

# (Higher-Order) Asynchronous Effects

Danel Ahman   Matija Pretnar   Janez Radešček

University of Ljubljana, Slovenia

July ???, 2021 @ ~~Dagstuhl~~ Online

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 834146.



This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA9550-17-1-0326 and FA9550-21-1-0024.

# Today's Plan

- Synchrony of algebraic effects
- Asynchrony through decoupling operation calls
- $\lambda_{\text{æ}}$ -calculus
- Examples

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

<https://github.com/matijapretnar/aeff>

<https://github.com/danelahman/aeff-agda>

- Some recent extensions (the higher-order part of the talk's title)

# Synchrony of algebraic effects

# Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$\dots \rightsquigarrow \text{op} (V, y.M)$$

# Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$\begin{array}{c} M_{\text{op}}[V/x] \\ \uparrow \\ \text{signal op's implementation} \\ \dots \rightsquigarrow \text{op}(V, y.M) \end{array}$$

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

# Synchrony of algebraic effects

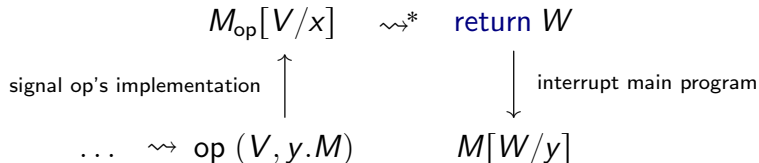
- The conventional operational treatment of algebraic effects

$$\begin{array}{ccc} M_{\text{op}}[V/x] & \rightsquigarrow^* & \text{return } W \\ \text{signal op's implementation} \uparrow & & \\ \dots \rightsquigarrow \text{op}(V, y.M) & & \end{array}$$

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

# Synchrony of algebraic effects

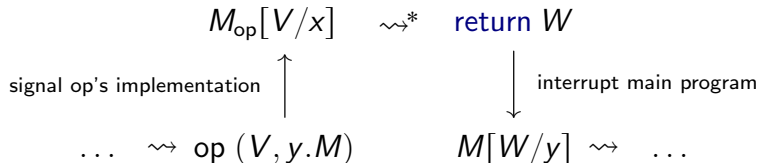
- The conventional operational treatment of algebraic effects



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

# Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

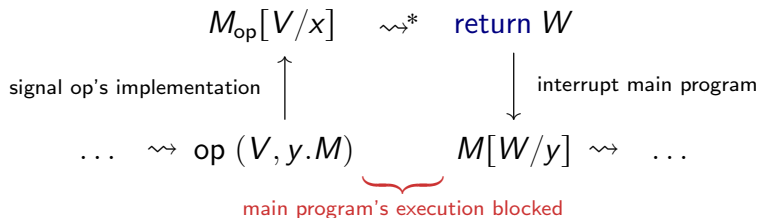


- $M_{\text{op}}$  - handler, runner, top-level default implementation,  $\dots$



# Synchrony of algebraic effects

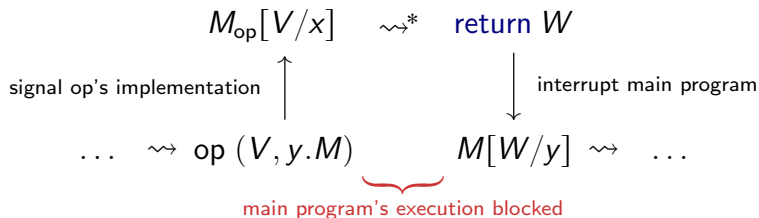
- The conventional operational treatment of algebraic effects



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

# Synchrony of algebraic effects

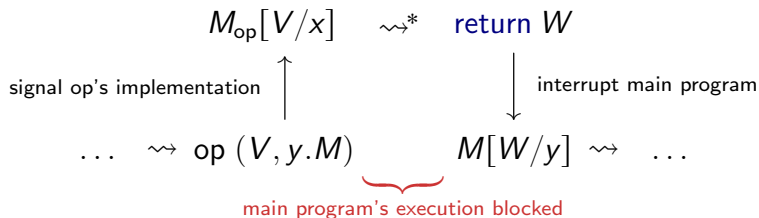
- The conventional operational treatment of algebraic effects



- $M_{\text{op}}$  - handler, runner, top-level default implementation, ...
- Forces **all uses** of algebraic effects to be synchronous

# Synchrony of algebraic effects

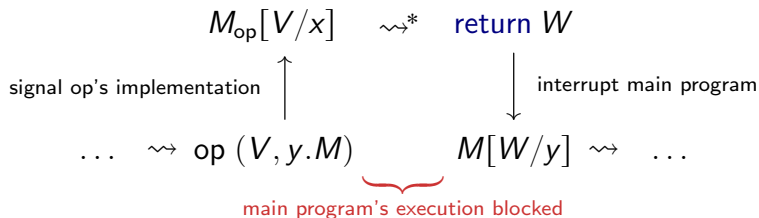
- The conventional operational treatment of algebraic effects



- $M_{\text{op}}$  - handler, runner, top-level default implementation, ...
- Forces **all uses** of algebraic effects to be synchronous
- Existing langs. do async. by **delegating** it to their lang. backends

# Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects



- $M_{\text{op}}$  - handler, runner, top-level default implementation, ...
- Forces **all uses** of algebraic effects to be synchronous
- Existing langs. do async. by **delegating** it to their lang. backends
- In contrast, we capture async. in a **self-contained core calculus**

$\lambda_{\text{æ}}$ -calculus

# $\lambda_{\text{æ}}$ -calculus: basics

- Extension of Levy's fine-grain call-by-value  $\lambda$ -calculus
- **Types:**  $X, Y ::= b \mid \dots \mid X \rightarrow Y ! (o, \iota) \mid \dots$
- **Values:**  $V, W ::= x \mid \dots \mid \text{fun } (x : X) \mapsto M \mid \dots$
- **Computations:**  $M, N ::= \text{return } V \mid \text{let } x = M \text{ in } N \mid \dots$
- **Typing judgements:**  $\Gamma \vdash V : X \qquad \Gamma \vdash M : X ! (o, \iota)$
- **Small-step operational semantics:**  $M \rightsquigarrow N$

## $\lambda_{\text{æ}}$ -calculus: signals

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

## $\lambda_{\text{æ}}$ -calculus: signals

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

$$\frac{\text{TYCOMP-SIGNAL} \quad \text{op} : A_{\text{op}} \in o \quad \Gamma \vdash V : A_{\text{op}} \quad \Gamma \vdash M : X!(o, \iota)}{\Gamma \vdash \uparrow \text{op}(V, M) : X!(o, \iota)}$$

where  $A_{\text{op}}$  is a ground type (prod. and sum of base types)



## $\lambda_{\text{æ}}$ -calculus: signals

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

$$\frac{\text{TYCOMP-SIGNAL} \quad \text{op} : A_{\text{op}} \in \mathcal{O} \quad \Gamma \vdash V : A_{\text{op}} \quad \Gamma \vdash M : X!(o, \iota)}{\Gamma \vdash \uparrow \text{op}(V, M) : X!(o, \iota)}$$

where  $A_{\text{op}}$  is a ground type (prod. and sum of base types)

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

# $\lambda_{\text{æ}}$ -calculus: signals

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

$$\frac{\text{TYCOMP-SIGNAL} \quad \text{op} : A_{\text{op}} \in \mathcal{O} \quad \Gamma \vdash V : A_{\text{op}} \quad \Gamma \vdash M : X ! (o, \iota)}{\Gamma \vdash \uparrow \text{op} (V, M) : X ! (o, \iota)}$$

where  $A_{\text{op}}$  is a ground type (prod. and sum of base types)

- Operationally behave like algebraic operations
  - $\text{let } x = \uparrow \text{op} (V, M) \text{ in } N \rightsquigarrow \uparrow \text{op} (V, \text{let } x = M \text{ in } N)$
- But importantly, they do not block their continuations
  - $M \rightsquigarrow M' \implies \uparrow \text{op} (V, M) \rightsquigarrow \uparrow \text{op} (V, M')$

## $\lambda_{\text{æ}}$ -calculus: interrupts

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

$$\text{TYCOMP-INTERRUPT} \quad \frac{\Gamma \vdash V : A_{\text{op}} \quad \Gamma \vdash M : X ! (o, \iota)}{\Gamma \vdash \downarrow_{\text{op}}(W, M) : X ! (\text{op} \downarrow (o, \iota))}$$

where op acts on the effect annotations in conclusion

# $\lambda_{\text{æ}}$ -calculus: interrupts

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

$$\frac{\text{TYCOMP-INTERRUPT} \quad \Gamma \vdash V : A_{\text{op}} \quad \Gamma \vdash M : X ! (o, \iota)}{\Gamma \vdash \downarrow \text{op} (W, M) : X ! (\text{op} \downarrow (o, \iota))}$$

where op acts on the effect annotations in conclusion

- Operationally behave like homomorphisms/effect handling
  - $\downarrow \text{op} (W, \text{return } V) \rightsquigarrow \text{return } V$
  - $\downarrow \text{op} (W, \uparrow \text{op}' (V, M)) \rightsquigarrow \uparrow \text{op}' (V, \downarrow \text{op} (W, M))$
  - ...
- And they also do not block their continuations
  - $M \rightsquigarrow M' \implies \downarrow \text{op} (V, M) \rightsquigarrow \downarrow \text{op} (V, M')$

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

- Allow computation to react to interrupts

TY-COMP-PROMISE

$$\frac{\begin{array}{c} \iota(\text{op}) = (o', \iota') \\ \Gamma, x : A_{\text{op}} \vdash M : \langle X \rangle ! (o', \iota') \quad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota) \end{array}}{\Gamma \vdash \text{promise } (\text{op } x \mapsto M) \text{ as } p \text{ in } N : Y ! (o, \iota)}$$

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

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

- Allow computation to react to interrupts

TY-COMP-PROMISE

$$\frac{\begin{array}{c} \iota(\text{op}) = (o', \iota') \\ \Gamma, x : A_{\text{op}} \vdash M : \langle X \rangle ! (o', \iota') \quad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota) \end{array}}{\Gamma \vdash \text{promise} (\text{op } x \mapsto M) \text{ as } p \text{ in } N : Y ! (o, \iota)}$$

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

- Operationally **behave like (scoped) algebraic operations (!)**
  - $\text{let } x = (\text{promise} (\text{op } x \mapsto M_1) \text{ as } p \text{ in } M_2) \text{ in } N$   
 $\rightsquigarrow \text{promise} (\text{op } x \mapsto M_1) \text{ as } p \text{ in } (\text{let } x = M_2 \text{ in } N)$
  - $\text{promise} (\text{op } x \mapsto M) \text{ as } p \text{ in } \uparrow \text{op} (V, N)$  (type safety!)  
 $\rightsquigarrow \uparrow \text{op} (V, \text{promise} (\text{op } x \mapsto M) \text{ as } p \text{ in } N)$  ( $p \notin FV(V)$ )

# $\lambda_{\text{æ}}$ -calculus: interrupt handlers ctd.

- Allow computation to react to interrupts

TY-COMP-PROMISE

$$\frac{\begin{array}{c} \iota(\text{op}) = (o', \iota') \\ \Gamma, x : A_{op} \vdash M : \langle X \rangle ! (o', \iota') \quad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota) \end{array}}{\Gamma \vdash \text{promise } (\text{op } x \mapsto M) \text{ as } p \text{ in } N : Y ! (o, \iota)}$$

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

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

# $\lambda_{\text{æ}}$ -calculus: interrupt handlers ctd.

- Allow computation to react to interrupts

TY-COMP-PROMISE

$$\frac{\begin{array}{c} \iota(\text{op}) = (o', \iota') \\ \Gamma, x : A_{\text{op}} \vdash M : \langle X \rangle ! (o', \iota') \quad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota) \end{array}}{\Gamma \vdash \text{promise } (\text{op } x \mapsto M) \text{ as } p \text{ in } N : Y ! (o, \iota)}$$

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

- They are triggered by matching interrupts
  - $\downarrow \text{op } (W, \text{promise } (\text{op } x \mapsto M) \text{ as } p \text{ in } N)$   
 $\rightsquigarrow \text{let } p = M[W/x] \text{ in } \downarrow \text{op } (W, N)$
- And **non-matching interrupts** ( $\text{op} \neq \text{op}'$ ) are passed through
  - $\downarrow \text{op } (W, \text{promise } (\text{op}' x \mapsto M) \text{ as } p \text{ in } N)$   
 $\rightsquigarrow \text{promise } (\text{op}' x \mapsto M) \text{ as } p \text{ in } \downarrow \text{op } (W, N)$



# $\lambda_{\text{æ}}$ -calculus: interrupt handlers ctd.

- Allow computation to react to interrupts

TY-COMP-PROMISE

$$\frac{\begin{array}{c} \iota(\text{op}) = (o', \iota') \\ \Gamma, x : A_{\text{op}} \vdash M : \langle X \rangle ! (o', \iota') \quad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota) \end{array}}{\Gamma \vdash \text{promise } (\text{op } x \mapsto M) \text{ as } p \text{ in } N : Y ! (o, \iota)}$$

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

- They also **do not block their continuations**

$$\bullet \quad N \rightsquigarrow N'$$

$\implies$

**promise** (op  $x \mapsto M$ ) **as**  $p$  **in**  $N$

$\rightsquigarrow$  **promise** (op  $x \mapsto M$ ) **as**  $p$  **in**  $N'$

# $\lambda_{\text{æ}}$ -calculus: interrupt handlers ctd.

- Allow computation to react to interrupts

TY-COMP-PROMISE

$$\frac{\begin{array}{c} \iota(\text{op}) = (o', \iota') \\ \Gamma, x : A_{\text{op}} \vdash M : \langle X \rangle ! (o', \iota') \quad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota) \end{array}}{\Gamma \vdash \text{promise } (\text{op } x \mapsto M) \text{ as } p \text{ in } N : Y ! (o, \iota)}$$

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

- They also **do not block their continuations**

$$\bullet N \rightsquigarrow N'$$

$\implies$

$$\text{promise } (\text{op } x \mapsto M) \text{ as } p \text{ in } N$$

$$\rightsquigarrow \text{promise } (\text{op } x \mapsto M) \text{ as } p \text{ in } N'$$

- For type safety, important that  **$p$  does not get an arbitrary type**

## $\lambda_{\text{æ}}$ -calculus: awaiting

- Enables programmers to selectively block execution

$$\frac{\text{TYCOMP-AWAIT} \quad \Gamma \vdash V : \langle X \rangle \quad \Gamma, x : X \vdash N : Y ! (o, \iota)}{\Gamma \vdash \text{await } V \text{ until } \langle x \rangle \text{ in } N : Y ! (o, \iota)}$$

# $\lambda_{\text{æ}}$ -calculus: awaiting

- Enables programmers to selectively block execution

$$\frac{\text{TYCOMP-AWAIT} \quad \Gamma \vdash V : \langle X \rangle \quad \Gamma, x : X \vdash N : Y ! (o, \iota)}{\Gamma \vdash \text{await } V \text{ until } \langle x \rangle \text{ in } N : Y ! (o, \iota)}$$

- Operationally behave like pattern-matching (and alg. ops.)
  - $\text{await } \langle V \rangle \text{ until } \langle x \rangle \text{ in } N \rightsquigarrow N[V/x]$
  - $\text{let } y = (\text{await } V \text{ until } \langle x \rangle \text{ in } M) \text{ in } N$   
 $\rightsquigarrow \text{await } V \text{ until } \langle x \rangle \text{ in } (\text{let } y = M \text{ in } N)$
- In contrast to earlier gadgets, await blocks its cont.'s execution (!)

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

- We model the environment by running computations in parallel

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

(omitting typing judgement, typing rules, and type reduction)

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

- We model the environment by running computations in parallel

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

(omitting typing judgement, typing rules, and type reduction)

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

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

- We model the environment by running computations in parallel

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

(omitting typing judgement, typing rules, and type reduction)

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

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

- We model the environment by running computations in parallel

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

(omitting typing judgement, typing rules, and type reduction)

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



# Examples

# Example: guarded interrupt handlers

- In examples we often write

```
promise (op x when guard  $\mapsto$  comp) as p in cont
```

as a syntactic sugar for the recursively defined interrupt handler

```
let rec waitForGuard () =  
  promise (op x  $\mapsto$  if guard then comp else waitForGuard ()) as p' in return p'  
in  
let p = waitForGuard () in cont
```

# Example: guarded interrupt handlers

- In examples we often write

```
promise (op x when guard  $\mapsto$  comp) as p in cont
```

as a syntactic sugar for the recursively defined interrupt handler

```
let rec waitForGuard () =  
  promise (op x  $\mapsto$  if guard then comp else waitForGuard ()) as p' in return p'  
in  
let p = waitForGuard () in cont
```

- For it to be well-typed, `comp` must be **promise-typed**
- **Necessitates gen. rec.** in the core calculus (more on that later)

# Example: remote function calls

- Server

```
let server f =  
  let rec loop () =  
    promise (call (x, callNo)  $\mapsto$  let y = f x in  $\uparrow$  result (y, callNo); loop ())  
    as p in return p  
  in loop ()
```

# Example: remote function calls

- Server

```
let server f =  
  let rec loop () =  
    promise (call (x, callNo)  $\mapsto$  let y = f x in  $\uparrow$  result (y, callNo); loop ())  
    as p in return p  
  in loop ()
```

- Client

```
let callWith x =  
  let callNo = !callCounter in callCounter := !callCounter + 1;  
   $\uparrow$  call (x, callNo);  
  promise (result (y, callNo') when callNo = callNo'  $\mapsto$  return  $\langle y \rangle$ ) as resultProm in  
  return (fun ()  $\rightarrow$  await resultProm until  $\langle$ resultValue $\rangle$  in return resultValue)
```

# Example: remote function calls

- Server

```
let server f =  
  let rec loop () =  
    promise (call (x, callNo)  $\mapsto$  let y = f x in  $\uparrow$  result (y, callNo); loop ())  
    as p in return p  
  in loop ()
```

- Client

```
let callWith x =  
  let callNo = !callCounter in callCounter := !callCounter + 1;  
   $\uparrow$  call (x, callNo);  
  promise (result (y, callNo') when callNo = callNo'  $\mapsto$  return  $\langle y \rangle$ ) as resultProm in  
  return (fun ()  $\rightarrow$  await resultProm until  $\langle$ resultValue $\rangle$  in return resultValue)
```

- Shortcomings

- Again necessitates general recursion in the core calculus
- No way to send the function  $f$  from client to server
- Subsequent calls are executed sequentially on the server

# Example: preemptive multi-threading

- At the core of our approach is the following recursive definition

```
let rec waitForStop () =  
  promise (stop _  $\mapsto$   
    promise (go _  $\mapsto$  return  $\langle() \rangle$ ) as p in (await p until  $\langle-\rangle$  in waitForStop ()))  
  ) as p' in return p'
```

- first wait for stop interrupt, but do not block execution
- next wait for go interrupt, and block execution
- repeat the cycle

## Example: preemptive multi-threading

- At the core of our approach is the following recursive definition

```
let rec waitForStop () =  
  promise (stop _  $\mapsto$   
    promise (go _  $\mapsto$  return <()>) as p in (await p until <-> in waitForStop ()))  
  ) as p' in return p'
```

- first wait for stop interrupt, but do not block execution
  - next wait for go interrupt, and block execution
  - repeat the cycle
- To initiate preemptive behaviour for some comp, run the composite

```
waitForStop (); comp
```

- op. sem. propagates promises out, and wrap them around comp



## Example: preemptive multi-threading

- At the core of our approach is the following recursive definition

```
let rec waitForStop () =  
  promise (stop _  $\mapsto$   
    promise (go _  $\mapsto$  return <()>) as p in (await p until <-> in waitForStop ())  
  ) as p' in return p'
```

- first wait for stop interrupt, but do not block execution
  - next wait for go interrupt, and block execution
  - repeat the cycle
- To initiate preemptive behaviour for some comp, run the composite

```
waitForStop (); comp
```

- op. sem. propagates promises out, and wrap them around comp
- Note:** No need to access the cont. (of comp) in waitForStop (!)

## Other examples (see <https://matija.pretnar.info/aeff/>)

- A multi-party web application
- Simulating cancellable remote function calls
- Parallel variant of runners of algebraic effects
- Non-blocking post-processing of promised values

## Other examples (see <https://matija.pretnar.info/aeff/>)

- A multi-party web application
- Simulating cancellable remote function calls
- Parallel variant of runners of algebraic effects
- Non-blocking post-processing of promised values

```
promise (op x  $\mapsto$  original_interrupt_handler) as p in
...
processop p with ( $\langle is \rangle \mapsto$  filter (fun i  $\mapsto$  i > 0) is) as q in
processop q with ( $\langle js \rangle \mapsto$  fold (fun j j'  $\mapsto$  j * j') 1 js) as r in
processop r with ( $\langle k \rangle \mapsto \uparrow$  productOfPositiveElements k) as _ in
...
```

where

```
processop p with ( $\langle x \rangle \mapsto$  comp) as q in cont
=
promise (op _  $\mapsto$  await p until  $\langle x \rangle$  in let y = comp in return  $\langle y \rangle$ ) as q in cont
```

**Resolving  $\lambda_{\text{æ}}$ 's shortcomings**

# S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers

# S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers
- **Solution:** **reinstallable** interrupt handlers

TY-COMP-REPROMISE

$$\frac{\begin{array}{c} \Gamma, x : A_{op}, \quad r : 1 \rightarrow \langle X \rangle ! (o', \iota') \vdash M : \langle X \rangle ! (o', \iota') \\ (o', \iota') \sqsubseteq \iota(op) \quad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota) \end{array}}{\Gamma \vdash \text{promise } (op \ x \ r \mapsto M) \text{ as } p \text{ in } N : Y ! (o, \iota)}$$

# S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers
- Solution:** **reinstallable** interrupt handlers

TY-COMP-REPROMISE

$$\Gamma, x : A_{op}, r : 1 \rightarrow \langle X \rangle ! (o', \iota') \vdash M : \langle X \rangle ! (o', \iota')$$

$$(o', \iota') \sqsubseteq \iota(op) \quad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota)$$


---


$$\Gamma \vdash \text{promise } (op \ x \ r \mapsto M) \text{ as } p \text{ in } N : Y ! (o, \iota)$$

- Operationally only difference in **triggering int. handlers**

- $\downarrow op (W, \text{promise } (op \ x \mapsto M) \text{ as } p \text{ in } N)$

$\rightsquigarrow \text{let } p = M[W/x,$

$$(\text{fun } \_ \mapsto \text{promise } (op \ x \ r \mapsto M) \text{ as } p \text{ in return } p) / r ]$$

$\text{in } \downarrow op (W, N)$

# S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers
- Solution:** **reinstallable** interrupt handlers

TY-COMP-REPROMISE

$$\Gamma, x : A_{op}, r : 1 \rightarrow \langle X \rangle ! (o', \iota') \vdash M : \langle X \rangle ! (o', \iota')$$
$$(o', \iota') \sqsubseteq \iota(op) \quad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota)$$
$$\Gamma \vdash \text{promise } (op \times r \mapsto M) \text{ as } p \text{ in } N : Y ! (o, \iota)$$

- For example, the preemptive multithreading now becomes

```
let waitForStop () =  
  promise (stop _ r  $\mapsto$   
    promise (go _ _  $\mapsto$  return  $\langle () \rangle$ ) as p in (await p until  $\langle - \rangle$  in r ()))  
  ) as p' in return p'
```



## S2: signal/interrupt payloads ground-typed

- E.g., cannot send functions for remote execution

## S2: signal/interrupt payloads ground-typed

- E.g., cannot send functions for remote execution
- Solution:** off-the-shelf **Fitch-style modal types** (Clouston et al.)

TYVAL-VARIABLE

$X \text{ is mobile} \vee \blacksquare \notin \Gamma'$

$\Gamma, x:X, \Gamma' \vdash x:X$

TYVAL-BOX

$\Gamma, \blacksquare \vdash V:X$

$\Gamma \vdash [V]:[X]$

TYCOMP-UNBOX

$\Gamma \vdash V:[X] \quad \Gamma, x:X \vdash M:Y!(o, \iota)$

$\Gamma \vdash \text{unbox } V \text{ as } [x] \text{ in } M:Y!(o, \iota)$

$A_{\text{op}} ::= \text{ground types} \mid [X] \quad (\text{mobile types})$

## S2: signal/interrupt payloads ground-typed

- E.g., cannot send functions for remote execution
- Solution:** off-the-shelf **Fitch-style modal types** (Clouston et al.)

TYVAL-VARIABLE

$X \text{ is mobile} \vee \blacksquare \notin \Gamma'$

$\Gamma, x : X, \Gamma' \vdash x : X$

TYVAL-BOX

$\Gamma, \blacksquare \vdash V : X$

$\Gamma \vdash [V] : [X]$

TYCOMP-UNBOX

$\Gamma \vdash V : [X] \quad \Gamma, x : X \vdash M : Y ! (o, \iota)$

$\Gamma \vdash \text{unbox } V \text{ as } [x] \text{ in } M : Y ! (o, \iota)$

$A_{\text{op}} ::= \text{ground types} \mid [X]$  (mobile types)

- Gives us **type-safe higher-order** payloads for signals/interrupts
  - $\Gamma, p : \langle X \rangle \vdash V : A_{\text{op}} \implies \Gamma \vdash V : A_{\text{op}}$

## S3: no dynamic process/thread creation

- E.g., remote functions have to be executed sequentially

## S3: no dynamic process/thread creation

- E.g., remote functions have to be executed sequentially
- **Solution:** type safe **spawn** via **modal types**

$$\frac{\text{TYCOMP-SPAWN} \quad \Gamma, \blacksquare \vdash M : 1! (o', \iota') \quad \Gamma \vdash N : X! (o, \iota)}{\Gamma \vdash \text{spawn} (M, N) : X! (o, \iota)}$$

## S3: no dynamic process/thread creation

- E.g., remote functions have to be executed sequentially
- **Solution:** type safe **spawn** via **modal types**

$$\frac{\text{TYCOMP-SPAWN} \quad \Gamma, \blacksquare \vdash M : 1! (o', \iota') \quad \Gamma \vdash N : X! (o, \iota)}{\Gamma \vdash \text{spawn} (M, N) : X! (o, \iota)}$$

- Operationally **propagates outwards** (like scoped alg. op.)
  - **let**  $x = \text{spawn} (M_1, M_2)$  **in**  $N \rightsquigarrow \text{spawn} (M_1, \text{let } x = M_2 \text{ in } N)$
  - also propagates through **promise**, where  $\blacksquare$  provides **type-safety**
- **Does not block** its continuation
- Eventually gives rise to a **new parallel process**
  - $\text{run} (\text{spawn} (M, N)) \rightsquigarrow \text{run } M \parallel \text{run } N$

## S3: no dynamic process/thread creation

- E.g., remote functions have to be executed sequentially
- Solution:** type safe **spawn** via **modal types**

$$\frac{\text{TYCOMP-SPAWN} \quad \Gamma, \blacksquare \vdash M : 1! (o', \iota') \quad \Gamma \vdash N : X! (o, \iota)}{\Gamma \vdash \text{spawn} (M, N) : X! (o, \iota)}$$

- Remote function calls can now execute in parallel

```
let server f =  
  promise (call (x, callNo) r  $\mapsto$   
    spawn (let y = f x in  $\uparrow$  result (y, callNo),  
           r ())  
  ) as p in return p
```

# Conclusion

- A core calculus for asynchronous algebraic effects
- Could it serve as a spec. for an efficient/practical implementation?
  - Janez has been working on a more efficient implementation of  $\lambda_{\text{æ}}$
  - Implementing this spec. using handlers? (Lindley & Poulson)



# Conclusion

- A core calculus for asynchronous algebraic effects
- Could it serve as a spec. for an efficient/practical implementation?
  - Janez has been working on a more efficient implementation of  $\lambda_{\infty}$
  - Implementing this spec. using handlers? (Lindley & Poulson)
- Same algebraic & modal ideas also applicable without  $||$

$\text{async } M \text{ as } p \text{ in } N$

with

$\text{async } (\uparrow \text{op } (V, M)) \text{ as } p \text{ in } N \rightsquigarrow \uparrow \text{op } (V, \text{async } M \text{ as } p \text{ in } N)$

$\text{async } M \text{ as } p \text{ in } (\uparrow \text{op } (V, N)) \rightsquigarrow \uparrow \text{op } (V, \text{async } M \text{ as } p \text{ in } N)$

# Appendix

## $\lambda_{\text{æ}}$ -calculus: effect annotations

- The effect annotations  $(o, \iota)$  are drawn from sets  $O$  and  $I$ , given by

$$O = \mathcal{P}(\Sigma) \qquad I = \nu Z . \Sigma \Rightarrow (O \times Z)_{\perp}$$

where  $\Sigma$  is the set of all signal/interrupt names

- Note: for meta-theory only, could also have  $I$  as a least fixpoint

## $\lambda_{\text{æ}}$ -calculus: effect annotations

- The effect annotations  $(o, \iota)$  are drawn from sets  $O$  and  $I$ , given by

$$O = \mathcal{P}(\Sigma) \qquad I = \nu Z . \Sigma \Rightarrow (O \times Z)_{\perp}$$

where  $\Sigma$  is the set of all signal/interrupt names

- Note: for meta-theory only, could also have  $I$  as a least fixpoint
- $O$  and  $I$  come with natural partial orders for subtyping

# $\lambda_{\text{æ}}$ -calculus: effect annotations

- The effect annotations  $(o, \iota)$  are drawn from sets  $O$  and  $I$ , given by

$$O = \mathcal{P}(\Sigma) \qquad I = \nu Z . \Sigma \Rightarrow (O \times Z)_{\perp}$$

where  $\Sigma$  is the set of all signal/interrupt names

- Note: for meta-theory only, could also have  $I$  as a least fixpoint
- $O$  and  $I$  come with natural partial orders for subtyping
- The action  $\text{op} \downarrow (o, \iota)$  reveals effects of int. handlers for  $\text{op}$

$$\text{op} \downarrow (o, \iota) \stackrel{\text{def}}{=} \begin{cases} (o \cup o', \iota[\text{op} \mapsto \perp] \cup \iota') & \text{if } \iota(\text{op}) = (o', \iota') \\ (o, \iota) & \text{otherwise} \end{cases}$$