(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 Sklodowska-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

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$
\begin{array}{rl}
 & M_{\mathsf{op}}[V/x] \\
\text{signal op's implementation} & \uparrow \\
\ldots \quad \rightsquigarrow & \mathsf{op}\,(V, y.M)
\end{array}
$$

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$
\begin{tikzcd}
 & M_{\mathsf{op}}[V/x] \arrow[r, phantom, "\rightsquigarrow^*"] & \mathsf{return}\ W \arrow[d, "\text{interrupt main program}"] \\
\ldots \arrow[r, phantom, "\rightsquigarrow"] & \mathsf{op}\,(V, y.M) \arrow[u, "\text{signal op's implementation}"] & M[W/y]
\end{tikzcd}
$$

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$
\begin{tikzcd}
 & M_{\mathsf{op}}[V/x] \arrow[r, rightsquigarrow, "*"] & \mathsf{return}\ W \arrow[d, "\text{interrupt main program}"] & \\
\ldots \arrow[r, rightsquigarrow] & \mathsf{op}\,(V, y.M) \arrow[u, "\text{signal op's implementation}"] & M[W/y] \arrow[r, rightsquigarrow] & \ldots
\end{tikzcd}
$$

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$
\begin{tikzcd}
 & M_{\mathsf{op}}[V/x] \arrow[r, rightsquigarrow, "*"] & \mathsf{return}\ W \arrow[d, "\text{interrupt main program}"] & \\
\dots \arrow[r, rightsquigarrow] & \mathsf{op}\,(V, y.M) \arrow[u, "\text{signal op's implementation}"] \arrow[r, phantom, "\underbrace{\qquad}_{\text{main program's execution blocked}}"'] & M[W/y] \arrow[r, rightsquigarrow] & \dots
\end{tikzcd}
$$

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

$$
\begin{array}{ccccccc}
 & & M_{\mathsf{op}}[V/x] & \rightsquigarrow^* & \mathsf{return}\ W & & \\
 & \text{signal op's implementation} & \uparrow & & \downarrow & \text{interrupt main program} & \\
\ldots & \rightsquigarrow & \mathsf{op}\,(V, y.M) & \underbrace{\qquad\qquad}_{\text{main program's execution blocked}} & M[W/y] & \rightsquigarrow & \ldots
\end{array}
$$

  - M_{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

$$
\begin{tikzcd}
 & M_{\mathsf{op}}[V/x] \arrow[r, squiggly, "*"] & \textcolor{blue}{\mathsf{return}}\ W \arrow[d, "\text{interrupt main program}"] & \\
\dots \arrow[r, squiggly] & \mathsf{op}\,(V, y.M) \arrow[u, "\text{signal op's implementation}"] \arrow[r, phantom, "\underbrace{\qquad\qquad}_{\textcolor{red}{\text{main program's execution blocked}}}"'] & M[W/y] \arrow[r, squiggly] & \dots
\end{tikzcd}
$$

  - M_{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

$$\begin{tikzcd}[column sep=small]
 & & M_{\mathsf{op}}[V/x] \arrow[rr, "\rightsquigarrow^*", phantom] & & \mathsf{return}\ W \arrow[d, "\text{interrupt main program}"] & & \\
\ldots \arrow[r, phantom, "\rightsquigarrow"] & & \mathsf{op}\ (V, y.M) \arrow[u, "\text{signal op's implementation}"] \arrow[rr, phantom, "\underbrace{\qquad\qquad}_{\text{main program's execution blocked}}"] & & M[W/y] \arrow[r, phantom, "\rightsquigarrow"] & \ldots &
\end{tikzcd}$$

  - M_{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

$\lambda_{æ}$-calculus: basics

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

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

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

$\lambda_{æ}$-calculus: signals

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

$\lambda_{æ}$-calculus: signals

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

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

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

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

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

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

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

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

$\lambda_{æ}$-calculus: signals

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

$$\dfrac{\begin{array}{l}\textsc{TyComp-Signal}\\ \mathsf{op} : A_{\mathsf{op}} \in o \qquad \Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash M : X \,!\, (o, \iota)\end{array}}{\Gamma \vdash \uparrow \mathsf{op}\,(V, M) : X \,!\, (o, \iota)}$$

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

- 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)

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

where op acts on the effect annotations in conclusion

$\lambda_{æ}$-calculus: interrupts

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

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

  where op acts on the effect annotations in conclusion

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

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

- Allow computation to react to interrupts

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

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

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

- Allow computation to react to interrupts

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

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

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

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

- Allow computation to react to interrupts

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

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

- 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 ctd.

- Allow computation to react to interrupts

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

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

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

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

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

- Allow computation to react to interrupts

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

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

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

    $\Longrightarrow$

    $$\begin{aligned} &\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N \\ \rightsquigarrow\ &\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N' \end{aligned}$$

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

- Allow computation to react to interrupts

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

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

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

    $\Longrightarrow$

    $\quad \mathtt{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathtt{as}\ p\ \mathtt{in}\ N$

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

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

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

- Enables programmers to selectively block execution

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

$\lambda_{æ}$-calculus: awaiting

- Enables programmers to selectively block execution

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

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

$\lambda_{æ}$-calculus: environment

- We model the environment by running computations in parallel

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

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

$\lambda_{æ}$-calculus: environment

- We model the environment by running computations in parallel

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

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

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

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

- We model the environment by running computations in parallel

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

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

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

$\lambda_{æ}$-calculus: environment

- We model the environment by running computations in parallel

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

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

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

Examples

Example: guarded interrupt handlers

- In examples we often write

```
promise (op x when guard ↦ comp) as p in cont
```

as a syntactic sugar for the recursively defined interrupt handler

```
let rec waitForGuard () =
    promise (op x ↦ 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 ↦ comp) as p in cont
```

  as a syntactic sugar for the recursively defined interrupt handler

```
let rec waitForGuard () =
    promise (op x ↦ 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) ↦ let y = f x in ↑ 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) ↦ let y = f x in ↑ result (y, callNo); loop ())
        as p in return p
    in loop ()
```

- Client

```
let callWith x =
    let callNo = !callCounter in callCounter := !callCounter + 1;
    ↑ call (x, callNo);
    promise (result (y, callNo') when callNo = callNo' ↦ return ⟨y⟩) as resultProm in
    return (fun () → await resultProm until ⟨resultValue⟩ in return resultValue)
```

Example: remote function calls

- Server

```
let server f =
    let rec loop () =
        promise (call (x, callNo) ↦ let y = f x in ↑ result (y, callNo); loop ())
        as p in return p
    in loop ()
```

- Client

```
let callWith x =
    let callNo = !callCounter in callCounter := !callCounter + 1;
    ↑ call (x, callNo);
    promise (result (y, callNo') when callNo = callNo' ↦ return ⟨y⟩) as resultProm in
    return (fun () → await resultProm until ⟨resultValue⟩ 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 _ ↦
        promise (go _ ↦ 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

Example: preemptive multi-threading

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

```
let rec waitForStop () =
    promise (stop _ ↦
        promise (go _ ↦ 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 preemtive 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 _ ↦
        promise (go _ ↦ 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 preemtive 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

$$
\begin{array}{l}
\mathsf{promise}\ (\mathrm{op}\ x \mapsto \mathrm{original\_interrupt\_handler})\ \mathsf{as}\ p\ \mathsf{in} \\
\ldots \\
\mathsf{process}_{\mathrm{op}}\ p\ \mathsf{with}\ (\langle is \rangle \mapsto \mathrm{filter}\ (\mathsf{fun}\ i \mapsto i > 0)\ is)\ \mathsf{as}\ q\ \mathsf{in} \\
\mathsf{process}_{\mathrm{op}}\ q\ \mathsf{with}\ (\langle js \rangle \mapsto \mathrm{fold}\ (\mathsf{fun}\ j\ j' \mapsto j * j')\ 1\ js)\ \mathsf{as}\ r\ \mathsf{in} \\
\mathsf{process}_{\mathrm{op}}\ r\ \mathsf{with}\ (\langle k \rangle \mapsto \uparrow \mathrm{productOfPositiveElements}\ k)\ \mathsf{as}\ \_\ \mathsf{in} \\
\ldots
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{process}_{\mathrm{op}}\ p\ \mathsf{with}\ (\langle x \rangle \mapsto \mathrm{comp})\ \mathsf{as}\ q\ \mathsf{in}\ \mathrm{cont} \\
= \\
\mathsf{promise}\ (\mathrm{op}\ \_ \mapsto \mathsf{await}\ p\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ \mathsf{let}\ y = \mathrm{comp}\ \mathsf{in}\ \mathsf{return}\ \langle y \rangle)\ \mathsf{as}\ q\ \mathsf{in}\ \mathrm{cont}
\end{array}
$$

Resolving $\lambda_{æ}$'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

$$\dfrac{\begin{array}{c}\textsc{Ty-Comp-RePromise} \\ \Gamma, x : A_{op}, \boxed{r : 1 \to \langle X \rangle \,!\, (o', \iota')} \vdash M : \langle X \rangle \,!\, (o', \iota') \\ \boxed{(o', \iota') \sqsubseteq \iota\,(\mathsf{op})} \qquad \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)\end{array}}{\Gamma \vdash \mathbf{promise}\ (\mathsf{op}\ x\ \boxed{r} \mapsto M)\ \mathbf{as}\ p\ \mathbf{in}\ N : Y \,!\, (o, \iota)}$$

S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers

- Solution: reinstallable interrupt handlers

$$\frac{\begin{array}{c}\text{TY-COMP-REPROMISE}\\ \Gamma, x : A_{op}, \boxed{r : 1 \to \langle X \rangle \,!\, (o', \iota')} \vdash M : \langle X \rangle \,!\, (o', \iota') \\ \boxed{(o', \iota') \sqsubseteq \iota\,(\mathsf{op})} \qquad \Gamma, p : \langle X \rangle \vdash N : Y \,!\, (o, \iota)\end{array}}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x\ \boxed{r} \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y \,!\, (o, \iota)}$$

- Operationally only difference in triggering int. handlers

  - $$\begin{array}{l}\downarrow \mathsf{op}\,(W, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N) \\ \rightsquigarrow \mathsf{let}\ p = M[W/x, \\ \qquad \boxed{(\mathsf{fun}\ \_ \mapsto \mathsf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p)/r}\,] \\ \quad \mathsf{in} \downarrow \mathsf{op}\,(W, N)\end{array}$$

S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers

- Solution: reinstallable interrupt handlers

$$\frac{\Gamma, x : A_{op}, \boxed{r : 1 \to \langle X \rangle ! (o', \iota')} \vdash M : \langle X \rangle ! (o', \iota') \qquad \boxed{(o', \iota') \sqsubseteq \iota(\mathsf{op})} \qquad \Gamma, p : \langle X \rangle \vdash N : Y ! (o, \iota)}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x\ \boxed{r} \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y ! (o, \iota)}\ \textsc{Ty-Comp-RePromise}$$

- For example, the preemptive multithreading now becomes

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

$$\frac{\textsc{TyVal-Variable}}{}\qquad \frac{X \text{ is mobile} \quad \vee \quad \blacksquare \notin \Gamma'}{\Gamma, x : X, \Gamma' \vdash x : X}$$

$$\textsc{TyVal-Box}\qquad \frac{\Gamma, \blacksquare \vdash V : X}{\Gamma \vdash [V] : [X]}$$

$$\textsc{TyComp-Unbox}\qquad \frac{\Gamma \vdash V : [X] \qquad \Gamma, x : X \vdash M : Y \mathbin{!} (o, \iota)}{\Gamma \vdash \texttt{unbox } V \texttt{ as } [x] \texttt{ in } M : Y \mathbin{!} (o, \iota)}$$

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

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.)

$$\textsc{TyVal-Variable}\quad \frac{\boxed{X \text{ is mobile} \quad \vee \quad \blacksquare \notin \Gamma'}}{\Gamma, x : X, \Gamma' \vdash x : X} \qquad \textsc{TyVal-Box}\quad \frac{\Gamma, \blacksquare \vdash V : X}{\Gamma \vdash [V] : [X]}$$

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

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

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

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

$$\textsc{TyComp-Spawn}\quad \frac{\Gamma, \blacksquare \vdash M : 1 \,!\, (o', \iota') \qquad \Gamma \vdash N : X \,!\, (o, \iota)}{\Gamma \vdash \mathsf{spawn}\,(M, N) : X \,!\, (o, \iota)}$$

S3: no dynamic process/thread creation

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

$$\dfrac{\Gamma, \blacksquare \vdash M : 1 \,!\, (o', \iota') \qquad \Gamma \vdash N : X \,!\, (o, \iota)}{\Gamma \vdash \mathsf{spawn}\,(M, N) : X \,!\, (o, \iota)}\;\textsc{TyComp-Spawn}$$

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

S3: no dynamic process/thread creation

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

- Solution: type safe spawn via modal types

$$\frac{\Gamma, \blacksquare \vdash M : 1 \,!\, (o', \iota') \qquad \Gamma \vdash N : X \,!\, (o, \iota)}{\Gamma \vdash \mathsf{spawn}\,(M, N) : X \,!\, (o, \iota)}\;\textsc{TyComp-Spawn}$$

- Remote function calls can now execute in parallel

```
let server f =
    promise (call (x, callNo) r ↦
        spawn (let y = f x in ↑ 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 $\lambda_{æ}$
  - Implementing this spec. using handlers? (Lindley & Poulson)
- Same algebraic & modal ideas also applicable without $\|$

$$\mathsf{async}\, M \,\mathsf{as}\, p \,\mathsf{in}\, N$$

with

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

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

Appendix

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

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

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

  where Σ 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, ι) are drawn from sets O and I, given by

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

  where Σ 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

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

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

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

  where Σ 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 $\mathsf{op} \downarrow (o, \iota)$ reveals effects of int. handlers for op

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