Higher-Order Asynchronous Effects

Danel Ahman   Matija Pretnar   Janez Radešček

University of Ljubljana, Slovenia

August 22 @ HOPE 2021

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

- Problem: Synchrony of algebraic effects
- Solution: Asynchrony through decoupling operation call execution
- $\lambda_{æ}$-calculus
- Examples
- Some recent extensions (the higher-order part of the talk's title)

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

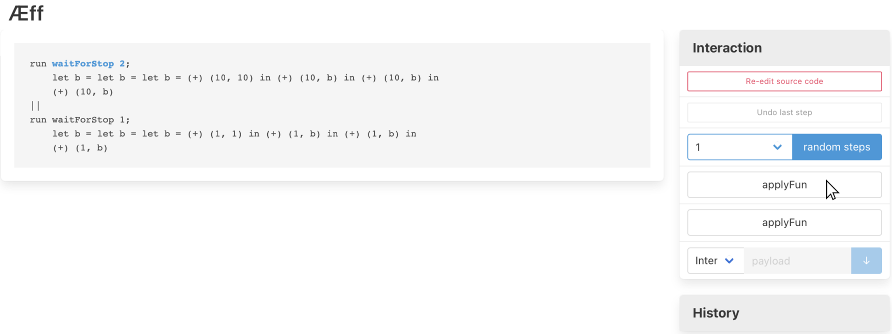
https://github.com/matijapretnar/aeff

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

https://github.com/danelahman/higher-order-aeff-agda

Æff web interface

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



Synchrony of algebraic effects

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

signalling op's implementation $\uparrow$

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

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

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

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

main program's execution blocked

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

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 ($\uparrow$) — interrupting main program ($\downarrow$)

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

main program's execution blocked

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

- In this work, we enable asynchrony for alg. ops. by
  - observing that alg. op. calls execute in multiple phases, and by
  - providing programming abstractions capturing these phases
  - in a self-contained core calculus

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

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

$\lambda_{æ}$-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 ground type (prod. and sum of base types)

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

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

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

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

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

$\lambda_{æ}$-calculus: interrupts

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

TyComp-Interrupt

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

$\lambda_{æ}$-calculus: interrupts

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

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

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

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

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

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

- 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
  - $M \leadsto M' \quad \Longrightarrow \quad \downarrow \mathsf{op}\,(V, M) \leadsto \downarrow \mathsf{op}\,(V, M')$

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

- Allow computation to react to interrupts

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

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

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

- Allow computation to react to interrupts

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

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

- Operationally behave like (scoped) algebraic operations (!)
  - $\mathsf{let}\ x = (\mathsf{promise}\ (\mathsf{op}\ x \mapsto M_1)\ \mathsf{as}\ p\ \mathsf{in}\ M_2)\ \mathsf{in}\ N$
    $\rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x \mapsto M_1)\ \mathsf{as}\ p\ \mathsf{in}\ (\mathsf{let}\ x = M_2\ \mathsf{in}\ N)$

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

- Allow computation 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

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

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

- Allow computation 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 (!)
  - $\mathsf{let}\ x = (\mathsf{promise}\ (\mathsf{op}\ x \mapsto M_1)\ \mathsf{as}\ p\ \mathsf{in}\ M_2)\ \mathsf{in}\ N$
    $\rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x \mapsto M_1)\ \mathsf{as}\ p\ \mathsf{in}\ (\mathsf{let}\ x = M_2\ \mathsf{in}\ N)$
  - $\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in} \uparrow \mathsf{op}\,(V, N)$ (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

- Allow computation to react to interrupts

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

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 computation 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

- 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

- Allow computation 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 computation to react to interrupts

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

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

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

    $\Longrightarrow$

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

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

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

- Enables programmers to selectively block execution

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

$\lambda_{æ}$-calculus: awaiting

- Enables programmers to selectively block execution

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

- Operationally behave like pattern-matching (and 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

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

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

$\lambda_{æ}$-calculus: environment

- We model the environment by running computations in parallel

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

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

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

$\lambda_{æ}$-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 +
  - $\mathsf{run}\ (\uparrow\mathsf{op}\,(V, M)) \rightsquigarrow \uparrow\mathsf{op}\,(V, \mathsf{run}\ M)$
  - $(\uparrow\mathsf{op}\,(V, P)) \parallel Q \rightsquigarrow \uparrow\mathsf{op}\,(V, (P \parallel \downarrow\mathsf{op}\,(V, Q)))$
  - $P \parallel (\uparrow\mathsf{op}\,(V, Q)) \rightsquigarrow \uparrow\mathsf{op}\,(V, (\downarrow\mathsf{op}\,(V, P) \parallel Q))$

$\lambda_{æ}$-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: remote function calls

Example: remote function calls

- 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

- 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)
```

- 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

- 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)
```

- 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 ()
```

- **Shortcomings** (fixes for those later in the talk)
  - 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

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 wraps 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 wraps them around comp

- **Note:** No need to access the cont. (of comp) in waitForStop (!)

Other examples (see paper/prototype)

- Algebraic operation calls (special case of remote function calls)
- Multi-party web application
- (Simulating) cancellations of remote function calls
- Parallel variant of runners of algebraic effects
- Non-blocking post-processing of promised values

Other examples (see paper/prototype)

- Algebraic operation calls (special case of remote function calls)
- Multi-party web application
- (Simulating) cancellations of remote function calls
- Parallel variant of runners of algebraic effects
- Non-blocking post-processing of promised values

```
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
...
```

where

```
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
```

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

TY-COMP-REPROMISE

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

S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers

- Solution: reinstallable interrupt handlers

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

- Operationally only difference in triggering int. handlers

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

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

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

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

S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers

- Solution: reinstallable interrupt handlers

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

- 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
  - (need to be able to propagate payloads past binders in promises)

S2: signal/interrupt payloads ground-typed

- E.g., cannot send functions for remote execution
  - (need to be able to propagate payloads past binders in promises)

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

$$X \;::=\; \ldots \;\mid\; [X] \qquad\qquad A_{\mathsf{op}} \;::=\; \text{ground types} \;\mid\; [X]$$

TyVal-Variable

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

TyVal-Box

$$\frac{\Gamma, \text{[binoculars]} \vdash V : X}{\Gamma \vdash [V] : [X]}$$

TyComp-Unbox

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

S2: signal/interrupt payloads ground-typed

- E.g., cannot send functions for remote execution
  - (need to be able to propagate payloads past binders in promises)

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

$$X \;::=\; \ldots \;\mid\; [X] \qquad\qquad A_{\mathsf{op}} \;::=\; \text{ground types} \;\mid\; [X]$$

TyVal-Variable

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

TyVal-Box

$$\frac{\Gamma, \bowtie \vdash V : X}{\Gamma \vdash [V] : [X]}$$

TyComp-Unbox

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

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

S3: no dynamic process/thread creation

- E.g., remote function calls have to be executed sequentially
  - (need to propagate spawned procs. past binders in promises)

S3: no dynamic process/thread creation

- E.g., remote function calls have to be executed sequentially
  - (need to propagate spawned procs. past binders in promises)

- Solution: type safe spawn via modal types

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

S3: no dynamic process/thread creation

- E.g., remote function calls have to be executed sequentially
  - (need to propagate spawned procs. past binders in promises)

- Solution: type safe spawn via modal types

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

- 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 promises, where $\faBinoculars$ provides type-safety

- Eventually gives rise to a new parallel process
  - $\mathsf{run}\ (\mathsf{spawn}\,(M, N)) \rightsquigarrow \mathsf{run}\ M \parallel \mathsf{run}\ N$

- Does not block its continuation

S3: no dynamic process/thread creation

- E.g., remote function calls have to be executed sequentially
  - (need to propagate spawned procs. past binders in promises)

- Solution: type safe spawn via modal types

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

- 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
    - based on decoupling the execution of alg. operation calls
    - accommodates both cooperative and preemptive behaviour
- Ongoing work on
    - $\lambda_{\text{æ}}$'s denotational semantics
    - more efficient variant of the operational semantics

Conclusion

- A core calculus for asynchronous algebraic effects
  - based on decoupling the execution of alg. operation calls
  - accommodates both cooperative and preemptive behaviour
- Ongoing work on
  - $\lambda_{æ}$'s denotational semantics
  - more efficient variant of the operational semantics
- Same algebraic & modal ideas also useful in setting without $||$

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

  with

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

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

Appendix

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

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

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

  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

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

Example: (tail res.) alg. operation calls

- Based on the earlier observation

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

- At call site

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

$$
\stackrel{\text{def}}{=}
$$

$$
\uparrow \mathsf{call}_{\mathsf{op}}\,(V, \mathsf{promise}\,(\mathsf{result}_{\mathsf{op}}\ y \mapsto \mathsf{return}\,\langle y \rangle)\ \mathsf{as}\ p\ \mathsf{in}
$$

$$
\mathsf{await}\ p\ \mathsf{until}\ \langle y \rangle\ \mathsf{in}\ M)
$$

- At implementation site

$$
\mathsf{promise}\,(\mathsf{call}_{\mathsf{op}}\ x \mapsto \mathsf{let}\ y = M_{op}\ \mathsf{in}\ \mathsf{return}\,\langle y \rangle)\ \mathsf{as}\ p\ \mathsf{in}
$$

$$
\mathsf{await}\ p\ \mathsf{until}\ \langle y \rangle\ \mathsf{in} \uparrow \mathsf{result}_{\mathsf{op}}\,(y, \mathsf{return}\,())
$$

Example: guarded interrupt handlers

- In many examples we often write for convenience

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

  as a syntactic sugar for the recursively defined interrupt handler

```
promise (op x r ↦ if guard then comp else r ()) as p in cont
```

- For well-typedness, important we have comp : $\langle X \rangle$ instead of comp : X

- In POPL paper, again necessitated gen. rec. in the core calculus