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

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

DANEL AHMAN MATIJA PRETNAR

UNIVERSITY OF LJUBLJANA, SLOVENIA

07.01.2021

ASYNCHRONOUS EFFECTS

THE PROBLEM

THE SETTING WE WORK IN

THE SETTING WE WORK IN

- Effectful programming with algebraic effects and effect handlers

[Plotkin & Power '02, Plotkin & Pretnar '09]

THE SETTING WE WORK IN

- Effectful programming with algebraic effects and effect handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{op}\,(V, y\,.\,M) \quad | \quad \mathsf{handle}\; M \;\mathsf{with}\; H$$

$$H \quad ::= \quad \{\ \ldots\ ,\ \mathsf{op_i}\; x\; k \mapsto M_{\mathsf{op_i}}\ ,\ \ldots\ ,\ \mathsf{return}\; x \mapsto N_{\mathsf{ret}}\ \}$$

[Plotkin & Power '02, Plotkin & Pretnar '09]

THE SETTING WE WORK IN

- Effectful programming with algebraic effects and effect handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{op}\,(V, y\,.\,M) \quad | \quad \mathsf{handle}\ M\ \mathsf{with}\ H$$

$$H \quad ::= \quad \{\ \ldots\ ,\ \mathsf{op_i}\ x\ k \mapsto M_{\mathsf{op_i}}\ ,\ \ldots\ ,\ \mathsf{return}\ x \mapsto N_{\mathsf{ret}}\ \}$$

- Separates (operation-based) interfaces from (user-definable) implementations

[Plotkin & Power '02, Plotkin & Pretnar '09]

THE SETTING WE WORK IN

- Effectful programming with **algebraic effects** and **effect handlers**

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{op}\,(V, y\,.\,M) \quad | \quad \mathsf{handle}\ M\ \mathsf{with}\ H$$

$$H \quad ::= \quad \{\ \ldots\ ,\ \mathsf{op_i}\ x\ k \mapsto M_{\mathsf{op_i}}\ ,\ \ldots\ ,\ \mathsf{return}\, x \mapsto N_{\mathsf{ret}}\ \}$$

- Separates (operation-based) **interfaces** from (user-definable) **implementations**

$$\mathsf{handle}\ (\mathsf{return}\ V)\ \mathsf{with}\ H \quad \rightsquigarrow \quad N_{\mathsf{ret}}[V/x]$$

$$\mathsf{handle}\ (\mathsf{op}\ (V, y\,.\,M))\ \mathsf{with}\ H \quad \rightsquigarrow \quad M_{\mathsf{op}}[V/x, (\mathsf{fun}\ y \mapsto \mathsf{handle}\ M\ \mathsf{with}\ H)/k]$$

[Plotkin & Power '02, Plotkin & Pretnar '09]

THE SETTING WE WORK IN

- Effectful programming with **algebraic effects** and **effect handlers**

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{op}\,(V, y\,.\,M) \quad | \quad \mathsf{handle}\ M\ \mathsf{with}\ H$$

$$H \quad ::= \quad \{\ \ldots\ ,\ \mathsf{op_i}\ x\ k \mapsto M_{\mathsf{op_i}}\ ,\ \ldots\ ,\ \mathsf{return}\ x \mapsto N_{\mathsf{ret}}\ \}$$

- Separates (operation-based) **interfaces** from (user-definable) **implementations**

$$\mathsf{handle}\ (\mathsf{return}\ V)\ \mathsf{with}\ H \quad \rightsquigarrow \quad N_{\mathsf{ret}}[V/x]$$

$$\mathsf{handle}\ (\mathsf{op}\,(V, y\,.\,M))\ \mathsf{with}\ H \quad \rightsquigarrow \quad M_{\mathsf{op}}[V/x, (\mathsf{fun}\ y \mapsto \mathsf{handle}\ M\ \mathsf{with}\ H)/k]$$

- State, rollbacks, exceptions, non-determ., concurrency, prob. programming, ...

[Plotkin & Power '02, Plotkin & Pretnar '09]

THE PROBLEM WITH ALGEBRAIC EFFECTS

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

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

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

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

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation
* effect handler
* runner of alg. effects

$$M_{\mathsf{op}}[V/x]$$

$$\uparrow$$

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

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation
* effect handler
* runner of alg. effects

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

$$\uparrow$$

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

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation
* effect handler
* runner of alg. effects

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

$$\uparrow$$

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

continuation is blocked until W is computed

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation

* effect handler

* runner of alg. effects

$$\begin{array}{ccccccc} & & M_{\mathsf{op}}[V/x] & \rightsquigarrow^{*} & \mathsf{return}\ W & & \\ & & \uparrow & & \downarrow & & \\ \ldots & \rightsquigarrow & \mathsf{op}\,(V, y\,.\,M) & & M[W/y] & \rightsquigarrow & \ldots \end{array}$$

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation
* effect handler
* runner of alg. effects

$$
\begin{tikzcd}
 & M_{\mathsf{op}}[V/x] \arrow[rr, rightsquigarrow, "*"] & & \mathsf{return}\ W \arrow[d] & \\
\ldots \arrow[r, rightsquigarrow] & \mathsf{op}\ (V, y\,.\,M) \arrow[u] & & M[W/y] \arrow[r, rightsquigarrow] & \ldots
\end{tikzcd}
$$

- Blocking needed in the presence of (non-linear) general effect handlers, and to avoid having to reduce open terms (y is bound immediately)

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation
* effect handler
* runner of alg. effects

$$
\begin{array}{ccccccc}
 & & M_{\mathsf{op}}[V/x] & \leadsto^* & \mathsf{return}\ W & & \\
 & & \uparrow & & \downarrow & & \\
\ldots & \leadsto & \mathsf{op}\ (V, y\,.\,M) & & M[W/y] & \leadsto & \ldots
\end{array}
$$

- Blocking needed in the presence of (non-linear) general effect handlers, and to avoid having to reduce open terms (y is bound immediately)

- But it forces all uses of alg. effs. to be synchronous, even if this is not necessary

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation
* effect handler
* runner of alg. effects

$$\begin{array}{ccccccc} & & M_{\mathsf{op}}[V/x] & \leadsto^{*} & \mathsf{return}\ W & & \\ & & \uparrow & & \downarrow & & \\ \ldots & \leadsto & \mathsf{op}\,(V, y\,.\,M) & & M[W/y] & \leadsto & \ldots \end{array}$$

- Blocking needed in the presence of (non-linear) general effect handlers, and to avoid having to reduce open terms (y is bound immediately)
- But it forces all uses of alg. effs. to be synchronous, even if this is not necessary
- Existing approaches to asynchrony simply delegate it to language backends

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation
* effect handler
* runner of alg. effects

$$
\begin{array}{ccccccccc}
 & & M_{\mathsf{op}}[V/x] & \rightsquigarrow^* & \mathsf{return}\ W & & \\
 & & \uparrow & & \downarrow & & \\
\ldots & \rightsquigarrow & \mathsf{op}\ (V, y\,.\,M) & & M[W/y] & \rightsquigarrow & \ldots
\end{array}
$$

- Blocking needed in the presence of (non-linear) general effect handlers, and to avoid having to reduce open terms (y is bound immediately)

- But it forces all uses of alg. effs. to be synchronous, even if this is not necessary

- Existing approaches to asynchrony simply delegate it to language backends

Koka [Leijen '17], Multicore OCaml [Dolan et al. '18]

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation
* effect handler
* runner of alg. effects

$$\begin{array}{ccccccc} & & M_{\mathsf{op}}[V/x] & \rightsquigarrow^{*} & \mathsf{return}\ W & & \\ & & \uparrow & & \downarrow & & \\ \ldots & \rightsquigarrow & \mathsf{op}\,(V, y\,.\,M) & & M[W/y] & \rightsquigarrow & \ldots \end{array}$$

- Blocking needed in the presence of (non-linear) general effect handlers, and to avoid having to reduce open terms (y is bound immediately)
- But it forces all uses of alg. effs. to be synchronous, even if this is not necessary
- Existing approaches to asynchrony simply delegate it to language backends

THE PROBLEM WITH ALGEBRAIC EFFECTS

- Conventional treatment of algebraic effects is inherently synchronous

* top-level implementation
* effect handler
* runner of alg. effects

$$
\begin{array}{ccccccc}
 & & M_{\mathsf{op}}[V/x] & \rightsquigarrow^{*} & \mathsf{return}\ W & & \\
 & & \uparrow & & \downarrow & & \\
\ldots & \rightsquigarrow & \mathsf{op}\ (V, y \,.\, M) & & M[W/y] & \rightsquigarrow & \ldots
\end{array}
$$

- Blocking needed in the presence of (non-linear) general effect handlers, and to avoid having to reduce open terms (y is bound immediately)
- But it forces all uses of alg. effs. to be synchronous, even if this is not necessary
- Existing approaches to asynchrony simply delegate it to language backends

This paper: How to capture asynchrony in a self-contained core language?

THE IDEA

THE THREE PHASES OF ALGEBRAIC OPERATION CALLS

THE THREE PHASES OF ALGEBRAIC OPERATION CALLS

$$
\begin{array}{ccccccc}
 & & M_{\mathsf{op}}[V/x] & \leadsto^{*} & \mathsf{return}\ W & & \\
 & & \uparrow & & \downarrow & & \\
\ldots & \leadsto & \mathsf{op}\,(V, y\,.\,M) & & M[W/y] & \leadsto & \ldots
\end{array}
$$

THE THREE PHASES OF ALGEBRAIC OPERATION CALLS

executing op's impl.

(2)

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

signalling to execute some impl. of op

(1) $\uparrow$

$\downarrow$ (3)

interrupting M's blocking

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

- Execution of algebraic operation calls has **three distinct phases**

THE THREE PHASES OF ALGEBRAIC OPERATION CALLS

executing op's impl.

(2)

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

signalling to execute
some impl. of op

interrupting M's blocking

$$(1) \uparrow \qquad\qquad \downarrow (3)$$

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

- Execution of algebraic operation calls has three distinct phases
- Idea: Decouple all three phases into separate programming constructs, so that

THE THREE PHASES OF ALGEBRAIC OPERATION CALLS

executing op's impl.

(2)

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

signalling to execute some impl. of op

(1) $\uparrow$ $\qquad\qquad$ $\downarrow$ (3)

interrupting M's blocking

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

- Execution of algebraic operation calls has three distinct phases
- Idea: Decouple all three phases into separate programming constructs, so that
  - M would not block while (2) happens asynchronously,
  - programmers could choose if/when to block M for (3) to happen, and
  - (3) could happen without originating from (1) $\qquad$ (and vice versa)

THE APPROACH

THE SIGNALS

THE SIGNALS

- Our computations can issue outgoing signals

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

THE SIGNALS

- Our computations can issue outgoing signals

signal name

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \underbrace{\mathsf{op}}_{\text{signal name}} (\underbrace{V}_{\text{payload}}, M)$$

THE SIGNALS

- Our computations can issue outgoing signals

signal name

payload

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

continuation

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

  - propagate outwards (↑-notation) (just like algebraic operations)

$$\begin{aligned} & \mathsf{let}\ x = \big(\uparrow \mathsf{op}\,(V, M)\,\big)\ \mathsf{in}\ N \\ \rightsquigarrow \quad & \uparrow \mathsf{op}\,\big(V, (\mathsf{let}\ x = M\ \mathsf{in}\ N)\big) \end{aligned}$$

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)
  - do not block their continuation (unlike algebraic operations)

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\ (V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)
  - do not block their continuation (unlike algebraic operations)

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

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)
  - do not block their continuation (unlike algebraic operations)

$$\ldots \quad \rightsquigarrow \quad \uparrow \mathsf{op}\,(V, M) \qquad \xrightarrow{\mathsf{op}\ V}$$

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)
  - do not block their continuation (unlike algebraic operations)

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

$\mathsf{op}\ V \uparrow$

THE SIGNALS

- Our computations can **issue outgoing signals**

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)
  - do not block their continuation (unlike algebraic operations)

$$\ldots \quad \rightsquigarrow \quad \uparrow \mathsf{op}\,(V, M) \xrightarrow{\mathsf{op}\ V\ \uparrow} \quad \rightsquigarrow \quad M \quad \rightsquigarrow \quad \ldots$$

THE SIGNALS

- Our computations can **issue outgoing signals**

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\ (V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)
  - do not block their continuation (unlike algebraic operations)

$$\ldots \quad \rightsquigarrow \quad \uparrow \mathsf{op}\ (V, M) \xrightarrow{\ \mathsf{op}\ V \uparrow\ } \quad \rightsquigarrow \quad M \quad \rightsquigarrow \quad \ldots$$

- Example: scrolling through a seemingly infinite feed (user & client & server)

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\,(V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)
  - do not block their continuation (unlike algebraic operations)

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

$$\mathsf{op}\ V \uparrow$$

- Example: scrolling through a seemingly infinite feed (user & client & server)

$$\uparrow \mathsf{request}\,(\mathit{cachedSize} + 1, M_{\mathsf{feedClient}})$$

THE SIGNALS

- Our computations can issue outgoing signals

$$M, N \quad ::= \quad \ldots \quad | \quad \uparrow \mathsf{op}\ (V, M)$$

  - propagate outwards ($\uparrow$-notation) (just like algebraic operations)
  - do not block their continuation (unlike algebraic operations)

$$\ldots \quad \rightsquigarrow \quad \uparrow \mathsf{op}\ (V, M) \xrightarrow{\mathsf{op}\ V \uparrow} \quad \rightsquigarrow \quad M \quad \rightsquigarrow \quad \ldots$$

- Example: scrolling through a seemingly infinite feed (user & client & server)

$$\uparrow \mathsf{request}\ (\mathit{cachedSize} + 1, M_{\mathsf{feedClient}}) \qquad \uparrow \mathsf{display}\ (\mathit{message}, M_{\mathsf{feedClient}})$$

THE INTERRUPTS

THE INTERRUPTS

- Our computations can be interrupted

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

THE INTERRUPTS

- Our computations can be **interrupted**

interrupt name

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\ (W, M)$$

THE INTERRUPTS

interrupt name

payload

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

THE INTERRUPTS

- Our computations can be **interrupted**

interrupt name

payload

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\ (W, M)$$

continuation

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\ (W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)

$$\begin{aligned} & \downarrow \mathsf{op}\ \big(W,\ \uparrow \mathsf{op}'\ (V, M)\big) \\ \rightsquigarrow \quad & \uparrow \mathsf{op}'\ \big(V,\ \downarrow \mathsf{op}\ (W, M)\big) \end{aligned}$$

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\, (W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)

$$\begin{aligned} & \downarrow \mathsf{op}\, \big(W, \uparrow \mathsf{op}'\, (V, M)\big) \\ \rightsquigarrow \quad & \uparrow \mathsf{op}'\, \big(V, \downarrow \mathsf{op}\, (W, M)\big) \end{aligned}$$

$$\begin{aligned} & \downarrow \mathsf{op}\, (W, \mathsf{return}\ V) \\ \rightsquigarrow \quad & \mathsf{return}\ V \end{aligned}$$

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

  - propagate inwards ($\downarrow$-notation)

(just like effect handling)

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)
  - do not block their continuation (just like effect handling)

THE INTERRUPTS

- Our computations can be interrupted

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)
  - do not block their continuation (just like effect handling)
  - can interrupt any sequence of reduction steps

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

  - **propagate inwards** ($\downarrow$-notation) (just like effect handling)
  - **do not block** their continuation (just like effect handling)
  - can interrupt **any sequence** of reduction steps

$$\ldots \quad \rightsquigarrow \quad M$$

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\ (W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)
  - do not block their continuation (just like effect handling)
  - can interrupt any sequence of reduction steps

$$\begin{array}{ccc} & & \big\downarrow \mathsf{op}\ W \\ \ldots & \rightsquigarrow & M \end{array}$$

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\ (W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)
  - do not block their continuation (just like effect handling)
  - can interrupt any sequence of reduction steps

$$\ldots \quad \rightsquigarrow \quad M \quad \rightsquigarrow \quad \downarrow \mathsf{op}\ (W, M)$$

$$\downarrow \mathsf{op}\ W$$

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)
  - do not block their continuation (just like effect handling)
  - can interrupt any sequence of reduction steps

$$\ldots \rightsquigarrow \underset{\big\downarrow\, \mathsf{op}\ W}{M} \rightsquigarrow \downarrow \mathsf{op}\,(W, M) \rightsquigarrow \ldots$$

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\,(W, M)$$

  - **propagate inwards** ($\downarrow$-notation) (just like effect handling)
  - **do not block** their continuation (just like effect handling)
  - can interrupt **any** sequence of reduction steps

THE INTERRUPTS

- Our computations can be interrupted

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\ (W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)
  - do not block their continuation (just like effect handling)
  - can interrupt any sequence of reduction steps

- Example: scrolling through a seemingly infinite feed (user & client & server)

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\ (W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)
  - do not block their continuation (just like effect handling)
  - can interrupt any sequence of reduction steps

- Example: scrolling through a seemingly infinite feed (user & client & server)

$$\downarrow \mathsf{response}\ (\mathit{newBatch}, M_{\mathsf{feedClient}})$$

THE INTERRUPTS

- Our computations can be **interrupted**

$$M, N \quad ::= \quad \ldots \quad | \quad \downarrow \mathsf{op}\ (W, M)$$

  - propagate inwards ($\downarrow$-notation) (just like effect handling)
  - do not block their continuation (just like effect handling)
  - can interrupt any sequence of reduction steps

- Example: scrolling through a seemingly infinite feed (user & client & server)

$$\downarrow \mathsf{response}\ (newBatch, M_{\mathsf{feedClient}}) \qquad \downarrow \mathsf{nextItem}\ ((), M_{\mathsf{feedClient}})$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}})) \ || \ \mathsf{run}\ M_{\mathsf{feedServer}}$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}})\ )\ ||\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are **not expected to write interrupts explicitly** in their programs!
- Instead, **interrupts are (commonly) induced by signals** from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}}))\ ||\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(propagate)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedClient}})\ ||\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are **not expected to write interrupts explicitly** in their programs!
- Instead, **interrupts are (commonly) induced by signals** from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}})\ )\ \|\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(propagate)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedClient}})\ \|\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are **not expected to write interrupts explicitly** in their programs!
- Instead, **interrupts are (commonly) induced by signals** from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}}))\ \mid\mid\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(propagate)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedClient}})\ \mid\mid\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(broadcast)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ \left(V, \mathsf{run}\ M_{\mathsf{feedClient}}\ \mid\mid\ \downarrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedServer}})\right)$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}}))\ ||\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(propagate)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedClient}})\ ||\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(broadcast)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ \Big(V, \mathsf{run}\ M_{\mathsf{feedClient}}\ ||\ \downarrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedServer}})\Big)$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

$$\text{run}\ (\uparrow \text{request}\ (V, M_{\text{feedClient}}))\ ||\ \text{run}\ M_{\text{feedServer}}$$

(propagate)

$$\rightsquigarrow\ \uparrow \text{request}\ (V, \text{run}\ M_{\text{feedClient}})\ ||\ \text{run}\ M_{\text{feedServer}}$$

(broadcast)

$$\rightsquigarrow\ \uparrow \text{request}\ \left(V, \text{run}\ M_{\text{feedClient}}\ ||\ \downarrow \text{request}\ (V, \text{run}\ M_{\text{feedServer}})\right)$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}}))\ \|\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(propagate)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedClient}})\ \|\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(broadcast)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ \big(V, \mathsf{run}\ M_{\mathsf{feedClient}}\ \|\ \downarrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedServer}})\big)$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

$$\text{run}\ (\uparrow \text{request}\ (V, M_{\text{feedClient}}))\ ||\ \text{run}\ M_{\text{feedServer}}$$

(propagate)

$$\rightsquigarrow\ \uparrow \text{request}\ (V, \text{run}\ M_{\text{feedClient}})\ ||\ \text{run}\ M_{\text{feedServer}}$$

(broadcast)

$$\rightsquigarrow\ \uparrow \text{request}\ \big(V, \text{run}\ M_{\text{feedClient}}\ ||\ \downarrow \text{request}\ (V, \text{run}\ M_{\text{feedServer}})\big)$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}}))\ ||\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(propagate)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedClient}})\ ||\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(broadcast)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ \big(V, \mathsf{run}\ M_{\mathsf{feedClient}}\ ||\ \downarrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedServer}})\big)$$

(propagate)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ \big(V, \mathsf{run}\ M_{\mathsf{feedClient}}\ ||\ \mathsf{run}\ (\downarrow \mathsf{request}\ (V, M_{\mathsf{feedServer}}))\big)$$

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}}))\ ||\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(propagate)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedClient}})\ ||\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(broadcast)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ \big(V, \mathsf{run}\ M_{\mathsf{feedClient}}\ ||\ \downarrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedServer}})\big)$$

(propagate)

$$\rightsquigarrow\ \uparrow \mathsf{request}\ \big(V, \mathsf{run}\ M_{\mathsf{feedClient}}\ ||\ \mathsf{run}\ (\downarrow \mathsf{request}\ (V, M_{\mathsf{feedServer}}))\big)$$

- But interrupts can also appear spontaneously!

THE SIGNAL FOR THE SENDER IS AN INTERRUPT TO THE RECEIVER

- Programmers are not expected to write interrupts explicitly in their programs!
- Instead, interrupts are (commonly) induced by signals from other processes

$$\mathsf{run}\ (\uparrow \mathsf{request}\ (V, M_{\mathsf{feedClient}}))\ \|\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(propagate)

$$\leadsto\ \uparrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedClient}})\ \|\ \mathsf{run}\ M_{\mathsf{feedServer}}$$

(broadcast)

$$\leadsto\ \uparrow \mathsf{request}\ \big(V, \mathsf{run}\ M_{\mathsf{feedClient}}\ \|\ \downarrow \mathsf{request}\ (V, \mathsf{run}\ M_{\mathsf{feedServer}})\big)$$

(propagate)

$$\leadsto\ \uparrow \mathsf{request}\ \big(V, \mathsf{run}\ M_{\mathsf{feedClient}}\ \|\ \mathsf{run}\ (\downarrow \mathsf{request}\ (V, M_{\mathsf{feedServer}}))\big)$$

- But interrupts can also appear spontaneously!
  - e.g. the user clicking a button or the environment preempting a process

THE INTERRUPT HANDLERS

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto\ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto\ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

interrupt name

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

interrupt name

handler code

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

continuation

interrupt name

handler code

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto \ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto\ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)

$$\begin{aligned} & \mathsf{let}\ y = \left(\mathsf{promise}\ (\mathsf{op}\ x \ \mapsto\ M_1)\ \mathsf{as}\ p\ \mathsf{in}\ M_2\right)\ \mathsf{in}\ N \\ \leadsto \quad & \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto\ M_1)\ \mathsf{as}\ p\ \mathsf{in}\ \left(\mathsf{let}\ x = M_2\ \mathsf{in}\ N\right) \end{aligned}$$

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto \ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto \ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto \ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)

$$\downarrow \mathsf{op}\ \big(V, \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto \ M)\ \mathsf{as}\ p\ \mathsf{in}\ N\big)$$

$$\rightsquigarrow \quad \mathsf{let}\ p = M[V/x]\ \mathsf{in} \downarrow \mathsf{op}\ (V, N)$$

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto \ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \text{promise}\ (\text{op}\ x \ \mapsto\ M)\ \text{as}\ p\ \text{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)
  - not triggered by non-matching interrupts

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto\ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)
  - not triggered by non-matching interrupts

$$\begin{aligned} & \downarrow \mathsf{op}\ \big(V, \mathsf{promise}\ (\mathsf{op}'\ x \ \mapsto\ M)\ \mathsf{as}\ p\ \mathsf{in}\ N\big) \\ \rightsquigarrow\ & \mathsf{promise}\ (\mathsf{op}'\ x \ \mapsto\ M)\ \mathsf{as}\ p\ \mathsf{in} \downarrow \mathsf{op}\ (V, N) \end{aligned} \qquad (\mathsf{op} \neq \mathsf{op}')$$

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto \ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)
  - not triggered by non-matching interrupts

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto \ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)
  - not triggered by non-matching interrupts
  - do not block their continuation

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{promise}\ (\mathsf{op}\ x \ \mapsto\ M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)
  - not triggered by non-matching interrupts
  - do not block their continuation

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

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \texttt{promise}\ (\mathsf{op}\ x \ \mapsto\ M)\ \texttt{as}\ p\ \texttt{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)
  - not triggered by non-matching interrupts
  - do not block their continuation

execution of open terms

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

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \text{promise}\ (\text{op}\ x \ \mapsto\ M)\ \text{as}\ p\ \text{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)
  - not triggered by non-matching interrupts
  - do not block their continuation

execution of open terms

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

p has promise type $\langle X \rangle$

THE INTERRUPT HANDLERS

- To react to interrupts our computations can install interrupt handlers

$$M, N \quad ::= \quad \ldots \quad | \quad \texttt{promise}\ (\mathsf{op}\ x \ \mapsto \ M)\ \texttt{as}\ p\ \texttt{in}\ N$$

  - propagate outwards (just like algebraic operations)
  - triggered by matching interrupts (interrupts are like deep effect handling)
  - not triggered by non-matching interrupts
  - do not block their continuation

execution of open terms

promise types ensure type safety!

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

p has promise type $\langle X \rangle$

THE AWAITING

THE AWAITING

- Programmers can selectively block execution to await a promise to be fulfilled

THE AWAITING

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N$$

THE AWAITING

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N$$

promise-typed value

THE AWAITING

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N$$

continuation

promise-typed value

THE AWAITING

- Programmers can selectively block execution to **await a promise to be fulfilled**

$$M, N \quad ::= \quad \ldots \quad | \quad \underbrace{\mathsf{await}\ V}_{\text{promise-typed value}}\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ \underbrace{N}_{\text{continuation}}$$

  - **reduces** when provided a **fulfilled promise**

THE AWAITING

- Programmers can selectively block execution to **await a promise to be fulfilled**

$$M, N \quad ::= \quad \ldots \quad | \quad \texttt{await}\ V\ \texttt{until}\ \langle x \rangle\ \texttt{in}\ N$$

promise-typed value

continuation

  - **reduces** when provided a **fulfilled promise**

$$\begin{aligned} & \texttt{await}\ \langle V \rangle\ \texttt{until}\ \langle x \rangle\ \texttt{in}\ N \\ \rightsquigarrow\ & N[V/x] \end{aligned}$$

THE AWAITING

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \texttt{await}\ V\ \texttt{until}\ \langle x \rangle\ \texttt{in}\ N$$

continuation

promise-typed value

- reduces when provided a fulfilled promise

THE AWAITING

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \textsf{await}\ V\ \textsf{until}\ \langle x \rangle\ \textsf{in}\ N$$

continuation

promise-typed value

- reduces when provided a fulfilled promise
- blocks execution on yet-to-be-fulfilled promises

THE AWAITING

- Programmers can selectively block execution to **await a promise to be fulfilled**

$$M, N \quad ::= \quad \ldots \quad | \quad \textsf{await}\ V\ \textsf{until}\ \langle x \rangle\ \textsf{in}\ N$$

continuation

promise-typed value

  - **reduces** when provided a **fulfilled promise**
  - **blocks** execution on **yet-to-be-fulfilled promises**

$$\textsf{await}\ p\ \textsf{until}\ \langle x \rangle\ \textsf{in}\ N$$

$$\not\rightsquigarrow$$

THE AWAITING

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \textsf{await}\ V\ \textsf{until}\ \langle x \rangle\ \textsf{in}\ N$$

continuation

promise-typed value

- reduces when provided a fulfilled promise
- blocks execution on yet-to-be-fulfilled promises

THE AWAITING

Example: client blocks until server sends its batch size

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \textsf{await } V \textsf{ until } \langle x \rangle \textsf{ in } N$$

continuation

promise-typed value

- reduces when provided a fulfilled promise
- blocks execution on yet-to-be-fulfilled promises

THE AWAITING

Example: client blocks until server sends its batch size

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N$$

continuation

promise-typed value

  - reduces when provided a fulfilled promise
  - blocks execution on yet-to-be-fulfilled promises

- We now also have all the pieces to express alg. operation calls $\mathsf{op}\ (V, y\,.\,M)$ as

THE AWAITING

Example: client blocks until server sends its batch size

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \textsf{await}\ V\ \textsf{until}\ \langle x \rangle\ \textsf{in}\ N$$

continuation

promise-typed value

  - reduces when provided a fulfilled promise
  - blocks execution on yet-to-be-fulfilled promises

- We now also have all the pieces to express alg. operation calls $\textsf{op}\ (V, y\,.\,M)$ as

  $\uparrow \textsf{op-sig}\ \big(V, \textsf{promise}\ (\textsf{op-int}\ x \ \mapsto\ \textsf{return}\ \langle x \rangle)\ \textsf{as}\ p\ \textsf{in}\ (\textsf{await}\ p\ \textsf{until}\ \langle y \rangle\ \textsf{in}\ M)\big)$

THE AWAITING

Example: client blocks until server sends its batch size

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \texttt{await}\ V\ \texttt{until}\ \langle x \rangle\ \texttt{in}\ N$$

continuation

promise-typed value

  - reduces when provided a fulfilled promise
  - blocks execution on yet-to-be-fulfilled promises

- We now also have all the pieces to express alg. operation calls $\texttt{op}\ (V, y \,.\, M)$ as

THE AWAITING

Example: client blocks until server sends its batch size

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N$$

continuation

promise-typed value

  - reduces when provided a fulfilled promise
  - blocks execution on yet-to-be-fulfilled promises

- We now also have all the pieces to express alg. operation calls $\mathsf{op}\ (V, y\,.\,M)$ as
  - and the implementations of op in parallel processes as follows

THE AWAITING

Example: client blocks until server sends its batch size

- Programmers can selectively block execution to await a promise to be fulfilled

$$M, N \quad ::= \quad \ldots \quad | \quad \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N$$

continuation

promise-typed value

  - reduces when provided a fulfilled promise
  - blocks execution on yet-to-be-fulfilled promises

- We now also have all the pieces to express alg. operation calls $\mathsf{op}\ (V, y\,.\,M)$ as
  - and the implementations of op in parallel processes as follows

$$\mathsf{promise}\ (\text{op-sig}\ x\ \mapsto\ \langle M_{\text{op}} \rangle)\ \mathsf{as}\ p\ \mathsf{in}\ \big(\mathsf{await}\ p\ \mathsf{until}\ \langle y \rangle\ \mathsf{in}\ \uparrow \text{op-int}\ (y, \mathsf{return}\ ())\big)$$

THE RUNNING EXAMPLE

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

* request server's settings,
* install int. handler for the response, and
* block until they arrive (but only after useful work)

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
          requestNewData (cachedSize + 1)
       else
          return ());
      (if !currentItem < cachedSize then
          ↑ display (toString (nth !cachedData !currentItem));
          currentItem := !currentItem + 1
       else
          ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
          requestNewData (cachedSize + 1)
       else
          return ());
      (if !currentItem < cachedSize then
          ↑ display (toString (nth !cachedData !currentItem));
          currentItem := !currentItem + 1
       else
          ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

client's main loop is a rec. defined int. handler

* reacts to next item interrupts from user
* issues display signals or new data requests

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

||

```
let server batchSize =
  let rec waitForBatchSize () =
    promise (batchSizeRequest () ↦
      ↑ batchSizeResponse batchSize;
      waitForBatchSize ()
    ) as p in return p
  in
  let rec waitForRequest () =
    promise (request offset ↦
      let payload = map (fun x ↦ 10 ∗ x) (range offset (offset + batchSize − 1)) in
      ↑ response payload;
      waitForRequest ()
    ) as p in return p
  in
  waitForBatchSize (); waitForRequest ()
```

THE RUNNING EXAMPLE

server processes are commonly rec. defined int. handlers

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

||

```
let server batchSize =
  let rec waitForBatchSize () =
    promise (batchSizeRequest () ↦
      ↑ batchSizeResponse batchSize;
      waitForBatchSize ()
    ) as p in return p
  in
  let rec waitForRequest () =
    promise (request offset ↦
      let payload = map (fun x ↦ 10 * x) (range offset (offset + batchSize − 1)) in
      ↑ response payload;
      waitForRequest ()
    ) as p in return p
  in
  waitForBatchSize (); waitForRequest ()
```

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

||

```
let server batchSize =
  let rec waitForBatchSize () =
    promise (batchSizeRequest () ↦
      ↑ batchSizeResponse batchSize;
      waitForBatchSize ()
    ) as p in return p
  in
  let rec waitForRequest () =
    promise (request offset ↦
      let payload = map (fun x ↦ 10 ∗ x) (range offset (offset + batchSize − 1)) in
      ↑ response payload;
      waitForRequest ()
    ) as p in return p
  in
  waitForBatchSize (); waitForRequest ()
```

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

||

```
let server batchSize =
  let rec waitForBatchSize () =
    promise (batchSizeRequest () ↦
      ↑ batchSizeResponse batchSize;
      waitForBatchSize ()
    ) as p in return p
  in
  let rec waitForRequest () =
    promise (request offset ↦
      let payload = map (fun x ↦ 10 ∗ x) (range offset (offset + batchSize − 1)) in
      ↑ response payload;
      waitForRequest ()
    ) as p in return p
  in
  waitForBatchSize (); waitForRequest ()
```

||

```
let rec user () =
  let rec wait n =
    if n = 0 then return () else wait (n − 1)
  in
  ↑ nextItem (); wait 10; user ()
```

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

||

```
let server batchSize =
  let rec waitForBatchSize () =
    promise (batchSizeRequest () ↦
      ↑ batchSizeResponse batchSize;
      waitForBatchSize ()
    ) as p in return p
  in
  let rec waitForRequest () =
    promise (request offset ↦
      let payload = map (fun x ↦ 10 ∗ x) (range offset (offset + batchSize − 1)) in
      ↑ response payload;
      waitForRequest ()
    ) as p in return p
  in
  waitForBatchSize (); waitForRequest ()
```

||

```
let rec user () =
  let rec wait n =
    if n = 0 then return () else wait (n − 1)
  in
  ↑ nextItem (); wait 10; user ()
```

THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

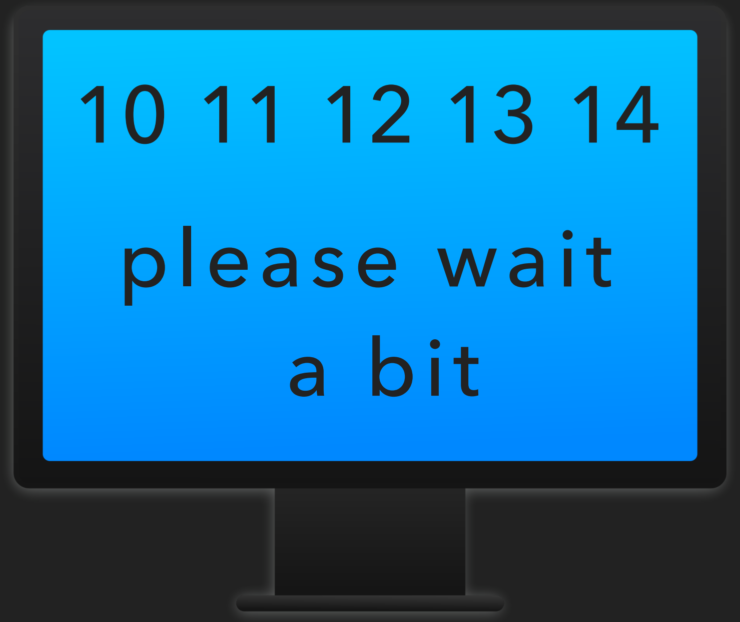
||

```
let server batchSize =
  let rec waitForBatchSize () =
    promise (batchSizeRequest () ↦
      ↑ batchSizeResponse batchSize;
      waitForBatchSize ()
    ) as p in return p
  in
  let rec waitForRequest () =
    promise (request offset ↦
      let payload = map (fun x ↦ 10 ∗ x) (range offset (offset + batchSize − 1)) in
      ↑ response payload;
      waitForRequest ()
    ) as p in return p
  in
  waitForBatchSize (); waitForRequest ()
```

||

```
let rec user () =
  let rec wait n =
    if n = 0 then return () else wait (n − 1)
  in
  ↑ nextItem (); wait 10; user ()
```

||



THE RUNNING EXAMPLE

```
let client () =
  ↑ batchSizeRequest ();
  promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) as batchSizePromise in

  let (cachedData , requestInProgress , currentItem) = (ref [] , ref false , ref 0) in

  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false; return ⟨()⟩
    ) as _ in return ()
  in

  let rec clientLoop batchSize =
    promise (nextItem () ↦
      let cachedSize = length !cachedData in
      (if (!currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
         requestNewData (cachedSize + 1)
       else
         return ());
      (if !currentItem < cachedSize then
         ↑ display (toString (nth !cachedData !currentItem));
         currentItem := !currentItem + 1
       else
         ↑ display "please wait a bit and try again");
      clientLoop batchSize
    ) as p in return p
  in

  await batchSizePromise until ⟨batchSize⟩ in clientLoop batchSize
```

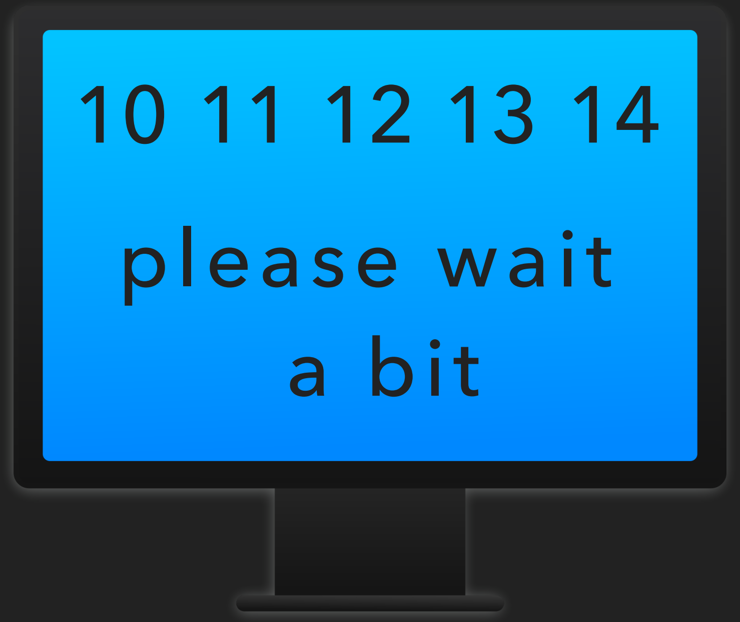
||

```
let server batchSize =
  let rec waitForBatchSize () =
    promise (batchSizeRequest () ↦
      ↑ batchSizeResponse batchSize;
      waitForBatchSize ()
    ) as p in return p
  in
  let rec waitForRequest () =
    promise (request offset ↦
      let payload = map (fun x ↦ 10 ∗ x) (range offset (offset + batchSize − 1)) in
      ↑ response payload;
      waitForRequest ()
    ) as p in return p
  in
  waitForBatchSize (); waitForRequest ()
```

||

```
let rec user () =
  let rec wait n =
    if n = 0 then return () else wait (n − 1)
  in
  ↑ nextItem (); wait 10; user ()
```

||



THE CALCULUS

THE $\lambda_{æ}$-CALCULUS

THE $\lambda_{æ}$-CALCULUS

- Extension of the **fine-grain call-by-value λ-calculus** [Levy et al. '03]
  - values

    $$V, W \quad ::= \quad \ldots \mid \langle V \rangle$$

  - computations

    $$M, N \quad ::= \quad \ldots \mid \text{gen. recursion} \mid \text{previously shown computations}$$

  - processes

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

THE $\lambda_{æ}$-CALCULUS

- Extension of the fine-grain call-by-value λ-calculus [Levy et al. '03]

  - values

    $$V, W \quad ::= \quad \ldots \;\mid\; \langle V \rangle$$

  - computations

    $$M, N \quad ::= \quad \ldots \;\mid\; \text{gen. recursion} \;\mid\; \text{previously shown computations}$$

  - processes

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

THE $\lambda_{æ}$-CALCULUS

- Extension of the fine-grain call-by-value λ-calculus [Levy et al. '03]
  - values

    $$V, W \quad ::= \quad \ldots \mid \langle V \rangle$$

    a fulfilled promise

  - computations

    M, N ::= … | gen. recursion | previously shown computations

  - processes

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

THE TYPES

THE TYPES

- Typing judgements $\Gamma \vdash V : X$ $\Gamma \vdash M : \mathscr{C}$ $\Gamma \vdash P : \mathscr{P}$

THE TYPES

- Typing judgements $\Gamma \vdash V : X \qquad \Gamma \vdash M : \mathscr{C} \qquad \Gamma \vdash P : \mathscr{P}$
- Value types $X, Y \;::=\; b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

THE TYPES

- Typing judgements $\Gamma \vdash V : X \qquad \Gamma \vdash M : \mathscr{C} \qquad \Gamma \vdash P : \mathscr{P}$

- Value types $X, Y \;::=\; b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

promise type

THE TYPES

- Typing judgements $\Gamma \vdash V : X$ $\Gamma \vdash M : \mathscr{C}$ $\Gamma \vdash P : \mathscr{P}$
- Value types $X, Y ::= b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

  promise type
- Ground/mobile types $A, B ::= b \mid 1 \mid 0 \mid A \times B \mid A + B$

THE TYPES

- Typing judgements $\Gamma \vdash V : X \qquad \Gamma \vdash M : \mathscr{C} \qquad \Gamma \vdash P : \mathscr{P}$

- Value types $X, Y \; ::= \; b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

  promise type

- Ground/mobile types $A, B \; ::= \; b \mid 1 \mid 0 \mid A \times B \mid A + B$

  used to type payloads of signals & interrupts

THE TYPES

- Typing judgements $\Gamma \vdash V : X$ $\Gamma \vdash M : \mathscr{C}$ $\Gamma \vdash P : \mathscr{P}$
- Value types $X, Y \ ::= \ b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

promise type

- Ground/mobile types $A, B \ ::= \ b \mid 1 \mid 0 \mid A \times B \mid A + B$

used to type payloads of signals & interrupts

- Computation types $\mathscr{C}, \mathscr{D} \ ::= \ X \,!\, (o, \iota)$

THE TYPES

- Typing judgements $\Gamma \vdash V : X \qquad \Gamma \vdash M : \mathscr{C} \qquad \Gamma \vdash P : \mathscr{P}$

- Value types $X, Y \ ::= \ b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

  promise type

- Ground/mobile types $A, B \ ::= \ b \mid 1 \mid 0 \mid A \times B \mid A + B$

  used to type payloads of signals & interrupts

- Computation types $\mathscr{C}, \mathscr{D} \ ::= \ X \,!\, (o, \iota)$

  type of returned values

THE TYPES

- Typing judgements $\Gamma \vdash V : X \qquad \Gamma \vdash M : \mathscr{C} \qquad \Gamma \vdash P : \mathscr{P}$

- Value types $X, Y \ ::= \ b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

  promise type

- Ground/mobile types $A, B \ ::= \ b \mid 1 \mid 0 \mid A \times B \mid A + B$

  used to type payloads of signals & interrupts

- Computation types $\mathscr{C}, \mathscr{D} \ ::= \ X \,!\, (o, \iota)$

  type of returned values

  possible issued signals

  $o \subseteq \Sigma$

THE TYPES

- Typing judgements $\Gamma \vdash V : X \qquad \Gamma \vdash M : \mathscr{C} \qquad \Gamma \vdash P : \mathscr{P}$
- Value types $X, Y \ ::= \ b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

  promise type

- Ground/mobile types $A, B \ ::= \ b \mid 1 \mid 0 \mid A \times B \mid A + B$

  used to type payloads of signals & interrupts

- Computation types $\mathscr{C}, \mathscr{D} \ ::= \ X \,!\, (o, \iota)$

  possible installed interrupt handlers

  $\iota \ = \ \{ \ldots \ , \ \mathsf{op}_i \to (\, o_i, \iota_i \,) \ , \ \ldots \}$

  type of returned values

  possible issued signals

  $o \subseteq \Sigma$

THE TYPES

- Typing judgements $\Gamma \vdash V : X$ $\Gamma \vdash M : \mathscr{C}$ $\Gamma \vdash P : \mathscr{P}$

- Value types $X, Y \ ::= \ b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

  promise type

- Ground/mobile types $A, B \ ::= \ b \mid 1 \mid 0 \mid A \times B \mid A + B$

  used to type payloads of signals & interrupts

- Computation types $\mathscr{C}, \mathscr{D} \ ::= \ X \,!\, (o, \iota)$

  possible installed interrupt handlers

  $\iota \ = \ \{ \ldots \ , \ \mathsf{op}_i \to (o_i, \iota_i) \ , \ \ldots \}$

  type of returned values

  possible issued signals

  $o \subseteq \Sigma$

- Process types $\mathscr{P}, \mathscr{Q} \ ::= \ X \,!!\, (o, \iota) \mid \mathscr{P} \,||\, \mathscr{Q}$

THE TYPES

- Typing judgements $\Gamma \vdash V : X \qquad \Gamma \vdash M : \mathscr{C} \qquad \Gamma \vdash P : \mathscr{P}$
- Value types $X, Y \ ::= \ b \mid 1 \mid 0 \mid X \times Y \mid X + Y \mid X \to \mathscr{C} \mid \langle X \rangle$

  promise type
- Ground/mobile types $A, B \ ::= \ b \mid 1 \mid 0 \mid A \times B \mid A + B$

  used to type payloads of signals & interrupts
- Computation types $\mathscr{C}, \mathscr{D} \ ::= \ X \,!\, (o, \iota)$

  possible installed interrupt handlers

  $\iota = \{ \ldots , \mathsf{op}_i \to (o_i , \iota_i) , \ldots \}$

  type of returned values

  possible issued signals

  $o \subseteq \Sigma$
- Process types $\mathscr{P}, \mathscr{Q} \ ::= \ X \,!!\, (o, \iota) \mid \mathscr{P} \,||\, \mathscr{Q}$

  match the structure of processes

THE TYPING RULES

THE TYPING RULES

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

THE TYPING RULES

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

op is allowed to happen

THE TYPING RULES

payload value matches op's signature $\quad \mathsf{op} : A_{\mathsf{op}}$

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

op is allowed to happen

THE TYPING RULES

payload value matches op's signature $\quad \mathsf{op} : A_{\mathsf{op}}$

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

op is allowed to happen

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

THE TYPING RULES

payload value matches op's signature $\quad \mathsf{op} : A_{\mathsf{op}}$

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

op is allowed to happen

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

action of interrupts
on effect information

THE TYPING RULES

payload value matches op's signature $\quad \mathsf{op} : A_{\mathsf{op}}$

op is allowed

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

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

action of interrupts
on effect information

THE TYPING RULES

payload value matches op's signature $\quad \mathsf{op} : A_{\mathsf{op}}$

op is allowed to happen

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

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

action of interrupts
on effect information

THE TYPING RULES

payload value matches op's signature $\mathsf{op} : A_{\mathsf{op}}$

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

op is allowed to happen

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

action of interrupts on effect information

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

THE TYPING RULES

payload value matches op's signature $\mathsf{op} : A_{\mathsf{op}}$

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

op is allowed to happen

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

effects of op's handlers

action of interrupts
on effect information

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

THE TYPING RULES

payload value matches op's signature $\mathsf{op} : A_{\mathsf{op}}$

op is allowed to happen

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

effects of op's handlers

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

action of interrupts
on effect information

promise-typed

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

THE OPERATIONAL SEMANTICS

THE OPERATIONAL SEMANTICS

- Small-step reduction semantics $\quad M \rightsquigarrow N \qquad P \rightsquigarrow Q$

THE OPERATIONAL SEMANTICS

- Small-step reduction semantics $M \rightsquigarrow N \qquad P \rightsquigarrow Q$
  - standard reduction rules from the fine-grain call-by-value λ-calculus

THE OPERATIONAL SEMANTICS

- Small-step reduction semantics $M \rightsquigarrow N \qquad P \rightsquigarrow Q$
  - standard reduction rules from the fine-grain call-by-value λ-calculus
  - reduction rules we have already seen

THE OPERATIONAL SEMANTICS

- Small-step reduction semantics $M \rightsquigarrow N \qquad P \rightsquigarrow Q$
  - standard reduction rules from the fine-grain call-by-value λ-calculus
  - reduction rules we have already seen
  - commutativity of signals with int. handlers (makes type safety interesting)

$$\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \left(\uparrow \mathsf{op}'\ (V, N)\right)$$

$$\rightsquigarrow \quad \uparrow \mathsf{op}'\ \left(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N\right)$$

THE OPERATIONAL SEMANTICS

- Small-step reduction semantics $\qquad M \leadsto N \qquad\qquad P \leadsto Q$
  - standard reduction rules from the fine-grain call-by-value λ-calculus
  - reduction rules we have already seen
  - commutativity of signals with int. handlers $\qquad$ (makes type safety interesting)

$$\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \Big(\uparrow \mathsf{op}'\ (V, N)\Big)$$

$$\leadsto\ \uparrow \mathsf{op}'\ \Big(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N\Big)$$

  - evaluation context rules

THE TYPE SAFETY

THE TYPE SAFETY

- Progress

- Type preservation

THE TYPE SAFETY

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
- Type preservation

THE TYPE SAFETY

* signals
* interrupt handlers
* blocked awaits

or

return values

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists N \,.\, M \rightsquigarrow N$ or M in result form
- Type preservation

THE TYPE SAFETY

* signals
* interrupt handlers
* blocked awaits

or

return values

eval. ctxs. only bind promise-typed variables

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
- Type preservation

THE TYPE SAFETY

* signals
* interrupt handlers
* blocked awaits

or

return values

eval. ctxs. only bind promise-typed variables

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
  - $\vdash P : \mathscr{P}$ implies $\exists\, Q \,.\, P \rightsquigarrow Q$ or P in result form
- Type preservation

THE TYPE SAFETY

* signals
* interrupt handlers
* blocked awaits

or

return values

eval. ctxs. only bind promise-typed variables

- Progress
  - $\vdash M : X\,!\,(o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
  - $\vdash P : \mathscr{P}$ implies $\exists\, Q \,.\, P \rightsquigarrow Q$ or P in result form

* signals
* parallel compositions
* individual computation result forms (w/o signals)

- Type preservation

THE TYPE SAFETY

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
  - $\vdash P : \mathscr{P}$ implies $\exists\, Q \,.\, P \rightsquigarrow Q$ or P in result form
- Type preservation

THE TYPE SAFETY

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
  - $\vdash P : \mathscr{P}$ implies $\exists\, Q \,.\, P \rightsquigarrow Q$ or P in result form
- Type preservation
  - $\Gamma \vdash M : X \,!\, (o, \iota)$ and $M \rightsquigarrow N$ imply $\Gamma \vdash N : X \,!\, (o, \iota)$

THE TYPE SAFETY

- Progress
  - $\vdash M : X\,!\,(o, \iota)$ implies $\exists\, N\,.\, M \rightsquigarrow N$ or M in result form
  - $\vdash P : \mathscr{P}$ implies $\exists\, Q\,.\, P \rightsquigarrow Q$ or P in result form
- Type preservation
  - $\Gamma \vdash M : X\,!\,(o, \iota)$ and $M \rightsquigarrow N$ imply $\Gamma \vdash N : X\,!\,(o, \iota)$

payloads do not include nor depend on promises

THE TYPE SAFETY

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
  - $\vdash P : \mathscr{P}$ implies $\exists\, Q \,.\, P \rightsquigarrow Q$ or P in result form
- Type preservation
  - $\Gamma \vdash M : X \,!\, (o, \iota)$ and $M \rightsquigarrow N$ imply $\Gamma \vdash N : X \,!\, (o, \iota)$

payloads do not include nor depend on promises $\quad \Gamma, p : \langle X \rangle \vdash V : A_{\mathrm{op}} \;\Rightarrow\; \Gamma \vdash V : A_{\mathrm{op}}$

THE TYPE SAFETY

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
  - $\vdash P : \mathscr{P}$ implies $\exists\, Q \,.\, P \rightsquigarrow Q$ or P in result form
- Type preservation
  - $\Gamma \vdash M : X \,!\, (o, \iota)$ and $M \rightsquigarrow N$ imply $\Gamma \vdash N : X \,!\, (o, \iota)$

THE TYPE SAFETY

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
  - $\vdash P : \mathscr{P}$ implies $\exists\, Q \,.\, P \rightsquigarrow Q$ or P in result form
- Type preservation
  - $\Gamma \vdash M : X \,!\, (o, \iota)$ and $M \rightsquigarrow N$ imply $\Gamma \vdash N : X \,!\, (o, \iota)$
  - $\Gamma \vdash P : \mathscr{P}$ and $P \rightsquigarrow Q$ imply $\exists\, \mathscr{Q} \,.\, \mathscr{P} \rightsquigarrow \mathscr{Q}$ and $\Gamma \vdash Q : \mathscr{Q}$

THE TYPE SAFETY

- Progress
  - $\vdash M : X \,!\, (o, \iota)$ implies $\exists\, N \,.\, M \rightsquigarrow N$ or M in result form
  - $\vdash P : \mathscr{P}$ implies $\exists\, Q \,.\, P \rightsquigarrow Q$ or P in result form
- Type preservation
  - $\Gamma \vdash M : X \,!\, (o, \iota)$ and $M \rightsquigarrow N$ imply $\Gamma \vdash N : X \,!\, (o, \iota)$
  - $\Gamma \vdash P : \mathscr{P}$ and $P \rightsquigarrow Q$ imply $\exists\, \mathscr{Q} \,.\, \mathscr{P} \rightsquigarrow \mathscr{Q}$ and $\Gamma \vdash Q : \mathscr{Q}$

process types also "reduce"

THE TYPE SAFETY

- Progress
  - or M in result form
  - in result form

- $\mathcal{P} \rightsquigarrow \mathcal{P}$
- $\mathcal{P} \rightsquigarrow \mathsf{op} \downarrow \mathcal{P}$
- $\mathcal{P} \rightsquigarrow \mathcal{Q} \implies \mathsf{op} \downarrow \mathcal{P} \rightsquigarrow \mathsf{op} \downarrow \mathcal{Q}$

where

- $\mathsf{op} \downarrow (X \,!!\, (o, \iota)) = X \,!\, (\mathsf{op} \downarrow (o, \iota))$
- $\mathsf{op} \downarrow (\mathcal{P} \,||\, \mathcal{Q}) = (\mathsf{op} \downarrow \mathcal{P}) \,||\, (\mathsf{op} \downarrow \mathcal{Q})$

- Typ
  - $\Gamma \vdash N : X \,!\, (o, \iota)$
  - $\mathcal{P} \rightsquigarrow \mathcal{Q}$ and $\Gamma \vdash Q : \mathcal{Q}$

process types also "reduce"

THE T

$$\uparrow \mathsf{op}\,(V, P) \mid\mid Q \;\rightsquigarrow\; \uparrow \mathsf{op}\,(V, P \mid\mid \downarrow \mathsf{op}\,(V, Q))$$

- Progress
  - or M in result form
  - in result form

- $\mathscr{P} \rightsquigarrow \mathscr{P}$
- $\mathscr{P} \rightsquigarrow \mathsf{op} \downarrow \mathscr{P}$
- $\mathscr{P} \rightsquigarrow \mathscr{Q} \implies \mathsf{op} \downarrow \mathscr{P} \rightsquigarrow \mathsf{op} \downarrow \mathscr{Q}$

where

- $\mathsf{op} \downarrow (X \,!!\, (o, \iota)) = X \,!\, (\mathsf{op} \downarrow (o, \iota))$
- $\mathsf{op} \downarrow (\mathscr{P} \mid\mid \mathscr{Q}) = (\mathsf{op} \downarrow \mathscr{P}) \mid\mid (\mathsf{op} \downarrow \mathscr{Q})$

- Typ
  - $\Gamma \vdash N : X \,!\, (o, \iota)$
  - $\mathscr{P} \rightsquigarrow \mathscr{Q}$ and $\Gamma \vdash Q : \mathscr{Q}$

process types also "reduce"

THE ARTEFACT

THE ARTEFACT

THE ARTEFACT

- Agda formalisation of $\lambda_{æ}$'s type safety results

THE ARTEFACT

- Agda formalisation of $\lambda_{æ}$'s type safety results
  - only well-typed syntax, and subsumption rule as an explicit coercion

THE ARTEFACT

- Agda formalisation of $\lambda_{æ}$'s type safety results
  - only well-typed syntax, and subsumption rule as an explicit coercion
- Prototype implementation of $\lambda_{æ}$ in OCaml, called Æff

THE ARTEFACT

- Agda formalisation of $\lambda_{æ}$'s type safety results
  - only well-typed syntax, and subsumption rule as an explicit coercion
- Prototype implementation of $\lambda_{æ}$ in OCaml, called Æff
  - interpreter

THE ARTEFACT

- Agda formalisation of $\lambda_{æ}$'s type safety results
  - only well-typed syntax, and subsumption rule as an explicit coercion
- Prototype implementation of $\lambda_{æ}$ in OCaml, called Æff
  - interpreter
  - simple typechecker (does not yet check effect information)

THE ARTEFACT

- Agda formalisation of $\lambda_{æ}$'s type safety results
  - only well-typed syntax, and subsumption rule as an explicit coercion
- Prototype implementation of $\lambda_{æ}$ in OCaml, called *Æff*
  - interpreter
  - simple typechecker (does not yet check effect information)
  - all the examples in the paper (and more)

THE ARTEFACT

- Agda formalisation of $\lambda_{æ}$'s type safety results
  - only well-typed syntax, and subsumption rule as an explicit coercion
- Prototype implementation of $\lambda_{æ}$ in OCaml, called *Æff*
  - interpreter
  - simple typechecker (does not yet check effect information)
  - all the examples in the paper (and more)
  - command line interface (one nondeterministic reduction sequence)

THE ARTEFACT

- Agda formalisation of $\lambda_{æ}$'s type safety results
  - only well-typed syntax, and subsumption rule as an explicit coercion
- Prototype implementation of $\lambda_{æ}$ in OCaml, called *Æff*
  - interpreter
  - simple typechecker (does not yet check effect information)
  - all the examples in the paper (and more)
  - command line interface (one nondeterministic reduction sequence)
  - web interface (possible to explore all reduction sequences)

THE EXAMPLES

THE EXAMPLES

THE EXAMPLES

- Preemptive multi-threading

THE EXAMPLES

- Preemptive multi-threading
- Remote function calls
  - including simulating call cancellations

THE EXAMPLES

- Preemptive multi-threading
- Remote function calls
  - including simulating call cancellations
- (Concurrent) runners of algebraic effects [Ahman & Bauer '20]

THE EXAMPLES

- Preemptive multi-threading
- Remote function calls
  - including simulating call cancellations
- (Concurrent) runners of algebraic effects [Ahman & Bauer '20]
- Non-blocking post-processing of promised values
  - in the same spirit as how one is taught to program with futures and promises

THE EXAMPLES

- Preemptive multi-threading
- Remote function calls
  - including simulating call cancellations
- (Concurrent) runners of algebraic effects [Ahman & Bauer '20]
- Non-blocking post-processing of promised values
  - in the same spirit as how one is taught to program with futures and promises
- Go-like select statements (see the Æff examples' library)
  - essentially n-ary (blocking) interrupt handlers

THE EXAMPLES

- Preemptive multi-threading
- Remote function calls
  - including simulating call cancellations
- (Concurrent) runners of algebraic effects [Ahman & Bauer '20]
- Non-blocking post-processing of promised values
  - in the same spirit as how one is taught to program with futures and promises
- Go-like select statements (see the Æff examples' library)
  - essentially n-ary (blocking) interrupt handlers

THE PREEMPTIVE MULTI-THREADING EXAMPLE

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then
- It is possible to simulate preemptive multi-threading [Dolan et al. '18]
  - but it requires low-level access to the specific runtime environment

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then
- It is possible to simulate preemptive multi-threading [Dolan et al. '18]
  - but it requires low-level access to the specific runtime environment
- In contrast, we can express preemptiveness directly within our calculus

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then
- It is possible to simulate preemptive multi-threading [Dolan et al. '18]
  - but it requires low-level access to the specific runtime environment
- In contrast, we can express preemptiveness directly within our calculus

```
waitForStop (); comp
```

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then
- It is possible to simulate preemptive multi-threading [Dolan et al. '18]
  - but it requires low-level access to the specific runtime environment
- In contrast, we can express preemptiveness directly within our calculus

```
waitForStop (); comp
```

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

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then
- It is possible to simulate preemptive multi-threading [Dolan et al. '18]
  - but it requires low-level access to the specific runtime environment
- In contrast, we can express preemptiveness directly within our calculus

```
waitForStop (); comp
```

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

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then
- It is possible to simulate preemptive multi-threading [Dolan et al. '18]
  - but it requires low-level access to the specific runtime environment
- In contrast, we can express preemptiveness directly within our calculus

```
waitForStop (); comp
```

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

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then
- It is possible to simulate preemptive multi-threading [Dolan et al. '18]
  - but it requires low-level access to the specific runtime environment
- In contrast, we can express preemptiveness directly within our calculus

```
waitForStop (); comp
```

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

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then
- It is possible to simulate preemptive multi-threading [Dolan et al. '18]
  - but it requires low-level access to the specific runtime environment
- In contrast, we can express preemptiveness directly within our calculus

```
waitForStop (); comp
```

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

THE PREEMPTIVE MULTI-THREADING EXAMPLE

- Multi-threading is one of the most exciting applications of algebraic effects
  - but the evaluation strategies one can express are cooperative in nature
  - each thread needs to explicitly yield back control, stalling others until then
- It is possible to simulate preemptive multi-threading [Dolan et al. '18]
  - but it requires low-level access to the specific runtime environment
- In contrast, we can express preemptiveness directly within our calculus

```
waitForStop (); comp
```

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

THE FUTURE

THE FUTURE

THE FUTURE

- Bidirectional type system, effect-checking, and channel-based implementation

THE FUTURE

- Bidirectional type system, effect-checking, and channel-based implementation
- Higher-order payloads and dynamic process creation
  - e.g., Fitch-style modal types to rule out enveloping promises from payloads

THE FUTURE

- Bidirectional type system, effect-checking, and channel-based implementation
- Higher-order payloads and dynamic process creation
  - e.g., Fitch-style modal types to rule out enveloping promises from payloads
- Denotational semantics based on monads for scoped effects [Piróg et al. '18]

THE FUTURE

- Bidirectional type system, effect-checking, and channel-based implementation
- Higher-order payloads and dynamic process creation
  - e.g., Fitch-style modal types to rule out enveloping promises from payloads
- Denotational semantics based on monads for scoped effects [Piróg et al. '18]
- Using the effect system for effect-dependent optimisations

$\vdash M : X \,!\, (o, \iota)$ and $\iota\,(\mathsf{op}) = \bot$ imply $\downarrow \mathsf{op}\,(V, M) \rightsquigarrow^{*} M$

THE FUTURE

- Bidirectional type system, effect-checking, and channel-based implementation
- Higher-order payloads and dynamic process creation
  - e.g., Fitch-style modal types to rule out enveloping promises from payloads
- Denotational semantics based on monads for scoped effects [Piróg et al. '18]
- Using the effect system for effect-dependent optimisations

  $\vdash M : X \,!\, (o, \iota)$ and $\iota\,(\mathsf{op}) = \bot$ imply $\downarrow \mathsf{op}\ (V, M) \rightsquigarrow^* M$
- Refine the "broadcast everything everywhere" communication strategy

THE FUTURE

- Bidirectional type system, effect-checking, and channel-based implementation
- Higher-order payloads and dynamic process creation
  - e.g., Fitch-style modal types to rule out enveloping promises from payloads
- Denotational semantics based on monads for scoped effects [Piróg et al. '18]
- Using the effect system for effect-dependent optimisations

  $\vdash M : X \mathbin{!} (o, \iota)$ and $\iota\,(\mathrm{op}) = \bot$ imply $\downarrow \mathrm{op}\;(V, M) \rightsquigarrow^{*} M$
- Refine the "broadcast everything everywhere" communication strategy
- In depth comparison with message-passing concurrency frameworks

THE CONCLUSION

THE CONCLUSION

THE CONCLUSION

- We have shown how to incorporate asynchrony within algebraic effects, by
  - decoupling operation calls into signals and interrupts, and
  - installing interrupt handlers and selectively blocking execution

THE CONCLUSION

- We have shown how to incorporate asynchrony within algebraic effects, by
  - decoupling operation calls into signals and interrupts, and
  - installing interrupt handlers and selectively blocking execution
- We have captured these ideas in the $\lambda_{æ}$-calculus
  - type-and-effect system, sub-effecting, and small-step operational semantics
  - type safety (reduction of open terms, hoisting $\uparrow$ past promises, sel. blocking)

THE CONCLUSION

- We have shown how to incorporate asynchrony within algebraic effects, by
  - decoupling operation calls into signals and interrupts, and
  - installing interrupt handlers and selectively blocking execution
- We have captured these ideas in the $\lambda_{æ}$-calculus
  - type-and-effect system, sub-effecting, and small-step operational semantics
  - type safety (reduction of open terms, hoisting ↑ past promises, sel. blocking)
- Examples ranging from preemptive multi-threading to remote function calls

THE CONCLUSION

- We have shown how to incorporate asynchrony within algebraic effects, by
  - decoupling operation calls into signals and interrupts, and
  - installing interrupt handlers and selectively blocking execution
- We have captured these ideas in the $\lambda_{æ}$-calculus
  - type-and-effect system, sub-effecting, and small-step operational semantics
  - type safety (reduction of open terms, hoisting $\uparrow$ past promises, sel. blocking)
- Examples ranging from preemptive multi-threading to remote function calls
- Agda formalisation of $\lambda_{æ}$ and prototype implementation Æff