(Higher-Order) Asynchronous Effects

Danel Ahman   Matija Pretnar   Janez Radešček

University of Ljubljana, Slovenia

July 21, ~~2020~~ 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
- $\lambda_{æ}$-calculus
- Examples

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

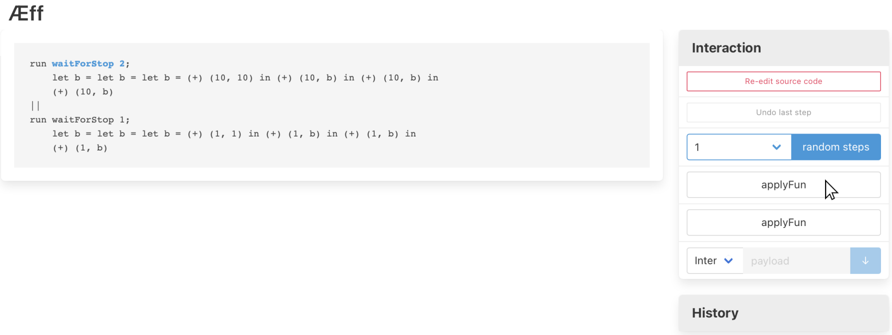
https://github.com/matijapretnar/aeff

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

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

Æff web interface

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



Synchrony of algebraic effects

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

  - 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

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

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

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

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

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

- Forces all uses of algebraic operations to be synchronous

Synchrony of algebraic effects

- The conventional operational treatment of algebraic effects

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

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

- Forces all uses of algebraic operations 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

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

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

- Forces all uses of algebraic operations 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

$\lambda_{æ}$-calculus

$\lambda_{æ}$-calculus: basics

- Extension of Levy's fine-grain call-by-value λ-calculus
- **Types:** $X, Y \ ::= \ \mathsf{b} \ | \ \ldots \ | \ X \to Y \,!\, (o, \iota) \ | \ \ldots$
- **Values:** $V, W \ ::= \ x \ | \ \ldots \ | \ \mathsf{fun}\,(x : X) \mapsto M \ | \ \ldots$
- **Computations:** $M, N \ ::= \ \mathsf{return}\ V \ | \ \mathsf{let}\ x = M \ \mathsf{in}\ N \ | \ \ldots$
- **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

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

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

$\lambda_{æ}$-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)

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

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

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

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

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

- 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)

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

where op acts on the effect annotations in conclusion

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

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

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

  where op acts on the effect annotations in conclusion

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

$\lambda_{æ}$-calculus: interrupts

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

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

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

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

- Allow computation to react to interrupts

Ty-Comp-Promise

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

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

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

- Allow computation to react to interrupts

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

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

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

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

- Allow computation to react to interrupts

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

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

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

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

- Allow computation to react to interrupts

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

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

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

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

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

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

- Allow computation to react to interrupts

TY-COMP-PROMISE

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

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

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

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

$\lambda_{æ}$-calculus: awaiting

- Enables programmers to selectively block execution

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

$\lambda_{æ}$-calculus: awaiting

- Enables programmers to selectively block execution

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

- 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 (!)

$\lambda_{\text{æ}}$-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)

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

$\lambda_{æ}$-calculus: environment

- We model the environment by running computations in parallel

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

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

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

Examples

Example: (tail res.) alg. operation calls

- Based on the earlier observation

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

Example: (tail res.) alg. operation calls

- Based on the earlier observation

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

- At call site

$$
\begin{array}{c}
\mathsf{op}\,(V, y.M) \\
\stackrel{\text{def}}{=} \\
\uparrow \mathsf{call}_{\mathsf{op}}\,(V, \mathsf{promise}\,(\mathsf{result}_{\mathsf{op}}\ y \mapsto \mathsf{return}\,\langle y\rangle)\ \mathsf{as}\ p\ \mathsf{in} \\
\mathsf{await}\ p\ \mathsf{until}\ \langle y\rangle\ \mathsf{in}\ M)
\end{array}
$$

Example: (tail res.) alg. operation calls

- Based on the earlier observation

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

- 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: 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 (fixes for those later)
    - 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: guarded interrupt handlers

- In previous example (and many others) 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 previous example (and many others) 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 well-typedness, important we have $\mathsf{comp} : \langle X \rangle$ instead of $\mathsf{comp} : X$

- Again necessitates gen. rec. in the core calculus

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/)

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

- 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

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

S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers

- Solution: reinstallable interrupt handlers

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

- Operationally only difference in triggering int. handlers

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

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

S1: general recursion in the core calculus

- Used in almost all examples for reinstalling interrupt handlers

- Solution: reinstallable interrupt handlers

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

- 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.)

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

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

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

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.)

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

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

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

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

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

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

S3: no dynamic process/thread creation

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

- Solution: type safe spawn via modal types

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

- 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 promises, where 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 functions have to be executed sequentially
- Solution: type safe spawn via modal types

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

- 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 worked on a more efficient implementation of $\lambda_{æ}$
  - Implementing this spec. using handlers? (Lindley & Poulson)
- Various yet to be resolved details concerning $\lambda_{æ}$'s denot. sem.

Conclusion

- A core calculus for asynchronous algebraic effects
- Could it serve as a spec. for an efficient/practical implementation?
  - Janez has worked on a more efficient implementation of $\lambda_{æ}$
  - Implementing this spec. using handlers? (Lindley & Poulson)
- Various yet to be resolved details concerning $\lambda_{æ}$'s denot. sem.
- 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

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

  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

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