 \Rightarrow (\Sigma'_2, S_2, C_2) *)$

we can **pair them** to get the runner

let $R = \{ \dots, \quad (* : \Sigma_1 + \Sigma_2 \Rightarrow (\Sigma'_1 + \Sigma'_2, S_1 + S_2, C_1 \times C_2) *)$
 $\text{op}_{1i} \ x_{1i} \rightarrow$ **let** $(c, c') = \text{getenv} ()$ **in**
 user (**kernel** $(K_{1i} \ x_{1i})$ **@** c **finally** {
 return y **@** $c'' \rightarrow$ **return** $(\text{inl} (\text{inl } y, c''))$,
 raise e **@** $c'' \rightarrow$ **return** $(\text{inl} (\text{inr } e, c''))$, $(* e \in E_{\text{op}_{1i}} *)$
 kill $s \rightarrow$ **return** $(\text{inr } s)$ } $(* s \in S_1 *)$
 finally {
 return $(\text{inl} (\text{inl } y, c'')) \rightarrow$ **setenv** (c'', c') ; **return** y ,
 return $(\text{inl} (\text{inr } e, c'')) \rightarrow$ **setenv** (c'', c') ; **raise** e ,
 return $(\text{inr } s) \rightarrow$ **kill** s },
 $\dots,$
 $\text{op}_{2j} \ x_{2j} \rightarrow \dots, \dots \}_{C_1 \times C_2}$

Runners can also be horizontally paired

- Given runners for

```
let R1 = { ... , op1i x1i → K1i , ... }C1 (* : Σ1 ⇒ (Σ'1, S1, C1) *)  
let R2 = { ... , op2j x2j → K2j , ... }C2 (* : Σ2 ⇒ (Σ'2, S2, C2) *)
```

we can **pair them** to get the runner

```
let R = { ... , (* : Σ1 + Σ2 ⇒ (Σ'1 + Σ'2, S1 + S2, C1 × C2) *)  
  op1i x1i → let (c, c'') = getenv () in  
    user (kernel (K1i x1i) @ c finally {  
      return y @ c'' → return (inl (inl y, c'')),  
      raise e @ c'' → return (inl (inr e, c'')), (* e ∈ Eop1i *)  
      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 x2j → ... , ... }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

Other examples (ambient vals. and funs.)

- Ambient values

```
ambient val f : int → int
```

```
ambient val x : int
```

```
with val x = 4
```

```
with val f = fun y → x + y
```

```
with val x = 2
```

```
f 1
```

(* Returns 3 *)

Other examples (ambient vals. and funs.)

- Ambient values

```
ambient val f : int → int
```

```
ambient val x : int
```

```
with val x = 4
```

```
with val f = fun y → x + y
```

```
with val x = 2
```

```
f 1
```

(* Returns 3 *)

- Ambient functions

```
ambient fun f : int → int
```

```
ambient val x : int
```

```
with val x = 4
```

```
with fun f = fun y → x + y
```

```
with val x = 2
```

```
f 1
```

(* Returns 5 *)

Other examples (ambient vals. and funs.)

```
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
```


Other examples (ambient vals. and funs.)

```
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
```

```
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
```

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 **(useful ?) programming construct** with controlled **initialisation** and **finalisation**
- Despite being **affine**, **T-runners** still **cover a lot of ground**
- Two **implementations**: COOP & HASKELL-COOP
- **Ongoing** and **future**: handlers, lenses in subtyping and semantics, cat. of runners, concurrency, refinement typing, compilation, ...

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-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.