

Effect handler oriented programming

Sam Lindley

The University of Edinburgh

SPLV 2024

Effect handler oriented programming languages

Eff	https://www.eff-lang.org/
Effekt	https://effekt-lang.org/
Frank	https://github.com/frank-lang/frank
Helium	https://bitbucket.org/pl-uwr/helium
Links	https://www.links-lang.org/
Koka	https://github.com/koka-lang/koka
OCaml 5	https://github.com/ocaml-labs/ocaml-multicore/wiki
Unison	https://www.unison-lang.org/

Resources



Jeremy Yallop's effects bibliography

<https://github.com/yallop/effects-bibliography>



Matija Pretnar's tutorial

[“An introduction to algebraic effects and handlers”](#), MFPS 2015



Andrej Bauer's tutorial

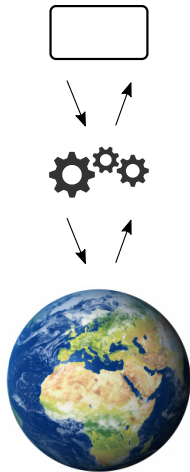
[“What is algebraic about algebraic effects and handlers?”](#), OPLSS 2018



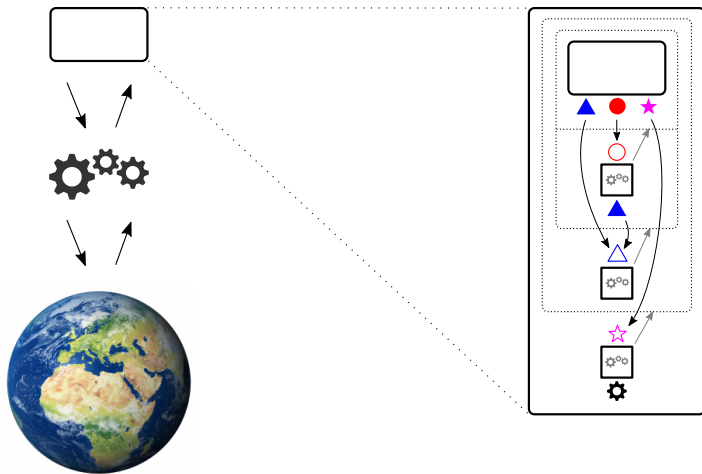
Daniel Hillerström's PhD thesis

[“Foundations for programming and implementing effect handlers”](#), 2022

Effect handlers as composable user-defined operating systems



Effect handlers as composable user-defined operating systems



Effect handlers for operating systems

EIO — effects-based direct-style concurrent I/O stack for OCaml

<https://github.com/ocaml-multicore/eio>

Composing UNIX with effect handlers

Foundations for programming and implementing effect handlers, Chapter 2

Daniel Hillerström, PhD thesis, The University of Edinburgh, 2022

<https://www.dhil.net/research/papers/thesis.pdf>

Example 1: choice and failure

Drunk coin tossing

$\text{toss} : 1 \rightarrow \text{Toss!}(E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}\})$

$\text{toss}() = \text{if } \text{choose}() \text{ then Heads else Tails}$

$\text{drunkToss} : 1 \rightarrow \text{Toss!}(E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}, \text{fail} : a.1 \rightarrow a\})$

$\text{drunkToss}() = \text{if } \text{choose}() \text{ then}$

$\text{if } \text{choose}() \text{ then Heads else Tails}$

else

$\text{fail}()$

$\text{drunkTosses} : \text{Nat} \rightarrow \text{List Toss!}(E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}, \text{fail} : a.1 \rightarrow a\})$

$\text{drunkTosses } n = \text{if } n = 0 \text{ then } []$

$\text{else drunkToss}() :: \text{drunkTosses}(n - 1)$

Example 1: choice and failure

Handlers

$\text{maybeFail} : A!(E \uplus \{\text{fail} : a.1 \twoheadrightarrow a\}) \Rightarrow \text{Maybe } A!E$

$\text{maybeFail} =$ — exception handler

$\text{return } x \mapsto \text{Just } x$

$\langle \text{fail } () \rangle \mapsto \text{Nothing}$

Example 1: choice and failure

Handlers

$\text{maybeFail} : A!(E \uplus \{\text{fail} : a.1 \twoheadrightarrow a\}) \Rightarrow \text{Maybe } A!E$

$\text{maybeFail} =$ — exception handler

return $x \mapsto \text{Just } x$

$\langle \text{fail } () \rangle \mapsto \text{Nothing}$

handle 42 **with** maybeFail $\Rightarrow \text{Just } 42$

handle fail () **with** maybeFail $\Rightarrow \text{Nothing}$

Example 1: choice and failure

Handlers

$\text{maybeFail} : A!(E \uplus \{\text{fail} : a.1 \rightarrow a\}) \Rightarrow \text{Maybe } A!E$

$\text{maybeFail} =$ — exception handler

$\text{return } x \mapsto \text{Just } x$

$\langle \text{fail} () \rangle \mapsto \text{Nothing}$

$\text{handle } 42 \text{ with maybeFail} \Rightarrow \text{Just } 42$

$\text{handle fail} () \text{ with maybeFail} \Rightarrow \text{Nothing}$

$\text{trueChoice} : A!(E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}\}) \Rightarrow A!E$

$\text{trueChoice} =$ — linear handler

$\text{return } x \mapsto x$

$\langle \text{choose} () \rightarrow r \rangle \mapsto r \text{ tt}$

Example 1: choice and failure

Handlers

$\text{maybeFail} : A!(E \uplus \{\text{fail} : a.1 \rightarrow a\}) \Rightarrow \text{Maybe } A!E$

$\text{maybeFail} =$ — exception handler

$\text{return } x \mapsto \text{Just } x$

$\langle \text{fail} () \rangle \mapsto \text{Nothing}$

$\text{handle } 42 \text{ with maybeFail} \Rightarrow \text{Just } 42$

$\text{handle fail} () \text{ with maybeFail} \Rightarrow \text{Nothing}$

$\text{trueChoice} : A!(E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}\}) \Rightarrow A!E$

$\text{trueChoice} =$ — linear handler

$\text{return } x \mapsto x$

$\langle \text{choose} () \rightarrow r \rangle \mapsto r \text{ tt}$

$\text{handle } 42 \text{ with trueChoice} \Rightarrow 42$

$\text{handle toss} () \text{ with trueChoice} \Rightarrow \text{Heads}$

Example 1: choice and failure

Handlers

$\text{maybeFail} : A!(E \uplus \{\text{fail} : a.1 \rightarrow a\}) \Rightarrow \text{Maybe } A!E$

$\text{maybeFail} =$ — exception handler

$\text{return } x \mapsto \text{Just } x$

$\langle \text{fail} () \rangle \mapsto \text{Nothing}$

$\text{handle } 42 \text{ with maybeFail} \Rightarrow \text{Just } 42$

$\text{handle fail} () \text{ with maybeFail} \Rightarrow \text{Nothing}$

$\text{trueChoice} : A!(E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}\}) \Rightarrow A!E$

$\text{trueChoice} =$ — linear handler

$\text{return } x \mapsto x$

$\langle \text{choose} () \rightarrow r \rangle \mapsto r \text{ tt}$

$\text{handle } 42 \text{ with trueChoice} \Rightarrow 42$

$\text{handle toss} () \text{ with trueChoice} \Rightarrow \text{Heads}$

$\text{allChoices} : A!(E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}\}) \Rightarrow \text{List } A!E$

$\text{allChoices} =$ — non-linear handler

$\text{return } x \mapsto [x]$

$\langle \text{choose} () \rightarrow r \rangle \mapsto r \text{ tt} ++ r \text{ ff}$

Example 1: choice and failure

Handlers

$\text{maybeFail} : A!(E \uplus \{\text{fail} : a.1 \rightarrow a\}) \Rightarrow \text{Maybe } A!E$

$\text{maybeFail} =$ — exception handler

$\text{return } x \mapsto \text{Just } x$

$\langle \text{fail} () \rangle \mapsto \text{Nothing}$

$\text{handle } 42 \text{ with maybeFail} \Rightarrow \text{Just } 42$

$\text{handle fail} () \text{ with maybeFail} \Rightarrow \text{Nothing}$

$\text{trueChoice} : A!(E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}\}) \Rightarrow A!E$

$\text{trueChoice} =$ — linear handler

$\text{return } x \mapsto x$

$\langle \text{choose} () \rightarrow r \rangle \mapsto r \text{ tt}$

$\text{handle } 42 \text{ with trueChoice} \Rightarrow 42$

$\text{handle toss} () \text{ with trueChoice} \Rightarrow \text{Heads}$

$\text{allChoices} : A!(E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}\}) \Rightarrow \text{List } A!E$

$\text{allChoices} =$ — non-linear handler

$\text{return } x \mapsto [x]$

$\langle \text{choose} () \rightarrow r \rangle \mapsto r \text{ tt} ++ r \text{ ff}$

$\text{handle } 42 \text{ with allChoices} \Rightarrow [42]$

$\text{handle toss} () \text{ with allChoices} \Rightarrow [\text{Heads}, \text{Tails}]$

Example 1: choice and failure

Handler composition

handle (**handle** drunkTosses 2 **with** maybeFail) **with** allChoices

Example 1: choice and failure

Handler composition

handle (**handle** drunkTosses 2 **with** maybeFail) **with** allChoices : List (Maybe Toss) \Rightarrow

Example 1: choice and failure

Handler composition

handle (**handle** drunkTosses 2 **with** maybeFail) **with** allChoices : List (Maybe Toss) \Rightarrow
[Just [Heads, Heads], Just [Heads, Tails], Nothing,
Just [Tails, Heads], Just [Tails, Tails], Nothing,
Nothing]

Example 1: choice and failure

Handler composition

handle (**handle** drunkTosses 2 **with** maybeFail) **with** allChoices : List (Maybe Toss) \Rightarrow
[Just [Heads, Heads], Just [Heads, Tails], Nothing,
Just [Tails, Heads], Just [Tails, Tails], Nothing,
Nothing]

handle (**handle** drunkTosses 2 **with** allChoices) **with** maybeFail

Example 1: choice and failure

Handler composition

handle (**handle** drunkTosses 2 **with** maybeFail) **with** allChoices : List (Maybe Toss) \Rightarrow
[Just [Heads, Heads], Just [Heads, Tails], Nothing,
Just [Tails, Heads], Just [Tails, Tails], Nothing,
Nothing]

handle (**handle** drunkTosses 2 **with** allChoices) **with** maybeFail : Maybe (List Toss) \Rightarrow

Example 1: choice and failure

Handler composition

handle (**handle** drunkTosses 2 **with** maybeFail) **with** allChoices : List (Maybe Toss) \implies
[Just [Heads, Heads], Just [Heads, Tails], Nothing,
Just [Tails, Heads], Just [Tails, Tails], Nothing,
Nothing]

handle (**handle** drunkTosses 2 **with** allChoices) **with** maybeFail : Maybe (List Toss) \implies
Nothing

Example 2: generators

Effect signature

$\{\text{send} : \text{Nat} \rightarrow 1\}$

Example 2: generators

Effect signature

$$\{\text{send} : \text{Nat} \twoheadrightarrow 1\}$$

A simple generator

$\text{nats} : \text{Nat} \rightarrow 1!(E \uplus \{\text{send} : \text{Nat} \twoheadrightarrow 1\})$
 $\text{nats } n = \text{send } n; \text{nats } (n + 1)$

Example 2: generators

Effect signature

$$\{\text{send} : \text{Nat} \twoheadrightarrow 1\}$$

A simple generator

$$\begin{aligned}\text{nats} &: \text{Nat} \rightarrow 1!(E \uplus \{\text{send} : \text{Nat} \twoheadrightarrow 1\}) \\ \text{nats } n &= \text{send } n; \text{nats } (n + 1)\end{aligned}$$

Handler — parameterised handler

$$\begin{aligned}\text{until} &: \text{Nat} \rightarrow 1!(E \uplus \{\text{send} : \text{Nat} \twoheadrightarrow 1\}) \Rightarrow \text{List Nat!}E \\ \text{until } \text{stop} &= \\ &\quad \text{return } () \quad \mapsto [] \\ &\quad \langle \text{send } n \rightarrow r \rangle \mapsto \text{if } n < \text{stop} \text{ then } n :: r \text{ stop } () \\ &\quad \quad \text{else } []\end{aligned}$$

Example 2: generators

Effect signature

$\{\text{send} : \text{Nat} \twoheadrightarrow 1\}$

A simple generator

$\text{nats} : \text{Nat} \rightarrow 1!(E \uplus \{\text{send} : \text{Nat} \twoheadrightarrow 1\})$
 $\text{nats } n = \text{send } n; \text{nats } (n + 1)$

Handler — parameterised handler

$\text{until} : \text{Nat} \rightarrow 1!(E \uplus \{\text{send} : \text{Nat} \twoheadrightarrow 1\}) \Rightarrow \text{List Nat}!E$

$\text{until stop} =$

$\text{return } () \mapsto []$

$\langle \text{send } n \rightarrow r \rangle \mapsto \text{if } n < \text{stop} \text{ then } n :: r \text{ stop } ()$
 $\text{else } []$

$\text{handle nats } 0 \text{ with until } 8 \Rightarrow [0, 1, 2, 3, 4, 5, 6, 7]$

Operational semantics (parameterised handlers)

Reduction rules

let $x = V$ **in** $N \rightsquigarrow N[V/x]$
handle V **with** H $W \rightsquigarrow N[V/x, W/h]$
handle $\mathcal{E}[\text{op } V]$ **with** H $W \rightsquigarrow N_{\text{op}}[V/p, W/h, (\lambda h x. \text{handle } \mathcal{E}[x] \text{ with } H h)/r], \quad \text{op} \# \mathcal{E}$

where $H \ h = \text{return } x \quad \mapsto N$
 $\langle \text{op}_1 p \rightarrow r \rangle \mapsto N_{\text{op}_1}$
 \dots
 $\langle \text{op}_k p \rightarrow r \rangle \mapsto N_{\text{op}_k}$

Evaluation contexts

$\mathcal{E} ::= [] \mid \text{let } x = \mathcal{E} \text{ in } N \mid \text{handle } \mathcal{E} \text{ with } H \ W$

Operational semantics (parameterised handlers)

Reduction rules

let $x = V$ **in** $N \rightsquigarrow N[V/x]$
handle V **with** H $W \rightsquigarrow N[V/x, W/h]$
handle $\mathcal{E}[\text{op } V]$ **with** H $W \rightsquigarrow N_{\text{op}}[V/p, W/h, (\lambda h x. \text{handle } \mathcal{E}[x] \text{ with } H h)/r], \quad \text{op} \# \mathcal{E}$

where $H \mathbf{h} = \text{return } x \quad \mapsto N$
 $\langle \text{op}_1 p \rightarrow r \rangle \mapsto N_{\text{op}_1}$
 \dots
 $\langle \text{op}_k p \rightarrow r \rangle \mapsto N_{\text{op}_k}$

Evaluation contexts

$\mathcal{E} ::= [] \mid \text{let } x = \mathcal{E} \text{ in } N \mid \text{handle } \mathcal{E} \text{ with } H W$

Exercise: express parameterised handlers as deep handlers

Typing rules (parameterised handlers)

Effects

$$E ::= \emptyset \mid E \uplus \{\text{op} : A \twoheadrightarrow B\}$$

Computations

$$C, D ::= A!E$$

Operations

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \text{op } V : B!(E \uplus \{\text{op} : A \twoheadrightarrow B\})}$$

Handlers

$$\frac{\Gamma \vdash M : C \quad \Gamma \vdash V : P \quad \Gamma \vdash H : P \rightarrow C \Rightarrow D}{\Gamma \vdash \text{handle } M \text{ with } H \ V : D}$$
$$\frac{\begin{array}{c} \Gamma, h : P, x : A \vdash N : D \\ [\text{op}_i : A_i \twoheadrightarrow B_i \in E]_i \quad [\Gamma, h : P, p : A_i, r : P \rightarrow B_i \rightarrow D \vdash N_i : D]_i \end{array}}{\Gamma \vdash \lambda h. \text{return } x \mapsto N \quad (\langle \text{op}_i p \rightarrow r \rangle \mapsto N_i)_i : P \rightarrow A!E \Rightarrow D}$$

Example 3: cooperative concurrency (parameterised handler)

Effect signature

$$\{\text{yield} : 1 \twoheadrightarrow 1\}$$

Example 3: cooperative concurrency (parameterised handler)

Effect signature

$$\{\text{yield} : 1 \twoheadrightarrow 1\}$$

Two cooperative lightweight threads

```
tA () = print ("A1 "); yield (); print ("A2 ")
```

```
tB () = print ("B1 "); yield (); print ("B2 ")
```

Example 3: cooperative concurrency (parameterised handler)

Types

$$\text{Thread } E = 1 \rightarrow 1!(E \uplus \{\text{yield} : 1 \twoheadrightarrow 1\})$$

$$\text{Res } E = \text{List } (\text{Res } E) \rightarrow 1 \rightarrow 1!E$$

Handler

$$\text{coop} : \text{List } (\text{Res } E) \rightarrow 1!(E \uplus \{\text{yield} : 1 \twoheadrightarrow 1\}) \Rightarrow 1!E$$

$$\text{coop}([]) =$$

$$\text{return } () \quad \mapsto ()$$

$$\langle \text{yield } () \rightarrow r' \rangle \mapsto r' [] ()$$

$$\text{coop}(r :: rs) =$$

$$\text{return } () \quad \mapsto r \text{ rs } ()$$

$$\langle \text{yield } () \rightarrow r' \rangle \mapsto r (rs ++ [r']) ()$$

Example 3: cooperative concurrency (parameterised handler)

Types

Thread $E = 1 \rightarrow 1!(E \uplus \{\text{yield} : 1 \rightarrow 1\})$

Res $E = \text{List} (\text{Res } E) \rightarrow 1 \rightarrow 1!E$

Handler

$\text{coop} : \text{List} (\text{Res } E) \rightarrow 1!(E \uplus \{\text{yield} : 1 \rightarrow 1\}) \Rightarrow 1!E$

$\text{coop}([]) =$

$\text{return}() \mapsto ()$

$\langle \text{yield}() \rightarrow r' \rangle \mapsto r' [] ()$

$\text{coop}(r :: rs) =$

$\text{return}() \mapsto r rs ()$

$\langle \text{yield}() \rightarrow r' \rangle \mapsto r (rs ++ [r']) ()$

Helpers

$\text{coopWith} : \text{Thread } E \rightarrow \text{Res } E$

$\text{coopWith } t = \lambda rs. \lambda (). \text{handle } t () \text{ with coop } rs$

$\text{cooperate} : \text{List} (\text{Thread } E) \rightarrow 1!E$

$\text{cooperate } ts = \text{coopWith id} (\text{map coopWith } ts) ()$

Example 3: cooperative concurrency (parameterised handler)

Types

Thread $E = 1 \rightarrow 1!(E \uplus \{\text{yield} : 1 \rightarrow 1\})$

Res $E = \text{List} (\text{Res } E) \rightarrow 1 \rightarrow 1!E$

Handler

$\text{coop} : \text{List} (\text{Res } E) \rightarrow 1!(E \uplus \{\text{yield} : 1 \rightarrow 1\}) \Rightarrow 1!E$

$\text{coop} ([]) =$

$\text{return } () \mapsto ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r' [] ()$

$\text{coop} (r :: rs) =$

$\text{return } () \mapsto r rs ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r (rs ++ [r']) ()$

Helpers

$\text{coopWith} : \text{Thread } E \rightarrow \text{Res } E$

$\text{coopWith } t = \lambda rs. \lambda (). \text{handle } t () \text{ with } \text{coop } rs$

$\text{cooperate} : \text{List} (\text{Thread } E) \rightarrow 1!E$

$\text{cooperate } ts = \text{coopWith id } (\text{map } \text{coopWith } ts) ()$

$\text{cooperate } [tA, tB] \Longrightarrow ()$

A1 B1 A2 B2

Example 4: cooperative concurrency (shallow handler)

Types

Thread $E = 1 \rightarrow 1!(E \uplus \{\text{yield} : 1 \rightarrow 1\})$

Res $E = \text{Thread } E$

Handler

$\text{cooperate} : \text{List } (\text{Thread } E) \rightarrow 1!E$

$\text{cooperate } [] = ()$

$\text{cooperate } (t :: ts) =$

handle[†] $t()$ **with**

return $() \quad \mapsto \text{cooperate } (ts)$

$\langle \text{yield } () \rightarrow t \rangle \mapsto \text{cooperate } (ts \mathrel{++} [t])$

Example 4: cooperative concurrency (shallow handler)

Types

Thread $E = 1 \rightarrow 1!(E \uplus \{\text{yield} : 1 \rightarrow 1\})$

Res $E = \text{Thread } E$

Handler

$\text{cooperate} : \text{List } (\text{Thread } E) \rightarrow 1!E$

$\text{cooperate } [] = ()$

$\text{cooperate } (t :: ts) =$

handle[†] $t()$ **with**

return $() \mapsto \text{cooperate } (ts)$

$\langle \text{yield } () \rightarrow t \rangle \mapsto \text{cooperate } (ts \mathrel{++} [t])$

$\text{cooperate } [tA, tB] \Longrightarrow ()$

A1 B1 A2 B2

Operational semantics (shallow handlers)

Reduction rules

$$\begin{aligned}\text{let } x = V \text{ in } N &\rightsquigarrow N[V/x] \\ \text{handle}^\dagger V \text{ with } H &\rightsquigarrow N[V/x] \\ \text{handle}^\dagger \mathcal{E}[\text{op } V] \text{ with } H &\rightsquigarrow N_{\text{op}}[V/p, (\lambda x. \mathcal{E}[x])/r], \quad \text{op} \# \mathcal{E}\end{aligned}$$

$$\begin{aligned}\text{where } H = \text{return } x &\mapsto N \\ \langle \text{op}_1 p \rightarrow r \rangle &\mapsto N_{\text{op}_1} \\ &\dots \\ \langle \text{op}_k p \rightarrow r \rangle &\mapsto N_{\text{op}_k}\end{aligned}$$

Evaluation contexts

$$\mathcal{E} ::= [] \mid \text{let } x = \mathcal{E} \text{ in } N \mid \text{handle}^\dagger \mathcal{E} \text{ with } H$$

Typing rules (shallow handlers)

Effects

$$E ::= \emptyset \mid E \uplus \{\text{op} : A \twoheadrightarrow B\}$$

Computations

$$C, D ::= A!E$$

Operations

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \text{op } V : B!(E \uplus \{\text{op} : A \twoheadrightarrow B\})}$$

Handlers

$$\frac{\Gamma \vdash M : C \quad \Gamma \vdash H : C \Rightarrow^{\dagger} D}{\Gamma \vdash \mathbf{handle}^{\dagger} M \mathbf{with } H : D}$$

$$\frac{\Gamma, x : A \vdash N : D \quad [\text{op}_i : A_i \twoheadrightarrow B_i \in E]_i \quad [\Gamma, p : A_i, r : B_i \rightarrow A!E \vdash N_i : D]_i}{\Gamma \vdash \mathbf{return } x \mapsto N \quad (\langle \text{op}_i p \rightarrow r \rangle \mapsto N_i)_i : A!E \Rightarrow^{\dagger} D}$$

Example 5: cooperative concurrency with UNIX-style fork

Effect signature

$$\text{CoU } E = E \uplus \{\text{yield} : 1 \twoheadrightarrow 1, \text{ufork} : 1 \twoheadrightarrow \text{Bool}\}$$

Example 5: cooperative concurrency with UNIX-style fork

Effect signature

$$\text{CoU } E = E \uplus \{\text{yield} : 1 \rightarrow 1, \text{ufork} : 1 \rightarrow \text{Bool}\}$$

A single cooperative program

$\text{main} : 1 \rightarrow \text{CoU } E!1$

```
main () = print "M1 "; if ufork () then print "A1 "; yield (); print "A2 "  
        else print "M2 "; if ufork () then print "B1 "; yield (); print "B2 " else print "M3 "
```

Example 5: cooperative concurrency with UNIX-style fork

Types

Thread $E = 1 \rightarrow \text{CoU } E!1$

Res $E = \text{List (Res } E) \rightarrow 1 \rightarrow 1!E$

Parameterised handler

$\text{coop} : \text{List (Res } E) \rightarrow \text{CoU } E!1 \Rightarrow 1!E$

$\text{coop} ([]) =$

return () $\mapsto ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r' [] ()$

$\langle \text{ufork } () \rightarrow r' \rangle \mapsto r' [\lambda rs (). r' rs \text{ ff}]$
tt

$\text{coop} (r :: rs) =$

return () $\mapsto r rs ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r (rs ++ [r']) ()$

$\langle \text{ufork } () \rightarrow r' \rangle \mapsto r' (r :: rs ++ [\lambda rs (). r' rs \text{ ff}])$
tt

Example 5: cooperative concurrency with UNIX-style fork

Types

Thread $E = 1 \rightarrow \text{CoU } E!1$

Res $E = \text{List (Res } E) \rightarrow 1 \rightarrow 1!E$

Parameterised handler

$\text{coop} : \text{List (Res } E) \rightarrow \text{CoU } E!1 \Rightarrow 1!E$

$\text{coop} ([]) =$

return () \mapsto ()

$\langle \text{yield} () \rightarrow r' \rangle \mapsto r' [] ()$

$\langle \text{ufork} () \rightarrow r' \rangle \mapsto r' [\lambda rs ().r' rs \text{ ff}]$
tt

$\text{coop} (r :: rs) =$

return () $\mapsto r rs ()$

$\langle \text{yield} () \rightarrow r' \rangle \mapsto r (rs ++ [r']) ()$

$\langle \text{ufork} () \rightarrow r' \rangle \mapsto r' (r :: rs ++ [\lambda rs ().r' rs \text{ ff}])$
tt

$\text{cooperate [main]} \Rightarrow ()$

M1 A1 M2 B1 A2 M3 B2

Example 5: cooperative concurrency with UNIX-style fork

Types

Thread $E = 1 \rightarrow \text{CoU } E!1$

Res $E = \text{List (Res } E) \rightarrow 1 \rightarrow 1!E$

Parameterised handler

$\text{coop} : \text{List (Res } E) \rightarrow \text{CoU } E!1 \Rightarrow 1!E$

$\text{coop} ([]) =$

return () $\mapsto ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r' [] ()$

$\langle \text{ufork } () \rightarrow r' \rangle \mapsto r' [\lambda rs ().r' rs \text{ tt}]$
ff

$\text{coop} (r :: rs) =$

return () $\mapsto r rs ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r (rs ++ [r']) ()$

$\langle \text{ufork } () \rightarrow r' \rangle \mapsto r' (r :: rs ++ [\lambda rs ().r' rs \text{ tt}])$
ff

Example 5: cooperative concurrency with UNIX-style fork

Types

Thread $E = 1 \rightarrow \text{CoU } E!1$

Res $E = \text{List (Res } E) \rightarrow 1 \rightarrow 1!E$

Parameterised handler

$\text{coop} : \text{List (Res } E) \rightarrow \text{CoU } E!1 \Rightarrow 1!E$

$\text{coop} ([]) =$

return () $\mapsto ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r' [] ()$

$\langle \text{ufork } () \rightarrow r' \rangle \mapsto r' [\lambda rs ().r' rs \text{ tt}]$
ff

$\text{coop} (r :: rs) =$

return () $\mapsto r rs ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r (rs ++ [r']) ()$

$\langle \text{ufork } () \rightarrow r' \rangle \mapsto r' (r :: rs ++ [\lambda rs ().r' rs \text{ tt}])$
ff

$\text{cooperate [main]} \Rightarrow ()$

M1 M2 M3 A1 B1 A2 B2

Example 6: cooperative concurrency with higher-order fork

Effect signature — recursive effect signature

$$\text{Co } E = E \uplus \{\text{yield} : 1 \rightarrow 1, \text{ fork} : (1 \rightarrow 1! \text{Co } E) \rightarrow 1\}$$

Example 6: cooperative concurrency with higher-order fork

Effect signature — recursive effect signature

$$\text{Co } E = E \uplus \{\text{yield} : 1 \rightarrow 1, \text{ fork} : (1 \rightarrow 1! \text{Co } E) \rightarrow 1\}$$

A single cooperative program

`main : 1 → 1!Co E`

```
main () = print "M1 "; fork (λ().print "A1 "; yield (); print "A2 ");  
         print "M2 "; fork (λ().print "B1 "; yield (); print "B2 "); print "M3 "
```

Example 6: cooperative concurrency with higher-order fork

Types

Thread $E = 1 \rightarrow 1! \text{Co } E$

Res $E = \text{List } (\text{Res } E) \rightarrow 1 \rightarrow 1!E$

Parameterised handler

$\text{coop} : \text{List } (\text{Res } E) \rightarrow 1! \text{Co } E \Rightarrow 1!E$

$\text{coop } ([]) =$

return () $\mapsto ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r' [] ()$

$\langle \text{fork } t \rightarrow r' \rangle \mapsto \text{coopWith } t [r'] ()$

$\text{coop } (r :: rs) =$

return () $\mapsto r rs ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r (rs ++ [r']) ()$

$\langle \text{fork } t \rightarrow r' \rangle \mapsto \text{coopWith } t (r :: rs ++ [r']) ()$

Example 6: cooperative concurrency with higher-order fork

Types

Thread $E = 1 \rightarrow 1!Co\ E$

Res $E = List\ (Res\ E) \rightarrow 1 \rightarrow 1!E$

Parameterised handler

$coop : List\ (Res\ E) \rightarrow 1!Co\ E \Rightarrow 1!E$

$coop\ ([]) =$

return $() \quad \mapsto ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r' [] ()$

$\langle \text{fork } t \rightarrow r' \rangle \mapsto coopWith\ t\ [r']\ ()$

$coop\ (r :: rs) =$

return $() \quad \mapsto r\ rs\ ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r\ (rs ++ [r'])\ ()$

$\langle \text{fork } t \rightarrow r' \rangle \mapsto coopWith\ t\ (r :: rs ++ [r'])\ ()$

$cooperate\ [main] \implies ()$

M1 A1 M2 B1 A2 M3 B2

Example 6: cooperative concurrency with higher-order fork

Types

Thread $E = 1 \rightarrow 1!Co\ E$

Res $E = List\ (Res\ E) \rightarrow 1 \rightarrow 1!E$

Parameterised handler

$coop : List\ (Res\ E) \rightarrow 1!Co\ E \Rightarrow 1!E$

$coop\ ([]) =$

return $() \quad \mapsto ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r' [] ()$

$\langle \text{fork } t \rightarrow r' \rangle \mapsto r' [coopWith\ t] ()$

$coop\ (r :: rs) =$

return $() \quad \mapsto r\ rs\ ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r\ (rs\ ++\ [r'])\ ()$

$\langle \text{fork } t \rightarrow r' \rangle \mapsto r' (r :: rs\ ++\ [coopWith\ t])\ ()$

Example 6: cooperative concurrency with higher-order fork

Types

Thread $E = 1 \rightarrow 1! \text{Co } E$

Res $E = \text{List } (\text{Res } E) \rightarrow 1 \rightarrow 1!E$

Parameterised handler

$\text{coop} : \text{List } (\text{Res } E) \rightarrow 1! \text{Co } E \Rightarrow 1!E$

$\text{coop } ([]) =$

return () \mapsto ()

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r' [] ()$

$\langle \text{fork } t \rightarrow r' \rangle \mapsto r' [\text{coopWith } t] ()$

$\text{coop } (r :: rs) =$

return () $\mapsto r \text{ } rs ()$

$\langle \text{yield } () \rightarrow r' \rangle \mapsto r (rs ++ [r']) ()$

$\langle \text{fork } t \rightarrow r' \rangle \mapsto r' (r :: rs ++ [\text{coopWith } t]) ()$

$\text{cooperate } [\text{main}] \Longrightarrow ()$

M1 M2 M3 A1 B1 A2 B2

Example 7: pipes

Effect signatures

Sender = {`send` : $\text{Nat} \rightarrow \mathbb{1}$ }

Receiver = {`receive` : $\mathbb{1} \rightarrow \text{Nat}$ }

Example 7: pipes

Effect signatures

Sender = $\{\text{send} : \text{Nat} \rightarrow 1\}$

Receiver = $\{\text{receive} : 1 \rightarrow \text{Nat}\}$

A producer and a consumer

$\text{nats} : \text{Nat} \rightarrow 1!(E \uplus \text{Sender})$

$\text{nats } n = \text{send } n; \text{nats } (n + 1)$

$\text{grabANat} : 1 \rightarrow \text{Nat}!(E \uplus \text{Receiver})$

$\text{grabANat } () = \text{receive } ()$

Example 7: pipes

Effect signatures

Sender = {**send** : $\text{Nat} \rightarrow 1$ }

Receiver = {**receive** : $1 \rightarrow \text{Nat}$ }

A producer and a consumer

$\text{nats} : \text{Nat} \rightarrow 1!(E \uplus \text{Sender})$

$\text{nats } n = \text{send } n; \text{nats } (n + 1)$

$\text{grabANat} : 1 \rightarrow \text{Nat}!(E \uplus \text{Receiver})$

$\text{grabANat } () = \text{receive } ()$

Pipes and copipes as shallow handlers

$\text{pipe } p \ c = \text{handle}^\dagger c \ () \text{ with}$

$\text{return } x \quad \mapsto x$

$\langle \text{receive } () \rightarrow r \rangle \mapsto \text{copipe } r \ p$

$\text{copipe } c \ p = \text{handle}^\dagger p \ () \text{ with}$

$\text{return } x \quad \mapsto x$

$\langle \text{send } n \rightarrow r \rangle \mapsto \text{pipe } r \ (\lambda().c \ n)$

Example 7: pipes

Effect signatures

Sender = {**send** : $\text{Nat} \rightarrow 1$ }

Receiver = {**receive** : $1 \rightarrow \text{Nat}$ }

A producer and a consumer

$\text{nats} : \text{Nat} \rightarrow 1!(E \uplus \text{Sender})$

$\text{nats } n = \text{send } n; \text{nats } (n + 1)$

$\text{grabANat} : 1 \rightarrow \text{Nat}!(E \uplus \text{Receiver})$

$\text{grabANat } () = \text{receive } ()$

Pipes and copipes as shallow handlers

$\text{pipe } p \ c = \text{handle}^\dagger c \ () \text{ with}$

$\text{return } x \quad \mapsto x$

$\langle \text{receive } () \rightarrow r \rangle \mapsto \text{copipe } r \ p$

$\text{copipe } c \ p = \text{handle}^\dagger p \ () \text{ with}$

$\text{return } x \quad \mapsto x$

$\langle \text{send } n \rightarrow r \rangle \mapsto \text{pipe } r \ (\lambda().c \ n)$

$\text{pipe } (\lambda().\text{nats } 0) \ \text{grabANat} \rightsquigarrow^+ \text{copipe } (\lambda x.x) \ (\lambda().\text{nats } 0)$

$\rightsquigarrow^+ \text{pipe } (\lambda().\text{nats } 1) \ (\lambda().0) \rightsquigarrow^+ 0$

Example 7: pipes

Effect signatures

Sender = {**send** : $\text{Nat} \rightarrow 1$ }

Receiver = {**receive** : $1 \rightarrow \text{Nat}$ }

A producer and a consumer

$\text{nats} : \text{Nat} \rightarrow 1!(E \uplus \text{Sender})$

$\text{nats } n = \text{send } n; \text{nats } (n + 1)$

$\text{grabANat} : 1 \rightarrow \text{Nat}!(E \uplus \text{Receiver})$

$\text{grabANat } () = \text{receive } ()$

Pipes and copipes as shallow handlers

$\text{pipe } p \ c = \text{handle}^\dagger c \ () \text{ with}$

$\text{return } x \mapsto x$

$\langle \text{receive } () \rightarrow r \rangle \mapsto \text{copipe } r \ p$

$\text{copipe } c \ p = \text{handle}^\dagger p \ () \text{ with}$

$\text{return } x \mapsto x$

$\langle \text{send } n \rightarrow r \rangle \mapsto \text{pipe } r \ (\lambda().c \ n)$

$\text{pipe } (\lambda().\text{nats } 0) \ \text{grabANat} \rightsquigarrow^+ \text{copipe } (\lambda x.x) \ (\lambda().\text{nats } 0)$

$\rightsquigarrow^+ \text{pipe } (\lambda().\text{nats } 1) \ (\lambda().0) \rightsquigarrow^+ 0$

Exercise: implement pipes using parameterised handlers

Built-in effects

Console I/O

Console = {
 inch : 1 \rightarrow char
 ouch : char \rightarrow 1}

print s = map (λc . **ouch** c) s; ()

Generative state

GenState = {
 new : $a. a \rightarrow \text{Ref } a$,
 write : $a. (\text{Ref } a \times a) \rightarrow 1$,
 read : $a. \text{Ref } a \rightarrow a$ }

Example 8: actors

Process ids

$\text{Pid } a = \text{Ref}(\text{List } a)$

Effect signature

$\text{Actor } a = \{$
 $\text{self} \quad : \quad 1 \twoheadrightarrow \text{Pid } a,$
 $\text{spawn} : b. (1 \rightarrow 1! \text{Actor } b) \twoheadrightarrow \text{Pid } b,$
 $\text{send} \quad : b. \quad (b \times \text{Pid } b) \twoheadrightarrow 1,$
 $\text{recv} \quad : \quad 1 \twoheadrightarrow a \}$

Example 8: actors

Process ids

$\text{Pid } a = \text{Ref}(\text{List } a)$

Effect signature

$\text{Actor } a = \{$
 $\text{self} \quad : \quad 1 \twoheadrightarrow \text{Pid } a,$
 $\text{spawn} : b. (1 \rightarrow 1! \text{Actor } b) \twoheadrightarrow \text{Pid } b,$
 $\text{send} \quad : b. \quad (b \times \text{Pid } b) \twoheadrightarrow 1,$
 $\text{recv} \quad : \quad 1 \twoheadrightarrow a \}$

An actor chain

$\text{spawnMany} : \text{Pid String} \rightarrow \text{Int} \rightarrow 1!(E \uplus \text{Actor String})$

$\text{spawnMany } p \ 0 = \text{send}(\text{"ping!"}, p)$

$\text{spawnMany } p \ n = \text{spawnMany}(\text{spawn}(\lambda(). \text{let } s = \text{recv}() \text{ in print "."; send}(s, p))) \ (n - 1)$

$\text{chain} : \text{Int} \rightarrow 1!(E \uplus \text{Actor String} \uplus \text{Console})$

$\text{chain } n = \text{spawnMany}(\text{self}()) \ n; \text{let } s = \text{recv}() \text{ in print } s$

Example 8: actors — via cooperative concurrency

$\text{act} : \text{Pid } a \rightarrow 1!(E \uplus \text{Actor } a) \Rightarrow 1!\text{Co } (E \uplus \text{GenState})$

$\text{act } \text{mine} =$

```
return ()            $\mapsto$  ()  
 $\langle \text{self } () \rightarrow r \rangle$   $\mapsto$   $r \text{ mine mine}$   
 $\langle \text{spawn } \text{you} \rightarrow r \rangle$   $\mapsto$  let  $\text{yours} = \text{new } []$  in  
                         $\text{fork } (\lambda(). \text{act } \text{yours } (\text{you } ()))$ ;  $r \text{ mine yours}$   
 $\langle \text{send } (m, \text{yours}) \rightarrow r \rangle$   $\mapsto$  let  $ms = \text{read } \text{yours}$  in  
                         $\text{write } (\text{yours}, ms ++ [m])$ ;  $r \text{ mine } ()$   
 $\langle \text{recv } () \rightarrow r \rangle$   $\mapsto$  letrec  $\text{recvWhenReady } () =$   
                        case  $\text{read } \text{mine}$  of  
                             $[]$   $\mapsto$   $\text{yield } ()$ ;  $\text{recvWhenReady } ()$   
                             $(m :: ms)$   $\mapsto$   $\text{write } (\text{mine}, ms)$ ;  $r \text{ mine } m$   
                        in  $\text{recvWhenReady } ()$ 
```


Example 8: actors — via cooperative concurrency

```
cooperate [handle chain 64 with act (new [])]  $\Rightarrow$  ()  
.....ping!
```

Example 8: actors — via cooperative concurrency

cooperate [**handle** chain 64 **with** act (new [])] \implies ()
.....ping!

Exercise: Compare three different implementations of actors:

- ▶ the one we've just seen (factored through a parameterised handler for cooperative concurrency)
- ▶ the same thing, but using shallow handlers
- ▶ a single monolithic handler using parameterised handlers

Deep effect handlers

$$\frac{\Gamma, x : A \vdash N : D \quad [\text{op}_i : A_i \twoheadrightarrow B_i \in E]_i \quad [\Gamma, p : A_i, r : B_i \rightarrow D \vdash N_i : D]_i}{\Gamma \vdash \text{return } x \mapsto N \quad (\langle \text{op}_i \ p \rightarrow r \rangle \mapsto N_i)_i : A!E \Rightarrow D}$$

handle $\mathcal{E}[\text{op } V]$ **with** $H \rightsquigarrow N_{\text{op}}[V/p, (\lambda x. \text{handle } \mathcal{E}[x] \text{ with } H)/r], \quad \text{op} \# \mathcal{E}$

Deep effect handlers

$$\frac{\Gamma, x : A \vdash N : D \quad [\text{op}_i : A_i \twoheadrightarrow B_i \in E]_i \quad [\Gamma, p : A_i, r : B_i \rightarrow D \vdash N_i : D]_i}{\Gamma \vdash \text{return } x \mapsto N \quad (\langle \text{op}_i \ p \rightarrow r \rangle \mapsto N_i)_i : A!E \Rightarrow D}$$

handle $\mathcal{E}[\text{op } V]$ **with** $H \rightsquigarrow N_{\text{op}}[V/p, (\lambda x. \text{handle } \mathcal{E}[x] \text{ with } H)/r], \quad \text{op} \notin \mathcal{E}$

The body of the resumption r reinvokes the handler

Deep effect handlers

$$\frac{\Gamma, x : A \vdash N : D \quad [\text{op}_i : A_i \multimap B_i \in E]_i \quad [\Gamma, p : A_i, r : B_i \rightarrow D \vdash N_i : D]_i}{\Gamma \vdash \text{return } x \mapsto N \quad (\langle \text{op}_i \ p \rightarrow r \rangle \mapsto N_i)_i : A!E \Rightarrow D}$$

handle $\mathcal{E}[\text{op } V]$ **with** $H \rightsquigarrow N_{\text{op}}[V/p, (\lambda x. \text{handle } \mathcal{E}[x] \text{ with } H)/r], \quad \text{op} \notin \mathcal{E}$

The body of the resumption r reinvokes the handler

A deep handler performs a fold (catamorphism) on a computation tree

Shallow effect handlers

$$\frac{\Gamma, x : A \vdash N : D \quad [\text{op}_i : A_i \twoheadrightarrow B_i \in E]_i \quad [\Gamma, p : A_i, r : B_i \rightarrow A!E \vdash N_i : D]_i}{\Gamma \vdash \text{return } x \mapsto N \quad (\langle \text{op}_i p \rightarrow r \rangle \mapsto N_i)_i : A!E \Rightarrow^\dagger D}$$

$$\text{handle}^\dagger \mathcal{E}[\text{op } V] \text{ with } H \rightsquigarrow N_{\text{op}}[V/p, (\lambda x. \mathcal{E}[x])/r], \quad \text{op} \# \mathcal{E}$$

Shallow effect handlers

$$\frac{\Gamma, x : A \vdash N : D \quad [\text{op}_i : A_i \twoheadrightarrow B_i \in E]_i \quad [\Gamma, p : A_i, r : B_i \rightarrow A!E \vdash N_i : D]_i}{\Gamma \vdash \text{return } x \mapsto N \quad (\langle \text{op}_i p \rightarrow r \rangle \mapsto N_i)_i : A!E \Rightarrow^\dagger D}$$

$$\text{handle}^\dagger \mathcal{E}[\text{op } V] \text{ with } H \rightsquigarrow N_{\text{op}}[V/p, (\lambda x. \mathcal{E}[x])/r], \quad \text{op} \# \mathcal{E}$$

The body of the resumption r does not reinvoke the handler

Shallow effect handlers

$$\frac{\Gamma, x : A \vdash N : D \quad [\text{op}_i : A_i \twoheadrightarrow B_i \in E]_i \quad [\Gamma, p : A_i, r : B_i \rightarrow A!E \vdash N_i : D]_i}{\Gamma \vdash \text{return } x \mapsto N \quad (\langle \text{op}_i p \rightarrow r \rangle \mapsto N_i)_i : A!E \Rightarrow^\dagger D}$$

$$\text{handle}^\dagger \mathcal{E}[\text{op } V] \text{ with } H \rightsquigarrow N_{\text{op}}[V/p, (\lambda x. \mathcal{E}[x])/r], \quad \text{op} \# \mathcal{E}$$

The body of the resumption r does not reinvoke the handler

A shallow handler performs a case-split on a computation tree

Deep vs shallow

Choice $E = E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}\}$

Always choose true

Deep

$\text{trueChoice} : (1 \rightarrow A! \text{Choice } E) \rightarrow A!E$
 $\text{trueChoice } m = \mathbf{handle } m () \mathbf{with}$
 $\mathbf{return } x \quad \mapsto x$
 $\text{choose } () \ r \mapsto r \ \text{true}$

Shallow

$\text{trueChoice}^\dagger : (1 \rightarrow A! \text{Choice } E) \rightarrow A!E$
 $\text{trueChoice}^\dagger m = \mathbf{handle}^\dagger m () \mathbf{with}$
 $\mathbf{return } x \quad \mapsto x$
 $\text{choose } () \ r \mapsto \text{trueChoice}^\dagger (\lambda().r \ \text{true})$

Deep vs shallow

Choice $E = E \uplus \{\text{choose} : 1 \rightarrow \text{Bool}\}$

Choose true and false

Deep

$\text{allChoices} : (1 \rightarrow A! \text{Choice } E) \rightarrow \text{List } A!E$

$\text{allChoices } m = \mathbf{handle} \ m() \ \mathbf{with}$

$\mathbf{return} \ x \mapsto [x]$

$\text{choose}() \ r \mapsto$

$r \text{ true } ++ r \text{ false}$

Shallow

$\text{allChoices}^\dagger : (1 \rightarrow A! \text{Choice } E) \rightarrow \text{List } A!E$

$\text{allChoices}^\dagger \ m = \mathbf{handle}^\dagger \ m() \ \mathbf{with}$

$\mathbf{return} \ x \mapsto [x]$

$\text{choose}() \ r \mapsto$

$\text{allChoices}^\dagger (\lambda().r \text{ true}) ++$

$\text{allChoices}^\dagger (\lambda().r \text{ false})$

Deep vs shallow

$$\text{Reader } S \ E = E \uplus \{\text{get} : 1 \rightarrow S\}$$

Read-only state

Deep

$\text{reader} : (1 \rightarrow A! \text{Reader } S \ E) \rightarrow S \rightarrow A!E$

$\text{reader } m = \text{handle } m () \text{ with}$

$\text{return } x \mapsto \lambda s. x$

$\text{get } () \ r \mapsto \lambda s. r \ s \ s$

Shallow

$\text{reader}^\dagger : (1 \rightarrow A! \text{Reader } S \ E) \rightarrow S \rightarrow A!E$

$\text{reader}^\dagger m \ s = \text{handle}^\dagger m () \text{ with}$

$\text{return } x \mapsto x$

$\text{get } () \ r \mapsto \text{reader}^\dagger (\lambda(). r \ s) \ s$

Deep vs shallow

Deep

- ▶ can be more concise
- ▶ simpler to implement efficiently and without memory leaks

Shallow

- ▶ convenient for parameterisation
- ▶ convenient for implementing structural recursion schemes other than catamorphisms (e.g. **pipes**)

Deep as shallow

$$\mathcal{S}[\text{handle } M \text{ with } H] = \text{letrec } h \text{ } f = \text{handle}^\dagger f () \text{ with } \mathcal{S}[H] h \text{ in} \\ h (\lambda(). \mathcal{S}[M])$$

$$\mathcal{S}[H] h = \mathcal{S}[H^{\text{ret}}] h \uplus \mathcal{S}[H^{\text{ops}}] h$$

$$\mathcal{S}[\{\text{return } x \mapsto N\}] h = \{\text{return } x \mapsto \mathcal{S}[N]\}$$

$$\mathcal{S}[\{\text{op}_i \text{ } p \text{ } r \mapsto N_i\}_i] h = \{\text{op}_i \text{ } p \text{ } r \mapsto \text{let } r = \text{return } \lambda x. h (\lambda(). r x) \text{ in } \mathcal{S}[N_i]\}_i$$

Exercise: prove an operational correspondence result for $\mathcal{S}[-]$

Shallow as deep

$$\mathcal{D}[\![C \Rightarrow^{\dagger} D]\!] = \mathcal{D}[\![C]\!] \Rightarrow (1 \rightarrow \mathcal{D}[\![C]\!], 1 \rightarrow \mathcal{D}[\![D]\!])$$

Shallow as deep

$$\mathcal{D}[\![C \Rightarrow^\dagger D]\!] = \mathcal{D}[\![C]\!] \Rightarrow (1 \rightarrow \mathcal{D}[\![C]\!], 1 \rightarrow \mathcal{D}[\![D]\!])$$

$$\mathcal{D}[\![\text{handle}^\dagger M \text{ with } H]\!] = \text{let } z = \text{handle } \mathcal{D}[\![M]\!] \text{ with } \mathcal{D}[\![H]\!] \text{ in} \\ \text{let } (f, g) = z \text{ in } g ()$$

$$\mathcal{D}[\![H]\!] = \mathcal{D}[\![H^{\text{ret}}]\!] \uplus \mathcal{D}[\![H^{\text{ops}}]\!]$$

$$\mathcal{D}[\![\{\text{return } x \mapsto N\}]\!] = \{\text{return } x \mapsto \text{return } (\lambda().\text{return } x, \lambda().\mathcal{D}[\![N]\!])\}$$

$$\mathcal{D}[\![\{\text{op}_i p r \mapsto N\}_i]\!] = \{\text{op}_i p r \mapsto \\ \text{let } r = \lambda x. \text{let } z = r x \text{ in let } (f, g) = z \text{ in } f () \text{ in} \\ \text{return } (\lambda(). \text{let } x = \text{op}_i p \text{ in } r x, \lambda().\mathcal{D}[\![N]\!])\}_i$$

Exercise: prove an operational correspondence result for $\mathcal{D}[\![-]\!]$

(the result is weaker and requires more sophisticated techniques than for $\mathcal{S}[\![-]\!]$)

Sheep effect handlers — a hybrid of shallow and deep handlers

$$\frac{\begin{array}{c} \Gamma, x : A \vdash N : D \\ [\text{op}_i : A_i \rightarrow B_i \in E]_i \quad [\Gamma, p : A_i, r : (A!E \Rightarrow D) \rightarrow B_i \rightarrow D \vdash N_i : D]_i \end{array}}{\Gamma \vdash \text{return } x \mapsto N \quad (\text{op}_i \ p \ r \mapsto N_i)_i : A!E \Rightarrow D}$$

handle $\mathcal{E}[\text{op } V]$ **with** $H \rightsquigarrow N_{\text{op}}[V/p, (\lambda h x. \text{handle } \mathcal{E}[x] \text{ with } h)/r], \quad \text{op} \# \mathcal{E}$

Like a shallow handler, the body of the resumption need not reinvoke the same handler

Like a deep handler, the body of the resumption must invoke *some* handler

Example 9: effect pollution

Effect signatures

Reader = $\{\text{get} : 1 \twoheadrightarrow \text{Nat}\}$

Failure = $\{\text{fail} : a.1 \twoheadrightarrow a\}$

Example 9: effect pollution

Effect signatures

Reader = {**get** : $1 \twoheadrightarrow \text{Nat}$ }

Failure = {**fail** : $a.1 \twoheadrightarrow a$ }

Handlers

$\text{reads} : \text{List Nat} \rightarrow A!(E \uplus \text{Reader}) \Rightarrow A!E$

$\text{reads}([]) = \text{return } x \quad \mapsto x$
 $\langle \text{get}() \rightarrow r \rangle \mapsto \text{fail}()$

$\text{reads}(n :: ns) = \text{return } x \quad \mapsto x$
 $\langle \text{get}() \rightarrow r \rangle \mapsto r \text{ ns } n$

$\text{maybeFail} : A!(E \uplus \text{Failure}) \Rightarrow \text{Maybe } A!E$

$\text{maybeFail} = \text{return } x \quad \mapsto \text{Just } x$
 $\langle \text{fail}() \rightarrow r \rangle \mapsto \text{Nothing}$

Example 9: effect pollution

$\text{bad} : \text{List Nat} \rightarrow (1 \rightarrow A!(E \uplus \text{Reader} \uplus \text{Failure})) \rightarrow \text{Maybe } A!E$
 $\text{bad } ns\ t = \mathbf{handle}\ (\mathbf{handle}\ t\ ())\ \mathbf{with}\ \text{reads } ns)\ \mathbf{with}\ \text{maybeFail}$

Example 9: effect pollution

$\text{bad} : \text{List Nat} \rightarrow (1 \rightarrow A!(E \uplus \text{Reader} \uplus \text{Failure})) \rightarrow \text{Maybe } A!E$

$\text{bad } ns\ t = \mathbf{handle}\ (\mathbf{handle}\ t\ ()\ \mathbf{with}\ \text{reads } ns)\ \mathbf{with}\ \text{maybeFail}$

$\text{bad } [1, 2]\ (\lambda().\text{get } () + \text{fail } ()) : \text{Maybe Nat} \Longrightarrow \text{Nothing}$

Example 9: effect pollution

$\text{bad} : \text{List Nat} \rightarrow (1 \rightarrow A!(E \uplus \text{Reader} \uplus \text{Failure})) \rightarrow \text{Maybe } A!E$
 $\text{bad } ns \ t = \mathbf{handle} \ (\mathbf{handle} \ t \ ()) \ \mathbf{with} \ \text{reads } ns) \ \mathbf{with} \ \text{maybeFail}$

$\text{bad } [1, 2] \ (\lambda().\text{get}() + \text{fail}()) : \text{Maybe Nat} \Longrightarrow \text{Nothing}$

Exercise: Design a mechanism to allow the Failure effect to be encapsulated.

(The aim is to define

$\text{good} : \text{List } A \rightarrow (1 \rightarrow \text{Nat}!(E \uplus \text{Reader})) \rightarrow \text{Maybe } A!E$

by composing reads and maybeFail such that

$\text{good } [1, 2] \ (\lambda().\text{get}() + \text{fail}()) : \text{Maybe Nat!Failure}$

performs the fail operation.)