

Algebraic effects and effect handlers

Sam Lindley

The University of Edinburgh

OPLSS 2022

Quiz

Quiz

What is an effect?

Quiz

What is an effect?

What is a pure computation?

Quiz

What is an effect?

What is a pure computation?

What is an effectful computation?

Effects

Programs as black boxes (Church-Turing model)?



Effects

Programs must interact with their environment



Effects

Programs must interact with their environment



Effects

Programs must interact with their environment



Effects are pervasive

- ▶ input/output
 user interaction
- ▶ concurrency
 web applications
- ▶ distribution
 cloud computing
- ▶ exceptions
 fault tolerance
- ▶ choice
 backtracking search

Typically ad hoc and hard-wired

Effect handlers



Gordon Plotkin



Matija Pretnar

Handlers of algebraic effects, ESOP 2009

Effect handlers



Gordon Plotkin



Matija Pretnar

Handlers of algebraic effects, ESOP 2009

Composable and **customisable** user-defined interpretation of effects in general

Effect handlers



Gordon Plotkin



Matija Pretnar

Handlers of algebraic effects, ESOP 2009

Composable and **customisable** user-defined interpretation of effects in general

Give programmer direct access to **context**

(c.f. resumable exceptions, monads, delimited control)

Effect handlers



Gordon Plotkin



Matija Pretnar

Handlers of algebraic effects, ESOP 2009

Composable and **customisable** user-defined interpretation of effects in general

Give programmer direct access to **context**

Growing industrial interest (c.f. resumable exceptions, monads, delimited control)

GitHub

`semantic`

Code analysis library (> 25 million repositories)




React

JavaScript UI library (> 2 million websites)



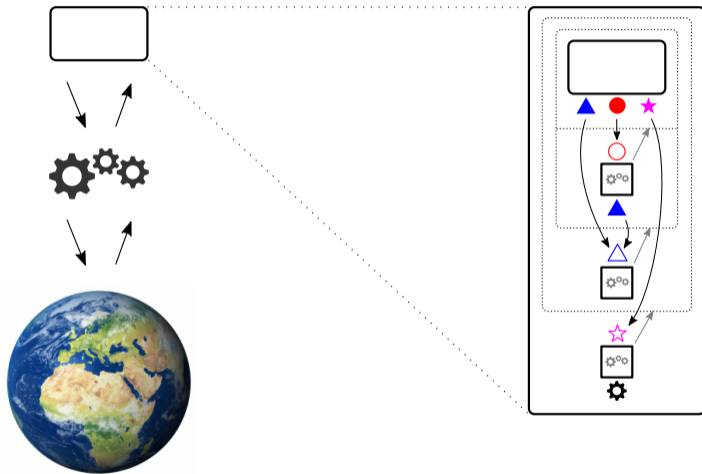

Pyro

Statistical inference (10% ad spend saving)

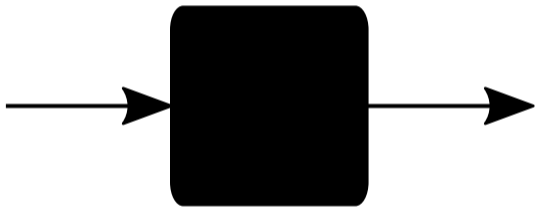
Effect handlers as composable user-defined operating systems



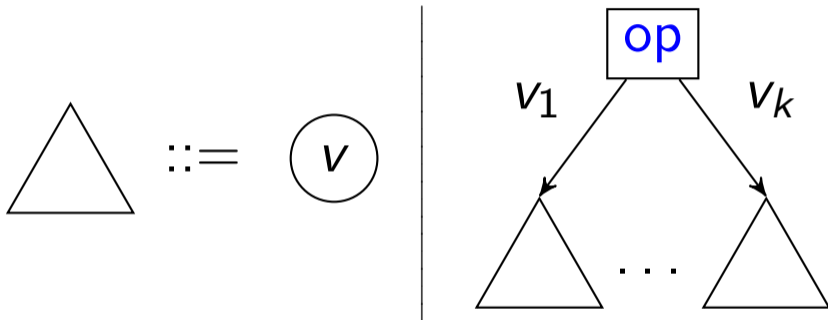
Effect handlers as composable user-defined operating systems



Pure computations

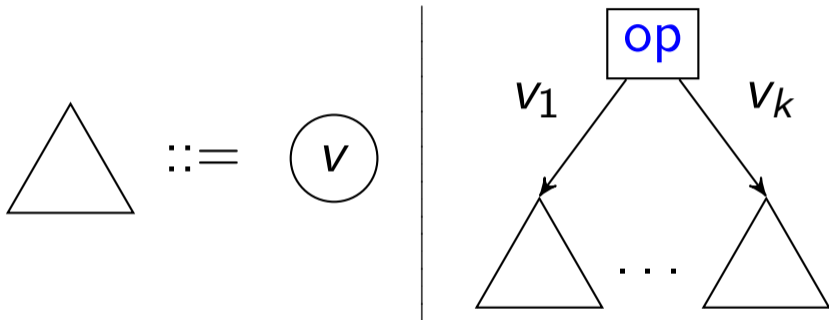


Effectful computations



A **command-response** tree (aka **interaction tree**)

Effectful computations



A **command-response** tree (aka **interaction tree**)

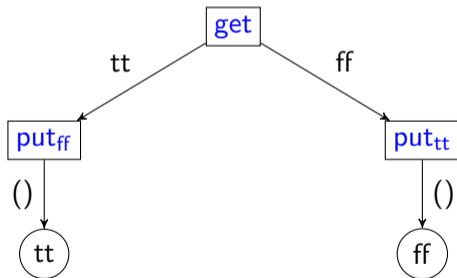
Effectful computation is all about **interaction** with some **context**

Example: boolean state (bit toggling)

get : Bool

put_{tt} : 1

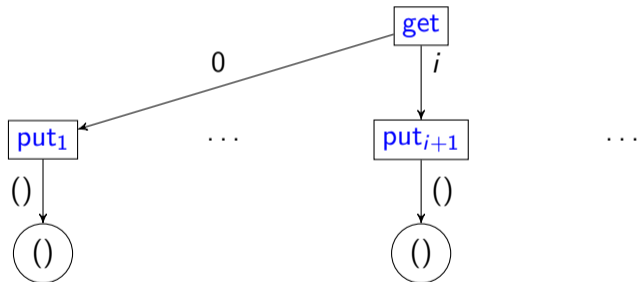
put_{ff} : 1



Example: natural number state (increment)

get : \mathbb{N}

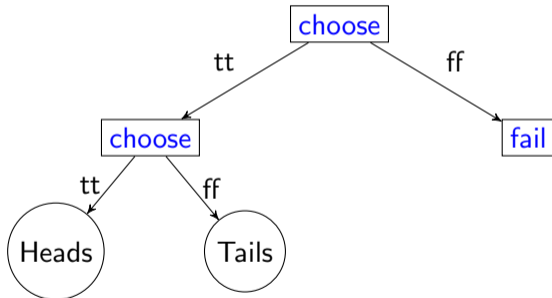
put_{*i*} : 1, $i \in \mathbb{N}$



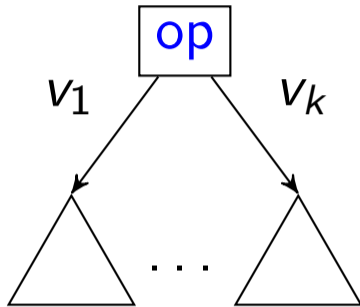
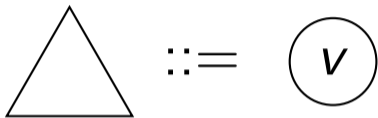
Example: nondeterminism (drunk coin toss)

choose : Bool

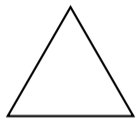
fail : 0



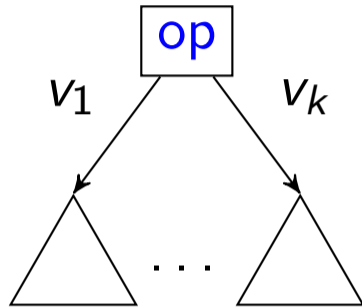
What is an effectful computation?



What is an effectful computation?



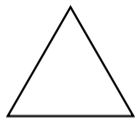
$::=$



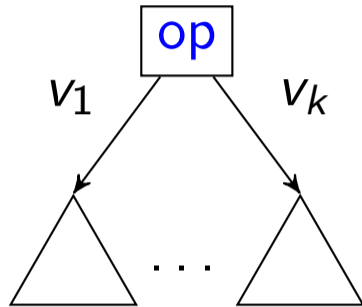
Equivalently (ignoring edge labels)

$m ::= \mathbf{return} \ v \mid \mathbf{op} \ \langle m_1, \dots, m_k \rangle$

What is an effectful computation?



$::=$



Equivalently (ignoring edge labels)

$m ::= \mathbf{return} \ v \mid \mathbf{op} \ \langle m_1, \dots, m_k \rangle$

Equivalently (accounting for edge labels)

$m ::= \mathbf{return} \ v \mid \mathbf{op} \ (\lambda x. \mathbf{case} \ x \{ v_1 \mapsto m_1; \dots; v_k \mapsto m_k \})$

Examples

Boolean state

```
toggle = get ⟨putff ⟨return tt⟩, puttt ⟨return ff⟩⟩
```

```
let s = get () in put (not s); s
```

Examples

Boolean state

toggle = `get` \langle `putff` \langle `return tt` \rangle , `puttt` \langle `return ff` \rangle \rangle

`let s = get () in put (not s); s`

Natural number state

increment = `get` \langle `put1` \langle `return ()` \rangle , ..., `puti+1` \langle `return ()` \rangle , ... \rangle

`put (1 + get ())`

Examples

Boolean state

toggle = `get` \langle `putff` \langle `return tt` \rangle , `puttt` \langle `return ff` \rangle \rangle

`let s = get () in put (not s); s`

Natural number state

increment = `get` \langle `put1` \langle `return ()` \rangle , \dots , `puti+1` \langle `return ()` \rangle , \dots \rangle

`put (1 + get ())`

Nondeterminism

drunkToss = `choose` \langle `choose` \langle `return Heads`, `return Tails` \rangle , `fail` \langle \rangle \rangle

`if choose () then (if choose () then Heads else Tails) else absurd (fail ())`

Command-response trees as free monads

- ▶ A computation of type `comp A` is a tree whose leaves have type `A`
- ▶ Return is **return**
- ▶ Bind performs substitution at the leaves

$$\begin{aligned} \mathbf{return} \ v \gg\gg r &= r \ v \\ \mathbf{op} \ \langle m_1, \dots, m_n \rangle \gg\gg r &= \mathbf{op} \ \langle m_1 \gg\gg r, \dots, m_n \gg\gg r \rangle \end{aligned}$$

Algebraic effects

An algebraic effect is given by

1. a **signature** of operations
2. a collection of **equations**

Algebraic effects

An algebraic effect is given by

1. a **signature** of operations
2. a collection of **equations**

Example: boolean state

Signature

`get` : Bool

`puttt` : 1

`putff` : 1

Algebraic effects

An algebraic effect is given by

1. a **signature** of operations
2. a collection of **equations**

Example: boolean state

Signature

`get` : Bool

`puttt` : 1

`putff` : 1

Equations

$$\text{put}_s \langle \text{put}_{s'} \langle m \rangle \rangle \simeq \text{put}_{s'} \langle m \rangle \quad (\text{put-put})$$

$$\text{put}_s \langle \text{get} \langle m_{tt}, m_{ff} \rangle \rangle \simeq \text{put}_s \langle m_s \rangle \quad (\text{put-get})$$

$$\text{get} \langle \text{put}_{tt} \langle m \rangle, \text{put}_{ff} \langle m \rangle \rangle \simeq m \quad (\text{get-put})$$

$$\text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \simeq \text{get} \langle m, n \rangle \quad (\text{get-get})$$

Aside: the (get-get) equation is redundant

$$\text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle$$

Aside: the (get-get) equation is redundant

$$\begin{aligned} & \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \\ \simeq & \quad (\text{get-put}) \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle \rangle \end{aligned}$$

Aside: the (get-get) equation is redundant

$$\begin{aligned} & \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \\ \simeq & \quad (\text{get-put}) \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle \rangle \\ \simeq & \quad (\text{put-get}) \times 2 \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle m, m' \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle n', n \rangle \rangle \rangle \end{aligned}$$

Aside: the (get-get) equation is redundant

$$\begin{aligned} & \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \\ \approx & \text{(get-put)} \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle \rangle \\ \approx & \text{(put-get)} \times 2 \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle m, m' \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle n', n \rangle \rangle \rangle \\ \approx & \text{(put-get)} \times 2 \\ & \text{get} \langle \text{put}_{\text{tt}} \langle m \rangle, \text{put}_{\text{ff}} \langle n \rangle \rangle \end{aligned}$$

Aside: the (get-get) equation is redundant

$$\begin{aligned} & \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \\ \approx & \text{(get-put)} \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle \rangle \\ \approx & \text{(put-get)} \times 2 \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle m, m' \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle n', n \rangle \rangle \rangle \\ \approx & \text{(put-get)} \times 2 \\ & \text{get} \langle \text{put}_{\text{tt}} \langle m \rangle, \text{put}_{\text{ff}} \langle n \rangle \rangle \\ \approx & \text{(put-get)} \times 2 \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle m, n \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle m, n \rangle \rangle \rangle \end{aligned}$$

Aside: the (get-get) equation is redundant

$$\begin{aligned} & \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \\ \approx & \text{(get-put)} \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle \text{get} \langle m, m' \rangle, \text{get} \langle n', n \rangle \rangle \rangle \rangle \\ \approx & \text{(put-get)} \times 2 \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle m, m' \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle n', n \rangle \rangle \rangle \\ \approx & \text{(put-get)} \times 2 \\ & \text{get} \langle \text{put}_{\text{tt}} \langle m \rangle, \text{put}_{\text{ff}} \langle n \rangle \rangle \\ \approx & \text{(put-get)} \times 2 \\ & \text{get} \langle \text{put}_{\text{tt}} \langle \text{get} \langle m, n \rangle \rangle, \text{put}_{\text{ff}} \langle \text{get} \langle m, n \rangle \rangle \rangle \\ \approx & \text{(get-put)} \\ & \text{get} \langle m, n \rangle \end{aligned}$$

Interpreting algebraic effects

Example: boolean state

Standard interpretation ($\llbracket \text{comp } A \rrbracket = \text{Bool} \rightarrow \llbracket A \rrbracket \times \text{Bool}$)

$$\begin{aligned}\llbracket \text{return } v \rrbracket &= \lambda s. (\llbracket v \rrbracket, s) \\ \llbracket \text{get } \langle m, n \rangle \rrbracket &= \lambda s. \text{if } s \text{ then } \llbracket m \rrbracket s \text{ else } \llbracket n \rrbracket s \\ \llbracket \text{put}_{s'} \langle m \rangle \rrbracket &= \lambda s. \llbracket m \rrbracket s'\end{aligned}$$

Interpreting algebraic effects

Example: boolean state

Standard interpretation ($\llbracket \text{comp } A \rrbracket = \text{Bool} \rightarrow \llbracket A \rrbracket \times \text{Bool}$)

$$\begin{aligned}\llbracket \text{return } v \rrbracket &= \lambda s. (\llbracket v \rrbracket, s) \\ \llbracket \text{get } \langle m, n \rangle \rrbracket &= \lambda s. \text{if } s \text{ then } \llbracket m \rrbracket s \text{ else } \llbracket n \rrbracket s \\ \llbracket \text{put}_{s'} \langle m \rangle \rrbracket &= \lambda s. \llbracket m \rrbracket s'\end{aligned}$$

Discard interpretation ($\llbracket \text{comp } A \rrbracket = \text{Bool} \rightarrow \llbracket A \rrbracket$)

$$\begin{aligned}\llbracket \text{return } v \rrbracket &= \lambda s. \llbracket v \rrbracket \\ \llbracket \text{get } \langle m, n \rangle \rrbracket &= \lambda s. \text{if } s \text{ then } \llbracket m \rrbracket s \text{ else } \llbracket n \rrbracket s \\ \llbracket \text{put}_{s'} \langle m \rangle \rrbracket &= \lambda s. \llbracket m \rrbracket s'\end{aligned}$$

Interpreting algebraic effects

Example: boolean state

Standard interpretation ($\llbracket \text{comp } A \rrbracket = \text{Bool} \rightarrow \llbracket A \rrbracket \times \text{Bool}$)

$$\begin{aligned}\llbracket \text{return } v \rrbracket &= \lambda s. (\llbracket v \rrbracket, s) \\ \llbracket \text{get } \langle m, n \rangle \rrbracket &= \lambda s. \text{if } s \text{ then } \llbracket m \rrbracket s \text{ else } \llbracket n \rrbracket s \\ \llbracket \text{put}_{s'} \langle m \rangle \rrbracket &= \lambda s. \llbracket m \rrbracket s'\end{aligned}$$

Discard interpretation ($\llbracket \text{comp } A \rrbracket = \text{Bool} \rightarrow \llbracket A \rrbracket$)

$$\begin{aligned}\llbracket \text{return } v \rrbracket &= \lambda s. \llbracket v \rrbracket \\ \llbracket \text{get } \langle m, n \rangle \rrbracket &= \lambda s. \text{if } s \text{ then } \llbracket m \rrbracket s \text{ else } \llbracket n \rrbracket s \\ \llbracket \text{put}_{s'} \langle m \rangle \rrbracket &= \lambda s. \llbracket m \rrbracket s'\end{aligned}$$

Logging interpretation ($\llbracket \text{comp } A \rrbracket = \text{Bool} \rightarrow \llbracket A \rrbracket \times \text{List Bool}$)

$$\begin{aligned}\llbracket \text{return } v \rrbracket &= \lambda s. (\llbracket v \rrbracket, [s]) \\ \llbracket \text{get } \langle m, n \rangle \rrbracket &= \lambda s. \text{if } s \text{ then } \llbracket m \rrbracket s \text{ else } \llbracket n \rrbracket s \\ \llbracket \text{put}_{s'} \langle m \rangle \rrbracket &= \lambda s. \text{let } (x, ss) \leftarrow \llbracket m \rrbracket s' \text{ in } (x, s :: ss)\end{aligned}$$

Example: boolean state, standard interpretation

$$\llbracket \text{comp } A \rrbracket = \text{Bool} \rightarrow \llbracket A \rrbracket \times \text{Bool}$$

$$\llbracket \text{return } v \rrbracket = \lambda s. (\llbracket v \rrbracket, s)$$

$$\llbracket \text{get } \langle m, n \rangle \rrbracket = \lambda s. \text{if } s \text{ then } \llbracket m \rrbracket s \text{ else } \llbracket n \rrbracket s$$

$$\llbracket \text{put}_{s'} \langle m \rangle \rrbracket = \lambda s. \llbracket m \rrbracket s'$$

Sound and complete with respect to the equations

$$m \simeq n \iff \llbracket m \rrbracket = \llbracket n \rrbracket$$

Bit toggling

$$\llbracket \text{toggle} \rrbracket = \lambda s. \text{if } s \text{ then } (\text{tt}, \text{ff}) \text{ else } (\text{ff}, \text{tt})$$

Example: boolean state, discard interpretation

$$\llbracket \text{comp } A \rrbracket = \text{Bool} \rightarrow \llbracket A \rrbracket$$

$$\llbracket \text{return } v \rrbracket = \lambda s. \llbracket v \rrbracket$$

$$\llbracket \text{get } \langle m, n \rangle \rrbracket = \lambda s. \text{if } s \text{ then } \llbracket m \rrbracket s \text{ else } \llbracket n \rrbracket s$$

$$\llbracket \text{put}_{s'} \langle m \rangle \rrbracket = \lambda s. \llbracket m \rrbracket s'$$

Sound with respect to the equations

$$m \simeq n \implies \llbracket m \rrbracket = \llbracket n \rrbracket$$

Not complete because:

$$\llbracket \text{put}_s \langle \text{return } v \rangle \rrbracket = \llbracket \text{return } v \rrbracket$$

Bit toggling

$$\llbracket \text{toggle} \rrbracket = \lambda s. \text{if } s \text{ then tt else ff} = \lambda s. s$$

Example: boolean state, logging interpretation

$$\llbracket \text{comp } A \rrbracket = \text{Bool} \rightarrow \llbracket A \rrbracket \times \text{List Bool}$$

$$\llbracket \text{return } v \rrbracket = \lambda s. (\llbracket v \rrbracket, [s])$$

$$\llbracket \text{get } \langle m, n \rangle \rrbracket = \lambda s. \text{if } s \text{ then } \llbracket m \rrbracket s \text{ else } \llbracket n \rrbracket s$$

$$\llbracket \text{put}_{s'} \langle m \rangle \rrbracket = \lambda s. \text{let } (x, ss) \leftarrow \llbracket m \rrbracket s' \text{ in } (x, s :: ss)$$

Complete with respect to the equations

$$m \simeq n \iff \llbracket m \rrbracket = \llbracket n \rrbracket$$

Not sound because:

$$\begin{aligned} \llbracket \text{put}_s \langle \text{put}_{s'} \langle m \rangle \rangle \rrbracket &\neq \llbracket \text{put}_{s'} \langle m \rangle \rrbracket \\ \llbracket \text{get } \langle \text{put}_{\text{tt}} \langle m \rangle, \text{put}_{\text{ff}} \langle n \rangle \rangle \rrbracket &\neq \llbracket \text{get } \langle m, n \rangle \rrbracket \end{aligned}$$

Bit toggling

$$\llbracket \text{toggle} \rrbracket = \lambda s. \text{if } s \text{ then } (\text{tt}, [\text{tt}, \text{ff}]) \text{ else } (\text{ff}, [\text{ff}, \text{tt}])$$

Algebraic effects without equations

Different interpretations are useful in practice

So we will adopt **free** algebraic effects — no equations

Algebraic computations are command-response trees modulo equations

Abstract computations are plain command-response trees

Different interpretations give different meanings to the same abstract computation

Interpretations as effect handlers

Example: boolean state — standard interpretation

Meta level interpretation (enumerated continuations)

$$\begin{aligned} \llbracket \mathbf{return} \ v \rrbracket &= \lambda s. (\llbracket v \rrbracket, s) \\ \llbracket \mathbf{get} \ \langle m, n \rangle \rrbracket &= \lambda s. \mathbf{if} \ s \ \mathbf{then} \ \llbracket m \rrbracket s \ \mathbf{else} \ \llbracket n \rrbracket s \\ \llbracket \mathbf{put}_{s'} \ \langle m \rangle \rrbracket &= \lambda s. \llbracket m \rrbracket s' \end{aligned}$$

Interpretations as effect handlers

Example: boolean state — standard interpretation

Meta level interpretation (enumerated continuations)

$$\begin{aligned}\llbracket \mathbf{return} \ v \rrbracket &= \lambda s. (\llbracket v \rrbracket, s) \\ \llbracket \mathbf{get} \ \langle m, n \rangle \rrbracket &= \lambda s. \mathbf{if} \ s \ \mathbf{then} \ \llbracket m \rrbracket s \ \mathbf{else} \ \llbracket n \rrbracket s \\ \llbracket \mathbf{put}_{s'} \ \langle m \rangle \rrbracket &= \lambda s. \llbracket m \rrbracket s'\end{aligned}$$

Meta level interpretation (continuations as functions)

$$\begin{aligned}\llbracket \mathbf{return} \ v \rrbracket &= \lambda s. (\llbracket v \rrbracket, s) \\ \llbracket \mathbf{get} \ k \rrbracket &= \lambda s. \llbracket k \ s \rrbracket s \\ \llbracket \mathbf{put}_{s'} \ k \rrbracket &= \lambda s. \llbracket k \ () \rrbracket s'\end{aligned}$$

Interpretations as effect handlers

Example: boolean state — standard interpretation

Meta level interpretation (enumerated continuations)

$$\begin{aligned} \llbracket \mathbf{return} \ v \rrbracket &= \lambda s. (\llbracket v \rrbracket, s) \\ \llbracket \mathbf{get} \ \langle m, n \rangle \rrbracket &= \lambda s. \mathbf{if} \ s \ \mathbf{then} \ \llbracket m \rrbracket s \ \mathbf{else} \ \llbracket n \rrbracket s \\ \llbracket \mathbf{put}_{s'} \ \langle m \rangle \rrbracket &= \lambda s. \llbracket m \rrbracket s' \end{aligned}$$

Meta level interpretation (continuations as functions)

$$\begin{aligned} \llbracket \mathbf{return} \ v \rrbracket &= \lambda s. (\llbracket v \rrbracket, s) \\ \llbracket \mathbf{get} \ k \rrbracket &= \lambda s. \llbracket k \ s \rrbracket s \\ \llbracket \mathbf{put}_{s'} \ k \rrbracket &= \lambda s. \llbracket k \ () \rrbracket s' \end{aligned}$$

Object level effect handler

$$\begin{aligned} \mathbf{return} \ v &\mapsto \lambda s. (v, s) \\ \langle \mathbf{get} \ () \rightarrow r \rangle &\mapsto \lambda s. r \ s \ s \\ \langle \mathbf{put} \ s' \rightarrow r \rangle &\mapsto \lambda s. r \ () \ s' \end{aligned}$$

Interpretations as effect handlers

Example: nondeterminism

Meta level interpretation (enumerated continuations)

$$\begin{aligned} \llbracket \mathbf{return} \ v \rrbracket &= \llbracket v \rrbracket \\ \llbracket \mathbf{choose} \ \langle m, n \rangle \rrbracket &= \llbracket m \rrbracket \text{ ++ } \llbracket n \rrbracket \\ \llbracket \mathbf{fail} \ \langle \rangle \rrbracket &= \llbracket \ \rrbracket \end{aligned}$$

Interpretations as effect handlers

Example: nondeterminism

Meta level interpretation (enumerated continuations)

$$\begin{aligned} \llbracket \mathbf{return} \ v \rrbracket &= \llbracket v \rrbracket \\ \llbracket \mathbf{choose} \ \langle m, n \rangle \rrbracket &= \llbracket m \rrbracket \text{ ++ } \llbracket n \rrbracket \\ \llbracket \mathbf{fail} \ \langle \rangle \rrbracket &= [] \end{aligned}$$

Meta level interpretation (continuations as functions)

$$\begin{aligned} \llbracket \mathbf{return} \ v \rrbracket &= \llbracket v \rrbracket \\ \llbracket \mathbf{choose} \ k \rrbracket &= \llbracket k \ \mathbf{tt} \rrbracket \text{ ++ } \llbracket k \ \mathbf{ff} \rrbracket \\ \llbracket \mathbf{fail} \ k \rrbracket &= [] \end{aligned}$$

Interpretations as effect handlers

Example: nondeterminism

Meta level interpretation (enumerated continuations)

$$\begin{aligned} \llbracket \mathbf{return} \ v \rrbracket &= \llbracket [v] \rrbracket \\ \llbracket \mathbf{choose} \ \langle m, n \rangle \rrbracket &= \llbracket [m] \rrbracket ++ \llbracket [n] \rrbracket \\ \llbracket \mathbf{fail} \ \langle \rangle \rrbracket &= [] \end{aligned}$$

Meta level interpretation (continuations as functions)

$$\begin{aligned} \llbracket \mathbf{return} \ v \rrbracket &= \llbracket [v] \rrbracket \\ \llbracket \mathbf{choose} \ k \rrbracket &= \llbracket [k \ \mathbf{tt}] \rrbracket ++ \llbracket [k \ \mathbf{ff}] \rrbracket \\ \llbracket \mathbf{fail} \ k \rrbracket &= [] \end{aligned}$$

Object level effect handler

$$\begin{aligned} \mathbf{return} \ v &\mapsto [v] \\ \langle \mathbf{choose} \ () \rightarrow r \rangle &\mapsto r \ \mathbf{tt} ++ r \ \mathbf{ff} \\ \langle \mathbf{fail} \ () \rightarrow r \rangle &\mapsto [] \end{aligned}$$

Parameterised operations (term parameters)

For convenience we write

$\text{op} : A \rightarrow B$ instead of $\text{op}_i : B, i \in A$

and to perform an operation:

$\text{op } i$ instead of op_i

Parameterised operations (term parameters)

For convenience we write

$\text{op} : A \rightarrow B$ instead of $\text{op}_i : B, i \in A$

and to perform an operation:

$\text{op } i$ instead of op_i

For uniformity we parameterise all operations in effect signatures.

Parameterised operations (term parameters)

For convenience we write

$op : A \rightarrow B$ instead of $op_i : B, i \in A$

and to perform an operation:

$op\ i$ instead of op_i

For uniformity we parameterise all operations in effect signatures.

Examples

natural number state	{ $put : Nat \rightarrow 1, get : 1 \rightarrow Nat$ }
boolean state	{ $put : Bool \rightarrow 1, get : 1 \rightarrow Bool$ }
nondeterminism	{ $choose : 1 \rightarrow Bool, fail : 1 \rightarrow 0$ }

Parametric operations (type parameters)

It can be useful to parameterise an operation by one or more types.

Example: nondeterminism

$$\{\text{choose} : 1 \rightarrow \text{Bool}, \text{fail} : 1 \rightarrow 0\}$$

becomes

$$\{\text{choose} : 1 \rightarrow \text{Bool}, \text{fail} : a.1 \rightarrow a\}$$

and

if `choose ()` **then** (**if** `choose ()` **then** Heads **else** Tails) **else** **absurd** (`fail ()`)

becomes

if `choose ()` **then** (**if** `choose ()` **then** Heads **else** Tails) **else** `fail ()`

Example: choice and failure

Effect signature

$$\{\text{choose} : 1 \rightarrow \text{Bool}, \text{fail} : a.1 \rightarrow a\}$$

Example: choice and failure

Effect signature

$\{\text{choose} : 1 \rightarrow \text{Bool}, \text{fail} : a.1 \rightarrow a\}$

Drunk coin tossing

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

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

$\text{drunkTosses } n = \text{if } n = 0 \text{ then } []$
 else $\text{drunkToss} () :: \text{drunkTosses } (n - 1)$

Example: choice and failure

Handlers

maybeFail = — exception handler

return $x \mapsto$ Just x

fail () \mapsto Nothing

Example: choice and failure

Handlers

maybeFail = — exception handler

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

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

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

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

Example: choice and failure

Handlers

maybeFail = — exception handler

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

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

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

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

trueChoice = — linear handler

return $x \mapsto x$

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

Example: choice and failure

Handlers

maybeFail = — exception handler

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

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

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

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

trueChoice = — linear handler

return $x \mapsto x$

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

handle 42 **with** trueChoice $\implies 42$

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

Example: choice and failure

Handlers

maybeFail = — exception handler

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

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

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

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

trueChoice = — linear handler

return $x \mapsto x$

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

handle 42 **with** trueChoice $\implies 42$

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

allChoices = — non-linear handler

return $x \mapsto [x]$

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

Example: choice and failure

Handlers

maybeFail = — exception handler

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

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

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

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

trueChoice = — linear handler

return $x \mapsto x$

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

handle 42 **with** trueChoice $\implies 42$

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

allChoices = — non-linear handler

return $x \mapsto [x]$

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

handle 42 **with** allChoices $\implies [42]$

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

Example: choice and failure

Handlers

maybeFail = — exception handler

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

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

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

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

trueChoice = — linear handler

return $x \mapsto x$

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

handle 42 **with** trueChoice $\implies 42$

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

allChoices = — non-linear handler

return $x \mapsto [x]$

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

handle 42 **with** allChoices $\implies [42]$

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

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

Example: choice and failure

Handlers

maybeFail = — exception handler

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

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

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

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

trueChoice = — linear handler

return $x \mapsto x$

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

handle 42 **with** trueChoice $\implies 42$

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

allChoices = — non-linear handler

return $x \mapsto [x]$

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

handle 42 **with** allChoices $\implies [42]$

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

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

Operational semantics

Reduction rules

let $x = V$ **in** $N \rightsquigarrow N[V/x]$

handle V **with** $H \rightsquigarrow N[V/x]$

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

where

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

Evaluation contexts

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

Typing rules

Effects

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

Computations

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

Operations

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

Handlers

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

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

Typing rules

Effects

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

Computations

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

Operations

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

Handlers

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

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

Exercise: Adapt the typing rules to accommodate parametric operations

What is an effect handler?

What is an effect handler?

- ▶ A **modular** interpreter for effectful computations

What is an effect handler?

- ▶ A **modular** interpreter for effectful computations
- ▶ A generalisation of an exception handler
 - ▶ based on exceptional syntax [Benton and Kennedy, 2001]

let $x = (\text{try } M \text{ with } H) \text{ in } N$	conventional exception handlers
↓	
try $M \text{ as } x \text{ in } N \text{ otherwise } H$	exceptional syntax
↓	
handle $M \text{ with } \{\text{return } x \mapsto N; H\}$	effect handlers

- ▶ success continuations aid composition, optimisation, and reasoning
- ▶ resumable

What is an effect handler?

- ▶ A **modular** interpreter for effectful computations
- ▶ A generalisation of an exception handler
 - ▶ based on exceptional syntax [Benton and Kennedy, 2001]

let $x = (\text{try } M \text{ with } H) \text{ in } N$	conventional exception handlers
↓	
try $M \text{ as } x \text{ in } N \text{ otherwise } H$	exceptional syntax
↓	
handle $M \text{ with } \{\text{return } x \mapsto N; H\}$	effect handlers

success continuations aid composition, optimisation, and reasoning

- ▶ resumable
- ▶ A morphism between (free) algebras

What is an effect handler?

- ▶ A **modular** interpreter for effectful computations
- ▶ A generalisation of an exception handler
 - ▶ based on exceptional syntax [Benton and Kennedy, 2001]

let $x = (\text{try } M \text{ with } H) \text{ in } N$	conventional exception handlers
↓	
try $M \text{ as } x \text{ in } N \text{ otherwise } H$	exceptional syntax
↓	
handle $M \text{ with } \{\text{return } x \mapsto N; H\}$	effect handlers

success continuations aid composition, optimisation, and reasoning

- ▶ resumable
- ▶ A morphism between (free) algebras
- ▶ A fold (catamorphism) over a command-response tree

What is an effect handler?

- ▶ A **modular** interpreter for effectful computations
- ▶ A generalisation of an exception handler
 - ▶ based on exceptional syntax [Benton and Kennedy, 2001]

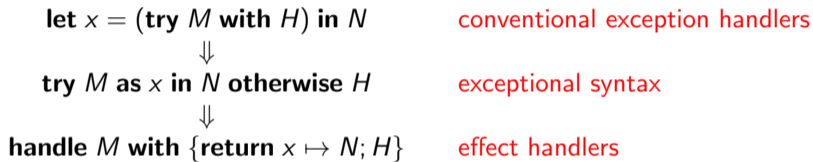
let $x = (\mathbf{try} \ M \ \mathbf{with} \ H) \ \mathbf{in} \ N$	conventional exception handlers
↓	
try $M \ \mathbf{as} \ x \ \mathbf{in} \ N \ \mathbf{otherwise} \ H$	exceptional syntax
↓	
handle $M \ \mathbf{with} \ \{\mathbf{return} \ x \mapsto N; H\}$	effect handlers

success continuations aid composition, optimisation, and reasoning

- ▶ resumable
- ▶ A morphism between (free) algebras
- ▶ A fold (catamorphism) over a command-response tree
- ▶ A structured delimited control operator

What is an effect handler?

- ▶ A **modular** interpreter for effectful computations
- ▶ A generalisation of an exception handler
 - ▶ based on exceptional syntax [Benton and Kennedy, 2001]



success continuations aid composition, optimisation, and reasoning

- ▶ resumable
- ▶ A morphism between (free) algebras
- ▶ A fold (catamorphism) over a command-response tree
- ▶ A structured delimited control operator
- ▶ A composable user-defined operating system

Example: cooperative concurrency (parameterised handler)

Effect signature

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

Example: 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: cooperative concurrency (parameterised handler)

Effect signature

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

Two cooperative lightweight threads

$tA () = \text{print} ("A1 "); \text{yield} (); \text{print} ("A2 ")$

$tB () = \text{print} ("B1 "); \text{yield} (); \text{print} ("B2 ")$

Handler — parameterised handler

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

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

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

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

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

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

Example: cooperative concurrency (parameterised handler)

Effect signature

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

Two cooperative lightweight threads

$tA () = \text{print} ("A1 "); \text{yield} (); \text{print} ("A2 ")$

$tB () = \text{print} ("B1 "); \text{yield} (); \text{print} ("B2 ")$

Handler — parameterised handler

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

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

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

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

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

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

Helpers

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

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

Example: cooperative concurrency (parameterised handler)

Effect signature

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

Two cooperative lightweight threads

$tA () = \text{print} ("A1 "); \text{yield} (); \text{print} ("A2 ")$

$tB () = \text{print} ("B1 "); \text{yield} (); \text{print} ("B2 ")$

Handler — parameterised handler

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

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

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

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

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

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

Helpers

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

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

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

A1 B1 A2 B2

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

where $H h = \text{return } x \mapsto N$

$\langle \text{op}_1 p \rightarrow r \rangle \mapsto N_{\text{op}_1}$

...

$\langle \text{op}_k p \rightarrow r \rangle \mapsto N_{\text{op}_k}$

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

where H $h = \text{return } x \mapsto N$

$\langle \text{op}_1 p \rightarrow r \rangle \mapsto N_{\text{op}_1}$

...

$\langle \text{op}_k p \rightarrow r \rangle \mapsto N_{\text{op}_k}$

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

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

Computations

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

Operations

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

Handlers

$$\frac{\Gamma \vdash M : C \quad \Gamma \vdash V : P \quad \Gamma \vdash H : P \rightarrow C \Rightarrow D}{\Gamma \vdash \text{handle } M \text{ with } H V : D}$$

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

Example: cooperative concurrency with UNIX-style fork

Effect signature

$\{\text{yield} : 1 \rightarrow 1, \text{ufork} : 1 \rightarrow \text{Bool}\}$

Example: cooperative concurrency with UNIX-style fork

Effect signature

$\{\text{yield} : 1 \rightarrow 1, \text{ufork} : 1 \rightarrow \text{Bool}\}$

A single cooperative program

```
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: cooperative concurrency with UNIX-style fork

Effect signature

$$\{\text{yield} : 1 \rightarrow 1, \text{ufork} : 1 \rightarrow \text{Bool}\}$$

A single cooperative program

```
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 "
```

Handler

coop ([]) =

return () \mapsto ()

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

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

coop (r :: rs) =

return () \mapsto r rs ()

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

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

Example: cooperative concurrency with UNIX-style fork

Effect signature

$$\{\text{yield} : 1 \rightarrow 1, \text{ufork} : 1 \rightarrow \text{Bool}\}$$

A single cooperative program

```
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 "
```

Handler

coop ([]) =

return () $\mapsto ()$

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

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

coop (r :: rs) =

return () $\mapsto r rs ()$

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

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

cooperate [main] $\Longrightarrow ()$

M1 A1 M2 B1 A2 M3 B2

Example: cooperative concurrency with UNIX-style fork

Effect signature

$\{\text{yield} : 1 \rightarrow 1, \text{ufork} : 1 \rightarrow \text{Bool}\}$

A single cooperative program

```
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 "
```

Handler

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

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

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

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

Example: cooperative concurrency with UNIX-style fork

Effect signature

$$\{\text{yield} : 1 \rightarrow 1, \text{ufork} : 1 \rightarrow \text{Bool}\}$$

A single cooperative program

```
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 "
```

Handler

coop ([]) =

```
return ()      ↦ ()
⟨yield () → r'⟩ ↦ r' [] ()
⟨ufork () → r'⟩ ↦ r' [λrs ().r' rs tt]
                  ff
```

coop (r :: rs) =

```
return ()      ↦ r rs ()
⟨yield () → r'⟩ ↦ r (rs ++ [r']) ()
⟨ufork () → r'⟩ ↦ r' (r :: rs ++ [λrs ().r' rs tt])
                  ff
```

cooperate [main] \Longrightarrow ()

M1 M2 M3 A1 B1 A2 B2

Example: cooperative concurrency with UNIX-style fork

Effect signature

$$\{\text{yield} : 1 \rightarrow 1, \text{ufork} : 1 \rightarrow \text{Bool}\}$$

A single cooperative program

```
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 "
```

Handler

coop ([]) =

```
return ()      ↦ ()
⟨yield () → r'⟩ ↦ r' [] ()
⟨ufork () → r'⟩ ↦ r' [λrs ().r' rs tt]
                  ff
```

coop (r :: rs) =

```
return ()      ↦ r rs ()
⟨yield () → r'⟩ ↦ r (rs ++ [r']) ()
⟨ufork () → r'⟩ ↦ r' (r :: rs ++ [λrs ().r' rs tt])
                  ff
```

cooperate [main] \Longrightarrow ()

M1 M2 M3 A1 B1 A2 B2

Exercise: implement a handler for a fork operation that uses the resumption linearly

Example: cooperative concurrency (shallow handler)

Effect signature

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

Example: cooperative concurrency (shallow handler)

Effect signature

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

Two cooperative lightweight threads

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

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

Example: cooperative concurrency (shallow handler)

Effect signature

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

Two cooperative lightweight threads

$$tA () = \text{print} ("A1 "); \text{yield} (); \text{print} ("A2 ")$$
$$tB () = \text{print} ("B1 "); \text{yield} (); \text{print} ("B2 ")$$

Handler — shallow handler

$$\text{cooperate} [] = ()$$
$$\begin{aligned} \text{cooperate} [r] &= \mathbf{\text{handle } r() \text{ with}} \\ \mathbf{\text{return } ()} &\quad \mapsto () \\ \langle \text{yield } () \rightarrow s \rangle &\mapsto \text{cooperate} [s] \end{aligned}$$
$$\begin{aligned} \text{cooperate} (r :: s :: rs) &= \mathbf{\text{handle } r() \text{ with}} \\ \mathbf{\text{return } ()} &\quad \mapsto \text{cooperate} (s :: rs) \\ \langle \text{yield } () \rightarrow t \rangle &\mapsto \text{cooperate} (s :: rs ++ [t]) \end{aligned}$$

Example: cooperative concurrency (shallow handler)

Effect signature

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

Two cooperative lightweight threads

$$tA () = \text{print} ("A1 "); \text{yield} (); \text{print} ("A2 ")$$
$$tB () = \text{print} ("B1 "); \text{yield} (); \text{print} ("B2 ")$$

Handler — shallow handler

$$\text{cooperate} [] = ()$$
$$\begin{aligned} \text{cooperate} [r] &= \text{handle } r() \text{ with} \\ \text{return} () &\mapsto () \\ \langle \text{yield} () \rightarrow s \rangle &\mapsto \text{cooperate} [s] \end{aligned}$$
$$\begin{aligned} \text{cooperate} (r :: s :: rs) &= \text{handle } r() \text{ with} \\ \text{return} () &\mapsto \text{cooperate} (s :: rs) \\ \langle \text{yield} () \rightarrow t \rangle &\mapsto \text{cooperate} (s :: rs ++ [t]) \end{aligned}$$
$$\text{cooperate} [tA, tB] \Longrightarrow ()$$

A1 B1 A2 B2

Operational semantics (shallow handlers)

Reduction rules

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

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

Evaluation contexts

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

Operational semantics (shallow handlers)

Reduction rules

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

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

Evaluation contexts

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

Exercise: express shallow handlers as deep handlers

Typing rules (shallow handlers)

Effects

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

Computations

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

Operations

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

Handlers

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

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

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
Multicore OCaml	https://github.com/ocaml-labs/ocaml-multicore/wiki

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