Higher-Order Asynchronous Effects

Danel Ahman

University of Tartu, Estonia

(joint work with Matija Pretnar)

EHOP Workshop ∼ 20.08.2025

Plan

- **Problems:**
  - usual (operational) treatment of alg. effs. is synchronous
  - some natural examples require language-specific hacks
- **Solution proposed at POPL'21:**
  - asynchrony through decoupling operation call execution into signals and interrupts
- **Solutions to some POPL'21 shortcomings in LMCS:**
  - modal type system for higher-order signals and interrupts
  - reinstallable and stateful interrupt handlers to remove gen. rec.

D. Ahman, M. Pretnar. *Asynchronous Effects* (POPL 2021)

D. Ahman, M. Pretnar. *Higher-Order Async. Effs.* (LMCS, 2024)

Problems

Problem 1: synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$\ldots \quad \rightsquigarrow \ \mathsf{op}\,(V, y.N)$$

Problem 1: synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$
\begin{array}{c}
M_{\mathrm{op}}[V/x] \\
\text{signalling op's implementation} \quad \uparrow \\
\ldots \quad \rightsquigarrow \ \mathrm{op}\,(V, y.N)
\end{array}
$$

  - M_{op} - handler, runner, top-level default implementation, ...

Problem 1: synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$M_{\mathsf{op}}[V/x] \quad \rightsquigarrow^* \quad \mathsf{return}\ W$$

signalling op's implementation $\uparrow$

$$\ldots \quad \rightsquigarrow \ \mathsf{op}\,(V, y.N)$$

  - M_{op} - handler, runner, top-level default implementation, ...

Problem 1: synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$
\begin{array}{ccccc}
 & & M_{\mathsf{op}}[V/x] & \rightsquigarrow^{*} & \mathsf{return}\ W \\
 & \text{signalling op's implementation} & \big\uparrow & & \big\downarrow\ \text{interrupting main program} \\
\ldots & \rightsquigarrow & \mathsf{op}\,(V, y.N) & & N[W/y]
\end{array}
$$

  - M_{op} - handler, runner, top-level default implementation, ...

Problem 1: synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$M_{\mathsf{op}}[V/x] \quad \leadsto^* \quad \mathsf{return}\ W$$

signalling op's implementation $\uparrow$ $\qquad$ $\downarrow$ interrupting main program

$$\ldots \quad \leadsto \ \mathsf{op}\,(V, y.N) \qquad\qquad N[W/y] \leadsto \quad \ldots$$

  - M_{op} - handler, runner, top-level default implementation, ...

Problem 1: synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$M_{\mathsf{op}}[V/x] \quad \rightsquigarrow^* \quad \mathsf{return}\ W$$

signalling op's implementation ↑ ↓ interrupting main program

$$\ldots \quad \rightsquigarrow \quad \mathsf{op}\,(V, y.N) \qquad\qquad N[W/y] \rightsquigarrow \quad \ldots$$

main program's execution is blocked

  - M_{op} - handler, runner, top-level default implementation, ...

Problem 1: synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$M_{\mathsf{op}}[V/x] \quad \rightsquigarrow^* \quad \mathsf{return}\ W$$

signalling op's implementation $\uparrow$ $\qquad$ $\downarrow$ interrupting main program

$$\ldots \quad \rightsquigarrow \ \mathsf{op}\ (V, y.N) \qquad\qquad N[W/y] \ \rightsquigarrow \quad \ldots$$

main program's execution is blocked

  - M_{op} - handler, runner, top-level default implementation, ...

- While such synchrony is needed for general effect handlers, it unnecessarily forces all uses of alg. effs. to be synchronous

Problem 2: some neat examples are hacky

Problem 2: some neat examples are hacky

- The leading example of eff. handlers is user-definable cooperative multi-threading (e.g., that's why handlers are in OCaml 5)

```
let rec scheduler () =
  handler {
    | yield _ k → enqueue k ; dequeue ()
    | fork  f k → enqueue k ; handle f () with (scheduler ()) to _ in dequeue () }

let runCooperatively f =
  handle f () with (scheduler ()) to _ in dequeue ()
```

Problem 2: some neat examples are hacky

- The leading example of eff. handlers is user-definable cooperative multi-threading (e.g., that's why handlers are in OCaml 5)

```
let rec scheduler () =
  handler {
    | yield _ k → enqueue k ; dequeue ()
    | fork  f k → enqueue k ; handle f () with (scheduler ()) to _ in dequeue () }

let runCooperatively f =
  handle f () with (scheduler ()) to _ in dequeue ()
```

- Usual attempts at preemptive multi-th. are much less principled
  - people typically rely on (low-level) language specifics (of OCaml, Node.js) to inject yields into their programs at runtime

Problem 2: some neat examples are hacky

- The leading example of eff. handlers is user-definable cooperative multi-threading (e.g., that's why handlers are in OCaml 5)

```
let rec scheduler () =
  handler {
    | yield _ k → enqueue k ; dequeue ()
    | fork  f k → enqueue k ; handle f () with (scheduler ()) to _ in dequeue () }

let runCooperatively f =
  handle f () with (scheduler ()) to _ in dequeue ()
```

- Usual attempts at preemptive multi-th. are much less principled
  - people typically rely on (low-level) language specifics (of OCaml, Node.js) to inject yields into their programs at runtime
- In our work, we show how this can be achieved in a natural and self-contained fashion (including insights for ordinary alg. effs.)

Our Solution

operation calls

=

signals + interrupts + interrupt handlers

The gist of our approach (1)

- Recall that the execution of operation calls has the shape

$$M_{\mathsf{op}}[V/x] \quad \leadsto^* \quad \mathsf{return}\ W$$

signalling op's implementation $\uparrow$ $\qquad$ $\downarrow$ interrupting main program

$$\ldots \quad \leadsto \ \mathsf{op}\,(V, y.N) \qquad\qquad N[W/y] \ \leadsto \quad \ldots$$

- We turn these phases into separate programming abstractions

The gist of our approach (1)

- Recall that the execution of operation calls has the shape

$$
\begin{array}{ccc}
& M_{\mathsf{op}}[V/x] \quad \rightsquigarrow^* \quad \mathsf{return}\ W & \\
\text{signalling op's implementation} \uparrow & & \downarrow \text{interrupting main program} \\
\ldots \quad \rightsquigarrow \ \mathsf{op}\,(V, y.N) & & N[W/y] \ \rightsquigarrow \quad \ldots
\end{array}
$$

- We turn these phases into separate programming abstractions

  - signals $\quad \cdots \rightsquigarrow \uparrow \mathsf{op}\,(V, M) \overset{\mathsf{op}\ V \uparrow}{\rightsquigarrow} M \rightsquigarrow \cdots$

The gist of our approach (1)

- Recall that the execution of operation calls has the shape

$$
\begin{array}{ccccc}
 & M_{\mathsf{op}}[V/x] & \rightsquigarrow^{*} & \mathsf{return}\ W & \\
\text{signalling op's implementation} & \uparrow & & \downarrow & \text{interrupting main program} \\
\ldots \rightsquigarrow & \mathsf{op}\,(V, y.N) & & N[W/y] & \rightsquigarrow \ldots
\end{array}
$$

- We turn these phases into separate programming abstractions

  - signals
  $$\cdots \rightsquigarrow \uparrow \mathsf{op}\,(V, M) \overset{\mathsf{op}\ V \uparrow}{\rightsquigarrow} M \rightsquigarrow \cdots$$

  - interrupts
  $$\cdots \rightsquigarrow M \overset{\downarrow \mathsf{op}\ W}{\rightsquigarrow} \downarrow \mathsf{op}\,(W, M) \rightsquigarrow \cdots$$

The gist of our approach (2)

- Recall that the execution of operation calls has the shape

$$\begin{array}{ccc} M_{\mathsf{op}}[V/x] & \rightsquigarrow^{*} & \mathbf{return}\ W \\ \text{signalling op's implementation} \uparrow & & \downarrow \text{interrupting main program} \\ \ldots \rightsquigarrow \mathsf{op}\,(V, y.N) & & N[W/y] \rightsquigarrow \ldots \end{array}$$

- We turn these phases into separate programming abstractions

  - interrupt handlers

$$M, N \ ::= \ \cdots \ \big|\ \ \mathbf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathbf{as}\ p : \langle X \rangle\ \mathbf{in}\ N$$

The gist of our approach (2)

- Recall that the execution of operation calls has the shape

$$
\begin{array}{ccc}
M_{\mathsf{op}}[V/x] & \leadsto^* & \mathbf{return}\ W \\
\text{signalling op's implementation} \uparrow & & \downarrow \text{interrupting main program} \\
\ldots \leadsto \mathsf{op}\,(V, y.N) & & N[W/y] \leadsto \ldots
\end{array}
$$

- We turn these phases into separate programming abstractions

  - interrupt handlers

$$M, N \ ::= \ \cdots \ \mid \ \mathbf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathbf{as}\ p : \langle X \rangle\ \mathbf{in}\ N$$

  - awaiting promises to be fulfilled

$$V, W \ ::= \ \cdots \ \mid \ \langle V \rangle$$

$$M, N \ ::= \ \cdots \ \mid \ \mathbf{await}\ V\ \mathbf{until}\ \langle x \rangle\ \mathbf{in}\ N$$

The gist of our approach (3)

- Recall that the execution of operation calls has the shape

$$M_{\mathsf{op}}[V/x] \quad \rightsquigarrow^* \quad \mathtt{return}\ W$$

signalling op's implementation $\uparrow$ $\qquad\qquad$ $\downarrow$ interrupting main program

$$\ldots \quad \rightsquigarrow \quad \mathsf{op}\,(V, y.N) \qquad\qquad N[W/y] \rightsquigarrow \quad \ldots$$

- We turn these phases into separate programming abstractions
  - parallel processes

$$P, Q \ ::= \ \mathtt{run}\ M \ \mid \ P \,||\, Q \ \mid \ \uparrow \mathsf{op}\,(V, P) \ \mid \ \downarrow \mathsf{op}\,(W, P)$$

which we use to model the programs' environment

$\lambda_{\text{æ}}$-calculus

$\lambda_{æ}$-calculus: basics

- Extension of Levy's fine-grain call-by-value λ-calculus (FGCBV)

- **Types:** $X, Y \ ::= \ \mathsf{b} \ \mid \ \ldots \ \mid \ X \to Y \,!\, (o, \iota) \ \mid \ \ldots$

- **Values:** $V, W \ ::= \ x \ \mid \ \ldots \ \mid \ \mathsf{fun}\ (x : X) \mapsto M \ \mid \ \ldots$

- **Computations:** $M, N \ ::= \ \mathsf{return}\ V \ \mid \ \mathsf{let}\ x = M\ \mathsf{in}\ N \ \mid \ \ldots$

- **Typing judgements:** $\Gamma \vdash V : X \qquad \Gamma \vdash M : X \,!\, (o, \iota)$

- **Effect annotations** (o, ι)**:**

$$o \subseteq \mathcal{O} \qquad \iota = \{\, \mathsf{op}_1 \mapsto (o_1, \iota_1)\,, \ \ldots\ , \mathsf{op}_n \mapsto (o_n, \iota_n) \,\}$$

- **Small-step operational semantics:** $M \rightsquigarrow N$

$\lambda_{æ}$-calculus: modal types

$\lambda_{æ}$-calculus: modal types

- (Almost) off-the-shelf Fitch-style modal $[X]$-type [Clouston et al.]

$$X \;::=\; \ldots \;\mid\; [X] \qquad\qquad \Gamma \;::=\; \emptyset \;\mid\; \Gamma, x : X \;\mid\; \Gamma, \text{🔓}$$

Ty-Val-Variable

$$\frac{X \text{ is mobile} \quad \vee \quad \text{🔓} \notin \Gamma'}{\Gamma, x : X, \Gamma' \vdash x : X}$$

Ty-Val-Box

$$\frac{\Gamma, \text{🔓} \vdash V : X}{\Gamma \vdash \mathsf{box}\ V : [X]}$$

Ty-Comp-Unbox

$$\frac{\Gamma \vdash V : [X] \qquad \Gamma, x : X \vdash M : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{unbox}\ V\ \mathsf{as\ box}\ x\ \mathsf{in}\ M : Y \,!\, (o, \iota)}$$

where X is mobile if X is a ground type or a modal type $[Y]$

- Intuition: $[X]$ contains X-typed vals. safe to send to other procs.

$\lambda_{æ}$-calculus: signals

- Signalling that some op's implementation needs to be executed

TyComp-Signal

$$\frac{\mathsf{op} : A_{\mathsf{op}} \in o \qquad \Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash M : X \,!\, (o, \iota)}{\Gamma \vdash \uparrow \mathsf{op}\,(V, M) : X \,!\, (o, \iota)}$$

where A_{op} is a mobile type (so it is safe to send to other procs.)

$\lambda_{æ}$-calculus: signals

- Signalling that some op's implementation needs to be executed

$$\textsc{TyComp-Signal}\quad \frac{\mathsf{op} : A_{\mathsf{op}} \in o \qquad \Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash M : X \,!\, (o, \iota)}{\Gamma \vdash \uparrow \mathsf{op}\,(V, M) : X \,!\, (o, \iota)}$$

where A_{op} is a mobile type (so it is safe to send to other procs.)

- Operationally behave like algebraic operations
  - $\mathsf{let}\ x = \left(\uparrow \mathsf{op}\,(V, M)\right)\ \mathsf{in}\ N \ \rightsquigarrow\ \uparrow \mathsf{op}\,(V, \mathsf{let}\ x = M\ \mathsf{in}\ N)$

$\lambda_{æ}$-calculus: signals

- Signalling that some op's implementation needs to be executed

$$\textsc{TyComp-Signal}\quad \frac{\mathsf{op} : A_{\mathsf{op}} \in o \qquad \Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash M : X \,!\, (o, \iota)}{\Gamma \vdash \uparrow \mathsf{op}\,(V, M) : X \,!\, (o, \iota)}$$

  where A_{op} is a mobile type (so it is safe to send to other procs.)

- Operationally behave like algebraic operations

  - $\mathsf{let}\ x = (\uparrow \mathsf{op}\,(V, M))\ \mathsf{in}\ N \ \rightsquigarrow\ \uparrow \mathsf{op}\,(V, \mathsf{let}\ x = M\ \mathsf{in}\ N)$

- But importantly, they do not block their continuations

  - $M \rightsquigarrow M' \qquad \Longrightarrow \qquad \uparrow \mathsf{op}\,(V, M) \rightsquigarrow \uparrow \mathsf{op}\,(V, M')$

$\lambda_{æ}$-calculus: interrupts

- Environment interrupting a computation (with some op's result)

TyComp-Interrupt

$$\frac{\Gamma \vdash W : A_{\mathsf{op}} \qquad \Gamma \vdash M : X \,!\, (o, \iota)}{\Gamma \vdash \downarrow \mathsf{op}\,(W, M) : X \,!\, (\mathsf{op} \downarrow (o, \iota))}$$

where A_{op} is a mobile type (so it is safe to send to other procs.)

$\lambda_{æ}$-calculus: interrupts

- Environment interrupting a computation (with some op's result)

$$\frac{\Gamma \vdash W : A_{\mathsf{op}} \qquad \Gamma \vdash M : X \,!\, (o, \iota)}{\Gamma \vdash \downarrow \mathsf{op}\,(W, M) : X \,!\, (\mathsf{op} \downarrow (o, \iota))}\ \textsc{TyComp-Interrupt}$$

  where A_{op} is a mobile type (so it is safe to send to other procs.)

- Operationally behave like homomorphisms/effect handling
  - $\downarrow \mathsf{op}\,(W, \mathbf{return}\ V) \rightsquigarrow \mathbf{return}\ V$
  - $\downarrow \mathsf{op}\,(W, \uparrow \mathsf{op}'\,(V, M)) \rightsquigarrow \uparrow \mathsf{op}'\,(V, \downarrow \mathsf{op}\,(W, M))$
  - ...

$\lambda_{æ}$-calculus: interrupts

- Environment interrupting a computation (with some op's result)

$$\textsc{TyComp-Interrupt}\quad \frac{\Gamma \vdash W : A_{\mathsf{op}} \qquad \Gamma \vdash M : X \,!\, (o, \iota)}{\Gamma \vdash \downarrow \mathsf{op}\,(W, M) : X \,!\, (\mathsf{op}\downarrow(o, \iota))}$$

  where A_{op} is a mobile type (so it is safe to send to other procs.)

- Operationally behave like homomorphisms/effect handling
  - $\downarrow \mathsf{op}\,(W, \mathsf{return}\ V) \leadsto \mathsf{return}\ V$
  - $\downarrow \mathsf{op}\,(W, \uparrow \mathsf{op}'\,(V, M)) \leadsto \uparrow \mathsf{op}'\,(V, \downarrow \mathsf{op}\,(W, M))$
  - ...

- And they also do not block their continuations
  - $M \leadsto M' \qquad \Longrightarrow \qquad \downarrow \mathsf{op}\,(V, M) \leadsto \downarrow \mathsf{op}\,(V, M')$

$\lambda_{æ}$-calculus: interrupt handlers

- Allow computations to react to interrupts

TY-COMP-PROMISE

$$\frac{\iota\,(\mathsf{op}) = (o', \iota') \qquad \Gamma, x : A_{op} \vdash M : \langle X \rangle \,!\, (o', \iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y \,!\, (o, \iota)}$$

where $p : \langle X \rangle$ is a promise-typed variable

$\lambda_{æ}$-calculus: interrupt handlers

- Allow computations to react to interrupts

$$\frac{\begin{array}{c}\textsc{Ty-Comp-Promise} \\ \iota\,(\mathrm{op}) = (o', \iota') \qquad \Gamma, x : A_{op} \vdash M : \langle X \rangle \,!\, (o', \iota') \\ \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)\end{array}}{\Gamma \vdash \mathsf{promise}\ (\mathrm{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y \,!\, (o, \iota)}$$

where $p : \langle X \rangle$ is a promise-typed variable

- Operationally behave like (scoped) algebraic operations (!)
  - $\quad \mathsf{let}\ x = \big(\mathsf{promise}\ (\mathrm{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N\big)\ \mathsf{in}\ L$
    $\leadsto \mathsf{promise}\ (\mathrm{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \big(\mathsf{let}\ x = N\ \mathsf{in}\ L\big)$

$\lambda_{æ}$-calculus: interrupt handlers

- Allow computations to react to interrupts

$$\frac{\textsc{Ty-Comp-Promise} \quad \iota\,(\mathsf{op}) = (o', \iota') \qquad \Gamma, x : A_{op} \vdash M : \langle X \rangle \,!\, (o', \iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y \,!\, (o, \iota)}$$

where $p : \langle X \rangle$ is a promise-typed variable

- Operationally behave like (scoped) algebraic operations (!)
  - $\quad \mathsf{let}\ x = \big(\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N\big)\ \mathsf{in}\ L$
    $\rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \big(\mathsf{let}\ x = N\ \mathsf{in}\ L\big)$
  - $\quad \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \uparrow \mathsf{op}'\,(V, N)$
    $\rightsquigarrow\ \uparrow \mathsf{op}'\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N)$

$\lambda_{æ}$-calculus: interrupt handlers

- Allow computations to react to interrupts

TY-COMP-PROMISE

$$\frac{\iota\,(\mathsf{op}) = (o', \iota') \qquad \Gamma, x : A_{\mathsf{op}} \vdash M : \langle X \rangle \,!\, (o', \iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y \,!\, (o, \iota)}$$

where $p : \langle X \rangle$ is a promise-typed variable

- Operationally behave like (scoped) algebraic operations (!)
  - $\mathsf{let}\ x = \big(\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N\big)\ \mathsf{in}\ L$
    $\rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ (\mathsf{let}\ x = N\ \mathsf{in}\ L)$
  - $\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ {\uparrow}\,\mathsf{op}'\,(V, N)$ (type safety!)
    $\rightsquigarrow {\uparrow}\,\mathsf{op}'\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N)$ $(p \notin FV(V))$

$\lambda_{æ}$-calculus: interrupt handlers

- Allow computations to react to interrupts

$$\dfrac{\iota\,(\mathsf{op}) = (o', \iota') \qquad \Gamma, x : A_{op} \vdash M : \langle X \rangle \,!\, (o', \iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y \,!\, (o, \iota)}\ \textsc{Ty-Comp-Promise}$$

  where $p : \langle X \rangle$ is a promise-typed variable

- They are triggered by matching interrupts
  - $\downarrow \mathsf{op}\,(W, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N)$
    $\rightsquigarrow \mathsf{let}\ p = M[W/x]\ \mathsf{in} \downarrow \mathsf{op}\,(W, N)$

$\lambda_{æ}$-calculus: interrupt handlers

- Allow computations to react to interrupts

$$\frac{\textsc{Ty-Comp-Promise} \quad \iota\,(\mathsf{op}) = (o', \iota') \qquad \Gamma, x : A_{\mathsf{op}} \vdash M : \langle X \rangle \,!\, (o', \iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathtt{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathtt{as}\ p\ \mathtt{in}\ N : Y \,!\, (o, \iota)}$$

where $p : \langle X \rangle$ is a promise-typed variable

- They are triggered by matching interrupts
  - $\downarrow \mathsf{op}\,(W, \mathtt{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathtt{as}\ p\ \mathtt{in}\ N)$
    $\rightsquigarrow \mathtt{let}\ p = M[W/x]\ \mathtt{in} \downarrow \mathsf{op}\,(W, N)$

- And non-matching interrupts ($\mathsf{op} \neq \mathsf{op}'$) are passed through
  - $\downarrow \mathsf{op}\,(W, \mathtt{promise}\ (\mathsf{op}'\ x \mapsto M)\ \mathtt{as}\ p\ \mathtt{in}\ N)$
    $\rightsquigarrow \mathtt{promise}\ (\mathsf{op}'\ x \mapsto M)\ \mathtt{as}\ p\ \mathtt{in} \downarrow \mathsf{op}\,(W, N)$

$\lambda_{æ}$-calculus: interrupt handlers

- Allow computations to react to interrupts

$$\frac{\textsc{Ty-Comp-Promise} \quad \iota\,(\mathsf{op}) = (o', \iota') \qquad \Gamma, x : A_{\mathsf{op}} \vdash M : \langle X \rangle \,!\, (o', \iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathtt{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathtt{as}\ p\ \mathtt{in}\ N : Y \,!\, (o, \iota)}$$

where $p : \langle X \rangle$ is a promise-typed variable

- They also do not block their continuations
  - $N \leadsto N'$ $\Longrightarrow$ $\mathtt{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathtt{as}\ p\ \mathtt{in}\ N$ $\leadsto \mathtt{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathtt{as}\ p\ \mathtt{in}\ N'$

$\lambda_{æ}$-calculus: interrupt handlers

- Allow computations to react to interrupts

$$\frac{\iota\,(\mathsf{op}) = (o', \iota') \qquad \Gamma, x : A_{op} \vdash M : \langle X \rangle \,!\, (o', \iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y \,!\, (o, \iota)}\ \textsc{Ty-Comp-Promise}$$

where $p : \langle X \rangle$ is a promise-typed variable

- They also do not block their continuations
    - $N \leadsto N'$

$$\Longrightarrow$$
$$\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$
$$\leadsto \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N'$$

For type safety, important that p does not get an arbitrary type!

$\lambda_{æ}$-calculus: interrupt handlers

- To remove general recursion from $\lambda_{æ}$, we extend int. handlers by
  - allowing them to reinstall themselves
  - allowing them to pass state between triggerings

$$M, N \;::=\; \cdots \;\Big|\; \texttt{promise}\ (\mathsf{op}\ x\ \boxed{r}\ \boxed{s} \mapsto M)\ @_S\ \boxed{V}\ \texttt{as}\ p\ \texttt{in}\ N$$

$\lambda_{æ}$-calculus: interrupt handlers

- To remove general recursion from $\lambda_{æ}$, we extend int. handlers by
  - allowing them to reinstall themselves
  - allowing them to pass state between triggerings

$$M, N \;::=\; \cdots \;\Big|\; \mathsf{promise}\,(\mathsf{op}\ x\ \boxed{r}\ \boxed{s} \mapsto M)\ @_S\ \boxed{V}\ \mathsf{as}\ p\ \mathsf{in}\ N$$

- Operationally only difference in how they trigger
  - $\downarrow\mathsf{op}\,(W, \mathsf{promise}\,(\mathsf{op}\ x\ r\ s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N)$

    $\rightsquigarrow \mathsf{let}\ p = M[W/x, \boxed{R/r}, \boxed{V/s}]\ \mathsf{in}\ \downarrow\mathsf{op}\,(W, N)$

    where

$$R \;\stackrel{\mathrm{def}}{=}\; \mathsf{fun}\ \boxed{s'} \mapsto \mathsf{promise}\,(\mathsf{op}\ x\ r\ s \mapsto M)\ @_S\ \boxed{s'}\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p$$

$\lambda_{æ}$-calculus: awaiting

- Enables programmers to selectively block execution

$$\textsc{TyComp-Await}\quad \frac{\Gamma \vdash V : \langle X \rangle \qquad \Gamma, x : X \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N : Y \,!\, (o, \iota)}$$

$\lambda_{æ}$-calculus: awaiting

- Enables programmers to selectively block execution

$$\textsc{TyComp-Await}\quad \frac{\Gamma \vdash V : \langle X \rangle \qquad \Gamma, x : X \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N : Y \,!\, (o, \iota)}$$

- Behaves like pattern-matching (and also like alg. ops.)
  - $\mathsf{await}\ \langle V \rangle\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N \rightsquigarrow N[V/x]$
  - $\ \mathsf{let}\ y = (\mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M)\ \mathsf{in}\ N$
    $\rightsquigarrow \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ (\mathsf{let}\ y = M\ \mathsf{in}\ N)$

- In contrast to earlier gadgets, await blocks its cont.'s execution !!!

$\lambda_{æ}$-calculus: environment

- We model the environment by running computations in parallel

$$P, Q \ ::= \ \mathsf{run}\ M \ \mid \ P \parallel Q \ \mid \ \uparrow \mathsf{op}\,(V, P) \ \mid \ \downarrow \mathsf{op}\,(W, P)$$

$\lambda_{æ}$-calculus: environment

- We model the environment by running computations in parallel

$$P, Q \;::=\; \mathsf{run}\ M \;\mid\; P \,||\, Q \;\mid\; \uparrow \mathsf{op}\,(V, P) \;\mid\; \downarrow \mathsf{op}\,(W, P)$$

- Small-step operational semantics $P \rightsquigarrow Q$: congruence rules +
  - $\mathsf{run}\ (\uparrow \mathsf{op}\,(V, M)) \rightsquigarrow \uparrow \mathsf{op}\,(V, \mathsf{run}\ M)$

$\lambda_{\text{æ}}$-calculus: environment

- We model the environment by running computations in parallel

$$P, Q \;::=\; \mathsf{run}\ M \;\mid\; P \,||\, Q \;\mid\; \uparrow \mathsf{op}\,(V, P) \;\mid\; \downarrow \mathsf{op}\,(W, P)$$

- Small-step operational semantics $P \rightsquigarrow Q$: congruence rules +
  - $\mathsf{run}\ (\uparrow \mathsf{op}\,(V, M)) \rightsquigarrow \uparrow \mathsf{op}\,(V, \mathsf{run}\ M)$
  - $(\uparrow \mathsf{op}\,(V, P)) \,||\, Q \rightsquigarrow \uparrow \mathsf{op}\,(V, (P \,||\, \downarrow \mathsf{op}\,(V, Q)))$ (broadcast)
  - $P \,||\, (\uparrow \mathsf{op}\,(V, Q)) \rightsquigarrow \uparrow \mathsf{op}\,(V, (\downarrow \mathsf{op}\,(V, P) \,||\, Q))$ (broadcast)

$\lambda_{æ}$-calculus: environment

- We model the environment by running computations in parallel

$$P, Q \; ::= \; \mathsf{run}\ M \;\mid\; P \,||\, Q \;\mid\; \uparrow \mathsf{op}\,(V, P) \;\mid\; \downarrow \mathsf{op}\,(W, P)$$

- Small-step operational semantics $P \rightsquigarrow Q$: congruence rules +
  - $\mathsf{run}\ (\uparrow \mathsf{op}\,(V, M)) \rightsquigarrow \uparrow \mathsf{op}\,(V, \mathsf{run}\ M)$
  - $(\uparrow \mathsf{op}\,(V, P)) \,||\, Q \rightsquigarrow \uparrow \mathsf{op}\,(V, (P \,||\, \downarrow \mathsf{op}\,(V, Q)))$ (broadcast)
  - $P \,||\, (\uparrow \mathsf{op}\,(V, Q)) \rightsquigarrow \uparrow \mathsf{op}\,(V, (\downarrow \mathsf{op}\,(V, P) \,||\, Q))$ (broadcast)
  - $\downarrow \mathsf{op}\,(W, \mathsf{run}\ M) \rightsquigarrow \mathsf{run}\ (\downarrow \mathsf{op}\,(W, M))$
  - ...

$\lambda_{æ}$-calculus: environment

- Compared to POPL'21, modal types give us a type-safe spawn

$$M, N \ ::= \ \cdots \ \mid \ \mathsf{spawn}\,(M, N)$$

$$\textsc{Ty-Comp-Spawn}\quad \frac{\Gamma, \text{🔓} \vdash M : X \,!\, (o', \iota') \qquad \Gamma \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{spawn}\,(M, N) : Y \,!\, (o, \iota)}$$

$\lambda_{æ}$-calculus: environment

- Compared to POPL'21, modal types give us a type-safe spawn

$$M, N ::= \cdots \mid \mathsf{spawn}\,(M, N)$$

$$\frac{\Gamma, 🔓 \vdash M : X \,!\, (o', \iota') \qquad \Gamma \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{spawn}\,(M, N) : Y \,!\, (o, \iota)}\ \textsc{Ty-Comp-Spawn}$$

- Operationally propagates outwards (like a scoped alg. op.)
  - $\mathsf{let}\ x = \big(\mathsf{spawn}\,(M_1, M_2)\big)\ \mathsf{in}\ N \rightsquigarrow \mathsf{spawn}\,(M_1, \mathsf{let}\ x = M_2\ \mathsf{in}\ N)$
  - also propagates through promises, where 🔓 provides type-safety

$\lambda_{\text{æ}}$-calculus: environment

- Compared to POPL'21, modal types give us a type-safe spawn

$$M, N \ ::= \ \cdots \ \mid \ \mathsf{spawn}\,(M, N)$$

$$\textsc{Ty-Comp-Spawn}\quad \frac{\Gamma, 🔓 \vdash M : X \,!\, (o', \iota') \qquad \Gamma \vdash N : Y \,!\, (o, \iota)}{\Gamma \vdash \mathsf{spawn}\,(M, N) : Y \,!\, (o, \iota)}$$

- Operationally propagates outwards (like a scoped alg. op.)
  - $\mathsf{let}\ x = \big(\mathsf{spawn}\,(M_1, M_2)\big)\ \mathsf{in}\ N \rightsquigarrow \mathsf{spawn}\,(M_1, \mathsf{let}\ x = M_2\ \mathsf{in}\ N)$
  - also propagates through promises, where 🔓 provides type-safety
- Eventually gives rise to a new parallel process
  - $\mathsf{run}\ (\mathsf{spawn}\,(M, N)) \rightsquigarrow \mathsf{run}\ M \mid\mid \mathsf{run}\ N$

$\lambda_{æ}$-calculus: environment

- Compared to POPL'21, modal types give us a type-safe spawn

$$M, N \;::=\; \cdots \;\Big|\; \mathsf{spawn}\,(M, N)$$

$$\textsc{Ty-Comp-Spawn}\;\frac{\Gamma, 🔓 \vdash M : X\,!\,(o', \iota') \qquad \Gamma \vdash N : Y\,!\,(o, \iota)}{\Gamma \vdash \mathsf{spawn}\,(M, N) : Y\,!\,(o, \iota)}$$

- Operationally propagates outwards (like a scoped alg. op.)
  - $\mathsf{let}\; x = \big(\mathsf{spawn}\,(M_1, M_2)\big)\;\mathsf{in}\; N \rightsquigarrow \mathsf{spawn}\,(M_1, \mathsf{let}\; x = M_2\;\mathsf{in}\; N)$
  - also propagates through promises, where 🔓 provides type-safety
- Eventually gives rise to a new parallel process
  - $\mathsf{run}\;(\mathsf{spawn}\,(M, N)) \rightsquigarrow \mathsf{run}\; M \,||\, \mathsf{run}\; N$
- Importantly, does not block its continuation !!!

Examples

Examples

- Multi-party web application
- Remote function call execution
- (Simulating) cancellations of remote function calls
- Preemptive multi-threading
- Parallel variant of runners of algebraic effects
- Non-blocking post-processing of promised values
- ...

Example: implementing algebraic ops.

- Algebraic operations $\mathsf{op}\,(V, y.M)$ are implemented at call site as

$$\uparrow \mathsf{op\text{-}req}\,\big(V, \mathsf{promise}\,(\mathsf{op\text{-}resp}\ y \mapsto \mathsf{return}\,\langle y\rangle)\ \mathsf{as}\ p\ \mathsf{in}$$
$$\mathsf{await}\ p\ \mathsf{until}\,\langle y\rangle\ \mathsf{in}\ M\big)$$

Example: implementing algebraic ops.

- Algebraic operations op $(V, y.M)$ are implemented at call site as

$$\uparrow \mathsf{op\text{-}req}\,\big(V, \mathsf{promise}\ (\mathsf{op\text{-}resp}\ y \mapsto \mathsf{return}\,\langle y\rangle)\ \mathsf{as}\ p\ \mathsf{in}$$
$$\mathsf{await}\ p\ \mathsf{until}\ \langle y\rangle\ \mathsf{in}\ M\big)$$

- The corresponding implementation using a recursively defined interrupt handler for op-req interrupt (in some other process)

$$\mathsf{promise}\ (\mathsf{op\text{-}req}\ x\ r \mapsto \mathsf{let}\ y = M\ \mathsf{in}$$
$$\uparrow \mathsf{op\text{-}resp}\,(y, r\,())$$
$$)\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p$$

Example: implementing algebraic ops.

- Algebraic operations $\mathsf{op}\,(V, y.M)$ are implemented at call site as

$$\uparrow \mathsf{op\text{-}req}\,\big(V, \mathsf{promise}\;(\mathsf{op\text{-}resp}\;y \mapsto \mathsf{return}\;\langle y\rangle)\;\mathsf{as}\;p\;\mathsf{in}\\ \mathsf{await}\;p\;\mathsf{until}\;\langle y\rangle\;\mathsf{in}\;M\big)$$

- The corresponding implementation using a recursively defined interrupt handler for op-req interrupt (in some other process)

$$\mathsf{promise}\;(\mathsf{op\text{-}req}\;x\;r \mapsto \mathsf{let}\;y = M\;\mathsf{in}\\ \uparrow \mathsf{op\text{-}resp}\,(y, r\,())\\ )\;\mathsf{as}\;p\;\mathsf{in}\;\mathsf{return}\;p$$

- The interaction happens then via parallel composition

$$M_{\mathsf{call\text{-}site}} \;||\; M_{\mathsf{op\text{-}implementation}}$$

Example: preemptive multi-threading

- We consider two interrupts: $\text{stop} : 1$ and $\text{go} : 1$

Example: preemptive multi-threading

- We consider two interrupts: stop : 1 and go : 1
- We define the following recursively defined interrupt handler

```
let waitForStop () =
  promise (stop _ r ↦
     promise (go _ _ ↦ return ⟨()⟩) as p in
     await p until ⟨_⟩ in r ()
  ) as p' in return p'
```

Example: preemptive multi-threading

- We consider two interrupts: $\mathsf{stop} : 1$ and $\mathsf{go} : 1$
- We define the following recursively defined interrupt handler

```
let waitForStop () =
  promise (stop _ r ↦
     promise (go _ _ ↦ return ⟨()⟩) as p in
     await p until ⟨_⟩ in r ()
  ) as p' in return p'
```

- We initialise the preemtive behaviour by running

```
waitForStop (); comp
```

Example: preemptive multi-threading

- We consider two interrupts: $\mathsf{stop} : 1$ and $\mathsf{go} : 1$
- We define the following recursively defined interrupt handler

```
let waitForStop () =
  promise (stop _ r ↦
     promise (go _ _ ↦ return ⟨()⟩) as p in
     await p until ⟨_⟩ in r ()
  ) as p' in return p'
```

- We initialise the preemtive behaviour by running

```
waitForStop (); comp
```

- Then $\downarrow \mathsf{stop}\,((), \mathsf{waitForStop}(); \mathsf{comp})$

Example: preemptive multi-threading

- We consider two interrupts: $\mathsf{stop} : 1$ and $\mathsf{go} : 1$
- We define the following recursively defined interrupt handler

```
let waitForStop () =
  promise (stop _ r ↦
    promise (go _ _ ↦ return ⟨()⟩) as p in
    await p until ⟨_⟩ in r ()
  ) as p' in return p'
```

- We initialise the preemtive behaviour by running

```
waitForStop (); comp
```

- Then

$$\begin{aligned} & \downarrow \mathsf{stop}\,((), \mathsf{waitForStop}(); \mathsf{comp}) \\ \leadsto^* & \downarrow \mathsf{stop}\,((), \mathsf{waitForStop}(); \mathsf{comp}') \end{aligned}$$

Example: preemptive multi-threading

- We consider two interrupts: stop : 1 and go : 1
- We define the following recursively defined interrupt handler

```
let waitForStop () =
  promise (stop _ r ↦
    promise (go _ _ ↦ return ⟨()⟩) as p in
    await p until ⟨_⟩ in r ()
  ) as p' in return p'
```

- We initialise the preemtive behaviour by running

```
waitForStop (); comp
```

- Then

$$\begin{aligned} & \downarrow \mathsf{stop}\,((), \mathsf{waitForStop}(); \mathsf{comp}) \\ \rightsquigarrow^* & \downarrow \mathsf{stop}\,((), \mathsf{waitForStop}(); \mathsf{comp}') \\ \rightsquigarrow^* & \downarrow \mathsf{stop}\,((), \mathsf{promise}\ (\mathsf{stop}\ \_\ r \mapsto \ldots)\ \mathsf{as}\ p'\ \mathsf{in}\ \mathsf{comp}') \end{aligned}$$

Example: preemptive multi-threading

- We consider two interrupts: stop : 1 and go : 1
- We define the following recursively defined interrupt handler

```
let waitForStop () =
  promise (stop _ r ↦
    promise (go _ _ ↦ return ⟨()⟩) as p in
    await p until ⟨_⟩ in r ()
  ) as p' in return p'
```

- We initialise the preemtive behaviour by running

```
waitForStop (); comp
```

- Then

$$
\begin{aligned}
& \downarrow \mathsf{stop}\,((), \mathsf{waitForStop}(); \mathsf{comp}) \\
\rightsquigarrow^* \; & \downarrow \mathsf{stop}\,((), \mathsf{waitForStop}(); \mathsf{comp}') \\
\rightsquigarrow^* \; & \downarrow \mathsf{stop}\,((), \mathsf{promise}\ (\mathsf{stop}\ \_\ r \mapsto \ldots)\ \mathsf{as}\ p'\ \mathsf{in}\ \mathsf{comp}') \\
\rightsquigarrow \; & \mathsf{promise}\ (\mathsf{go}\ \_\ \_ \mapsto \mathsf{return}\ \langle()\rangle)\ \mathsf{as}\ p\ \mathsf{in} \\
& \mathsf{await}\ p\ \mathsf{until}\ \langle\_\rangle\ \mathsf{in} \\
& \mathsf{promise}\ (\mathsf{stop}\ \_\ r \mapsto \ldots)\ \mathsf{as}\ p'\ \mathsf{in} \downarrow \mathsf{stop}\,((), \mathsf{comp}')
\end{aligned}
$$

Example: post-processing promised values

- As syntactic sugar (relies on propagating signals into conts.)

```
process_op p with (⟨x⟩ ↦ comp) as q in cont
=
promise (op _ ↦ await p until ⟨x⟩ in
                let y = comp in
                return ⟨y⟩) as q in cont
```

Example: post-processing promised values

- As syntactic sugar (relies on propagating signals into conts.)

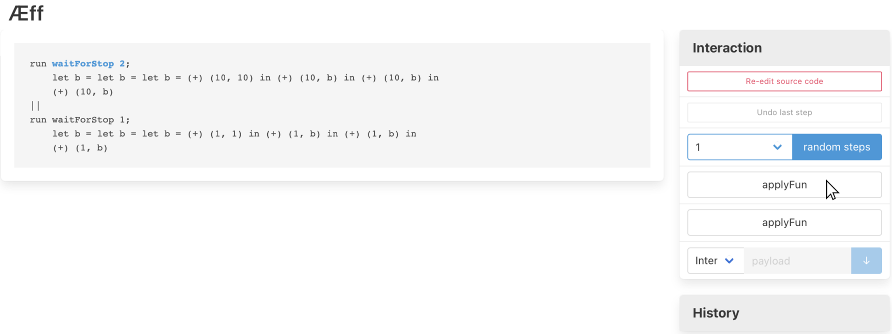
```
process_op p with (⟨x⟩ ↦ comp) as q in cont
=
promise (op _ ↦ await p until ⟨x⟩ in
                let y = comp in
                return ⟨y⟩) as q in cont
```

- E.g., we can then post-process a promised list in non-blocking way

```
promise (op x ↦ original_interrupt_handler) as p in
...
process_op p with (⟨is⟩ ↦ filter (fun i ↦ i > 0) is) as q in
process_op q with (⟨js⟩ ↦ fold (fun j j' ↦ j * j') 1 js) as r in
process_op r with (⟨k⟩ ↦ ↑ productOfPositiveElements k) as _ in
...
```

Æff web interface

https://matija.pretnar.info/aeff/



Conclusion

Conclusion

- $\lambda_{æ}$: a core calculus for asynchronous algebraic effects
  - based on decoupling the execution of alg. operation calls
  - teaches us that preemptive behaviour = interrupts = eff. handling
  - more details in the papers and Agda formalisations

Conclusion

- $\lambda_{æ}$: a core calculus for asynchronous algebraic effects
  - based on decoupling the execution of alg. operation calls
  - teaches us that preemptive behaviour = interrupts = eff. handling
  - more details in the papers and Agda formalisations
- Some ongoing work on $\lambda_{æ}$'s denotational semantics
  - requires factorisation of morphisms $\langle X \rangle \longrightarrow A$ through 1
  - presheaf categories give a suitable playground
  - signals, promises, awaits as alg. ops. / interrupts as handling

Conclusion

- $\lambda_{æ}$: a core calculus for asynchronous algebraic effects
  - based on decoupling the execution of alg. operation calls
  - teaches us that preemptive behaviour = interrupts = eff. handling
  - more details in the papers and Agda formalisations

- Some ongoing work on $\lambda_{æ}$'s denotational semantics
  - requires factorisation of morphisms $\langle X \rangle \longrightarrow A$ through 1
  - presheaf categories give a suitable playground
  - signals, promises, awaits as alg. ops. / interrupts as handling

- Some ongoing work on $\lambda_{æ}$'s normalisation ($\top\top$-lifting style)
  - seq. part with non-reinstallable int. handlers ✓
  - par. part with non-reinstallable int. handlers (maybe ✓)
  - seq. part with reinstallable int. handlers (naively ✗, but hope ✓)
  - par. part with reinstallable int. handlers ✗

asynchronous operation calls

=

signals + **interrupts** + **interrupt handlers**

(unary ops.)		(effect handling)		(scoped ops. + modalities)