Effect handler oriented programming

Sam Lindley

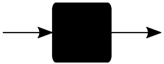
The University of Edinburgh

Huawei Joint Lab Meeting, Xi'an, April 2025

Effect handlers

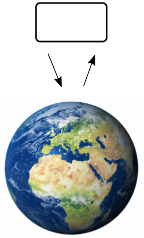
Effects

Programs as black boxes (Church-Turing model)?



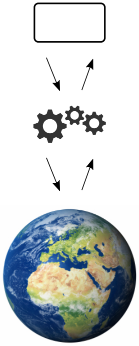
Effects

Programs must interact with their environment



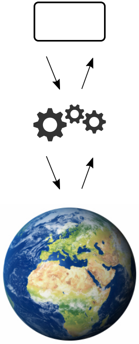
Effects

Programs must interact with their environment



Effects

Programs must interact with their environment

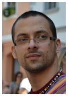


Effects are pervasive

- input/output
  user interaction
- concurrency
  web applications
- distribution
  cloud computing
- exceptions
  fault tolerance
- choice
  backtracking search

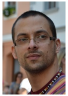
Typically ad hoc and hard-wired

Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009 (and ETAPS 2022 test of time award)

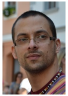
Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009 (and ETAPS 2022 test of time award)

Monads $\longrightarrow$ Algebraic Effects $\longrightarrow$ Effect Handlers

Effect handlers

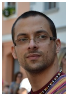
 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009 (and ETAPS 2022 test of time award)

Monads $\longrightarrow$ Algebraic Effects $\longrightarrow$ Effect Handlers

Composable and **customisable** user-defined interpretation of effects in general

Effect handlers

 Gordon Plotkin  Matija Pretnar

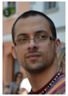
Handlers of algebraic effects, ESOP 2009 (and ETAPS 2022 test of time award)

Monads $\longrightarrow$ Algebraic Effects $\longrightarrow$ Effect Handlers

Composable and **customisable** user-defined interpretation of effects in general

Give programmer direct access to **context**
(c.f. resumable exceptions, monads, delimited control)

Effect handlers

 Gordon Plotkin  Matija Pretnar

Handlers of algebraic effects, ESOP 2009 (and ETAPS 2022 test of time award)

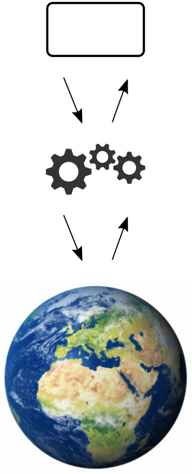
Monads $\longrightarrow$ Algebraic Effects $\longrightarrow$ Effect Handlers

Composable and **customisable** user-defined interpretation of effects in general

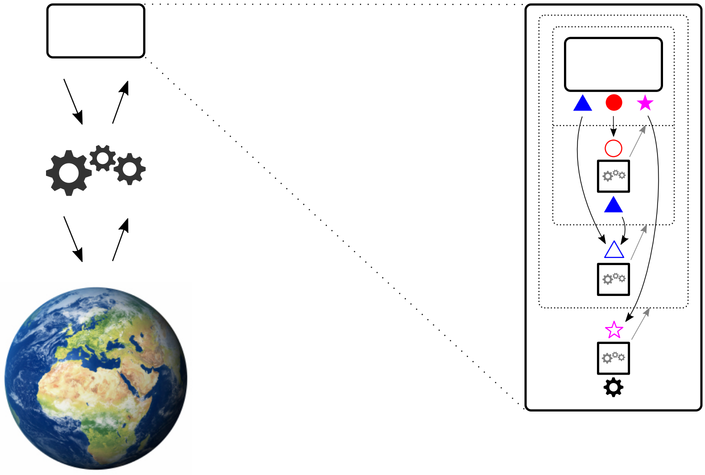
Give programmer direct access to **context**
(c.f. resumable exceptions, monads, delimited control)

Effect handlers in practice:
OCaml 5, GitHub (Semantic), Meta (React), Uber (Pyro), Wasm (WasmFX), ...

Effect handlers as composable user-defined operating systems



Effect handlers as composable user-defined operating systems



Effect handlers for operating systems

EIO — effects-based direct-style concurrent I/O stack for OCaml

Recently launched into space!

`https://github.com/ocaml-multicore/eio`

Composing UNIX with effect handlers

Foundations for programming and implementing effect handlers, Chapter 2

Daniel Hillerström, PhD thesis, The University of Edinburgh, 2022

`https://www.dhil.net/research/papers/thesis.pdf`

Effect handlers in space

- Parsimoni's SpaceOS deployed in SpaceX Transporter-13 payload
- Unikernel operating system built on OCaml 5
- Makes essential use of the EIO library

Example: cooperative lightweight threads

Effect signature

$$\begin{aligned}\mathrm{Coop} = \{&\mathsf{yield} : 1 \twoheadrightarrow 1,\\ &\mathsf{fork} : \mathrm{Thread} \twoheadrightarrow 1\}\end{aligned}$$

$$\mathrm{Thread} = [\mathrm{Coop}](1 \to 1)$$

Example: cooperative lightweight threads

Effect signature

$$\begin{aligned}\mathsf{Coop} &= \{\mathsf{yield} : 1 \twoheadrightarrow 1, \\ &\qquad \mathsf{fork} : \mathsf{Thread} \twoheadrightarrow 1\} \\ \mathsf{Thread} &= [\mathsf{Coop}](1 \to 1)\end{aligned}$$

A single cooperative program

$$\begin{aligned}&\mathsf{main} : \mathsf{Thread} \\ &\mathsf{main}\,() = \mathsf{print}\,(\text{``M1 ''});\, \mathsf{fork}\,(\mathbf{fun}\,() \to \mathsf{print}\,(\text{``A1 ''});\, \mathsf{yield}\,();\, \mathsf{print}\,(\text{``A2 ''}));\\ &\qquad\qquad \mathsf{print}\,(\text{``M2 ''});\, \mathsf{fork}\,(\mathbf{fun}\,() \to \mathsf{print}\,(\text{``B1 ''});\, \mathsf{yield}\,();\, \mathsf{print}\,(\text{``B2 ''}));\\ &\qquad\qquad \mathsf{print}\,(\text{``M3 ''})\end{aligned}$$

Example: cooperative lightweight threads

Handler

$$
\begin{array}{l}
\mathsf{cooperate} : \mathsf{List}\,(\mathsf{Thread}) \to 1 \\
\mathsf{cooperate}\,[\,] = (\,) \\
\mathsf{cooperate}\,(t :: ts) = \\
\quad \textbf{handle}\; t(\,)\; \textbf{with} \\
\qquad \begin{array}{ll}
\textbf{return}\,(\,) & \mapsto \mathsf{cooperate}\,(ts) \\
\langle \mathsf{yield}\,(\,) \to t \rangle & \mapsto \mathsf{cooperate}\,(ts \mathbin{+\!\!+} [t]) \\
\langle \mathsf{fork}\; t \to r \rangle & \mapsto \mathsf{cooperate}\,(ts \mathbin{+\!\!+} [t, r])
\end{array}
\end{array}
$$

Example: cooperative lightweight threads

Handler

$$
\begin{aligned}
&\mathsf{cooperate} : \mathsf{List}\ (\mathsf{Thread}) \to 1 \\
&\mathsf{cooperate}\,[] = () \\
&\mathsf{cooperate}\,(t :: ts) = \\
&\quad \textbf{handle}\ t()\ \textbf{with} \\
&\qquad \textbf{return}\ () \qquad\quad \mapsto \mathsf{cooperate}\,(ts) \\
&\qquad \langle \mathsf{yield}\ () \to t\rangle \mapsto \mathsf{cooperate}\,(ts \mathbin{+\!\!+} [t]) \\
&\qquad \langle \mathsf{fork}\ t \to r\rangle \quad \mapsto \mathsf{cooperate}\,(ts \mathbin{+\!\!+} [t, r])
\end{aligned}
$$

$$\mathsf{cooperate}\,[\mathsf{main}] \implies ()$$

M1 A1 M2 A2 B1 M3 B2

Example: cooperative lightweight threads

Handler

$$
\begin{aligned}
&\mathsf{cooperate} : \mathsf{List}\,(\mathsf{Thread}) \to 1\\
&\mathsf{cooperate}\,[\,] = ()\\
&\mathsf{cooperate}\,(t :: ts) =\\
&\quad \mathbf{handle}\; t()\;\mathbf{with}\\
&\qquad \mathbf{return}\,() \qquad\quad \mapsto \mathsf{cooperate}\,(ts)\\
&\qquad \langle \mathsf{yield}\,() \to t\rangle \mapsto \mathsf{cooperate}\,(ts \mathbin{+\!\!+} [t])\\
&\qquad \langle \mathsf{fork}\; t \to r\rangle \quad \mapsto \mathsf{cooperate}\,(ts \mathbin{+\!\!+} [r, t])
\end{aligned}
$$

Example: cooperative lightweight threads

Handler

$$
\begin{aligned}
&\mathsf{cooperate} : \mathsf{List}\,(\mathsf{Thread}) \to 1\\
&\mathsf{cooperate}\,[] = ()\\
&\mathsf{cooperate}\,(t :: ts) =\\
&\quad \textbf{handle}\; t()\; \textbf{with}\\
&\qquad \textbf{return}\,() \qquad\quad \mapsto \mathsf{cooperate}\,(ts)\\
&\qquad \langle \mathsf{yield}\,() \to t\rangle \mapsto \mathsf{cooperate}\,(ts +\!\!+ [t])\\
&\qquad \langle \mathsf{fork}\; t \to r\rangle \quad \mapsto \mathsf{cooperate}\,(ts +\!\!+ [r, t])
\end{aligned}
$$

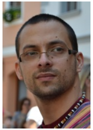
$$\mathsf{cooperate}\,[\mathsf{main}] \Longrightarrow ()$$

M1 M2 M3 A1 B1 A2 B2

Resources

EHOP web page
https://effect-handlers.org/

Jeremy Yallop's effects bibliography
https://github.com/yallop/effects-bibliography



Matija Pretnar's tutorial
"An introduction to algebraic effects and handlers", MFPS 2015

Libraries and languages

Effect handlers for C++

cpp-effects library
(with Dan Ghica, Maciej Piróg, Marcello Maroñas Bravo)
"High-level effect handlers in C++", OOPSLA 2022

- single-shot handlers
- commands (operations) are classes
- handlers are classes parameterised by commands they handle
- both unnamed and named handlers
- flat handlers (identity return clause)
- plain handler clauses (tail-resumptive)
- 'no resume' handler clauses (exceptions)

Implementation

- backend — `boost.context` fibers
- nested stack (one stacklet / fiber per handler)
- pre-allocation of resumptions
- reference counting
- move constructors as a crude alternative to substructural types
- ‘no manage’ optimisation (when handler and resumptions do not escape)

Example: cooperative lightweight threads in C++

```
struct Yield : eff::command<> { };
struct Fork : eff::command<> {
  std::function<void()> proc;
};

void yield() {
  eff::invoke_command(Yield{});
}
void fork(std::function<void()> proc) {
  eff::invoke_command(Fork{{}, proc});
}

void mainThread() {
  std::cout << "M1 "; fork([=]() {std::cout << "A1 "; yield(); std::cout << "A2 "});
  std::cout << "M2 "; fork([=]() {std::cout << "B1 "; yield(); std::cout << "B2 "});
  std::cout << "M3 ";
}
```

Example: cooperative lightweight threads in C++

```cpp
using Res = eff::resumption<void()>;
class Scheduler : public eff::handler<void, void, Yield, Fork> {
public:
  static void Start(std::function<void()> f) {
    queue.push_back(eff::wrap<Scheduler>(f));
    while (!queue.empty()) {
      Res resumption = std::move(queue.front());
      queue.pop_front();
      std::move(resumption).resume();
    }
  }
private:
  static std::list<Res> queue;
  void handle_command(Yield, Res r) override {
    queue.push_back(std::move(r));
  }
  void handle_command(Fork f, Res r) override {
    queue.push_back(eff::wrap<Scheduler>(f.proc));
    queue.push_back(std::move(r));
  }
  void handle_return() override { }
};
```

Example: cooperative lightweight threads in C++

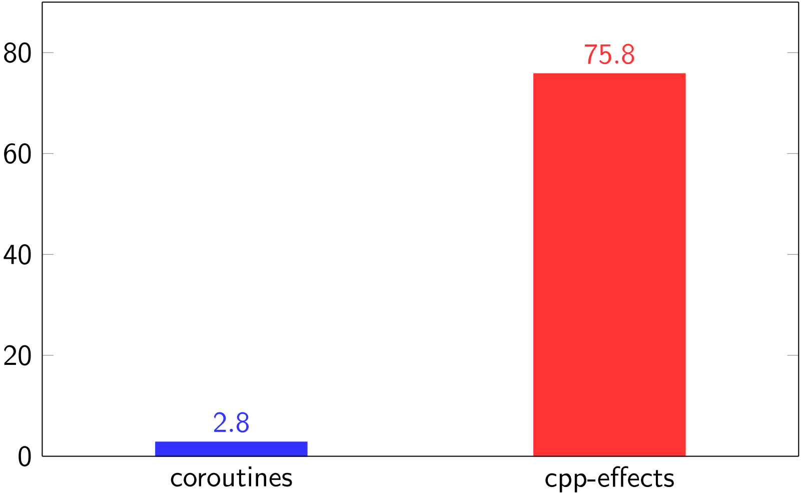
```
int main() {
  Scheduler::Start(mainThread);
}
```

Example: cooperative lightweight threads in C++

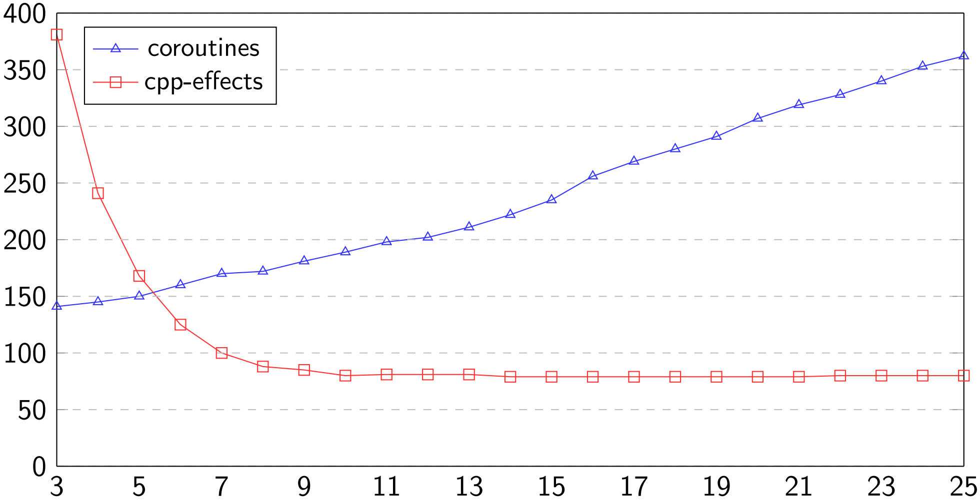
```
int main() {
  Scheduler::Start(mainThread);
}
```

M1 A1 M2 A2 B1 M3 B2

Generating a number (in ns)



Recursive tree traversal (ns per node)



Effect handlers for C

libseff library
(with Mario Alvarez-Picallo, Teodoro Freund, Dan Ghica)
"Effect handlers for C via coroutines", OOPSLA 2024

- Based on **mutable coroutines** rather than immutable continuations
- Stack resizing via **segmented stacks** (or **overcommitting virtual memory**)
- No special dispatch mechanism for effects (**request objects** + switch instead)
- **x64** and **ARM** backends

Example: cooperative lightweight threads in C

```
DEFINE_EFFECT(fork, 2, void, { void *(*fn)(void *); void *arg; });
DEFINE_EFFECT(yield, 3, void, {});

void *ta(void* param) {
  printf("%s", "A1 "); yield(); printf("%s", "A2 ")
}

void *tb(void* param) {
  printf("%s", "B1 "); yield(); printf("%s", "B2 ")
}

void *mainThread(void* param) {
  printf("%s", "M1 "); PERFORM(fork, ta, null);
  printf("%s", "M2 "); PERFORM(fork, tb, null);
  printf("%s", "M3 ");
}
```

Example: cooperative lightweight threads in C

```
void with_scheduler(seff_coroutine_t *initial_coroutine) {
  effect_set handles_scheduler = HANDLES(yield) | HANDLES(fork);
  tl_queue_t queue;
  tl_queue_init(&queue, 5);
  tl_queue_push(&queue, initial_coroutine);
  while (!tl_queue_empty(&queue)) {
    seff_coroutine_t *next = (seff_coroutine_t *)tl_queue_steal(&queue);
    seff_request_t req = seff_resume(next, NULL, handles_scheduler);
    switch (req.effect) {
      CASE_EFFECT(req, yield, { tl_queue_push(&queue, (struct task_t *)next); break; })
      CASE_EFFECT(req, fork, {
        seff_coroutine_t *new = seff_coroutine_new(payload.fn, payload.arg);
        tl_queue_push(&queue, (struct task_t *)new);
        tl_queue_push(&queue, (struct task_t *)next);
        break; })
      CASE_RETURN(req, { seff_coroutine_delete(next); break; })
    }
  }
}
```

Example: cooperative lightweight threads in C

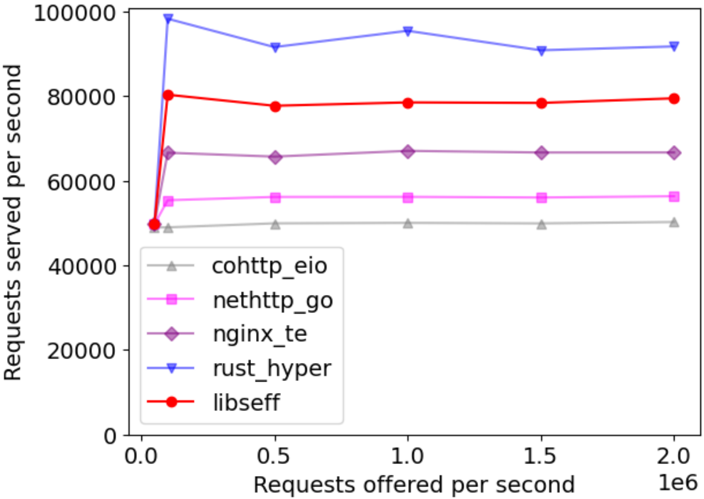
```
int main(void) {
  with_scheduler(seff_coroutine_new(mainThread, (void*)0)); return 0;
}
```

Example: cooperative lightweight threads in C

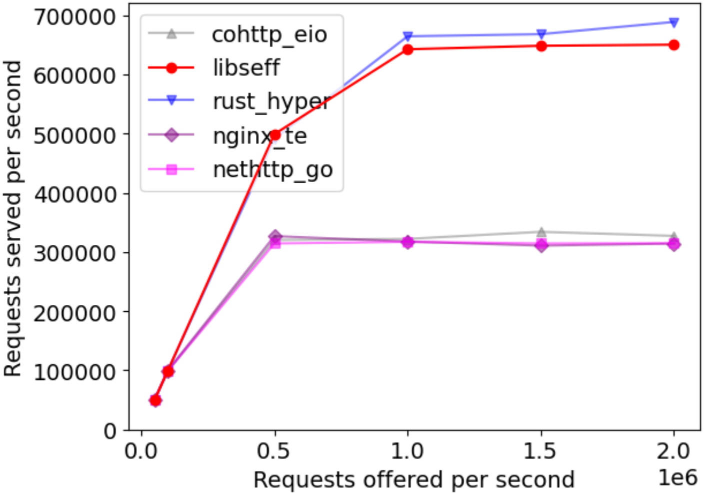
```
int main(void) {
  with_scheduler(seff_coroutine_new(mainThread, (void*)0)); return 0;
}
```

M1 A1 M2 A2 B1 M3 B2

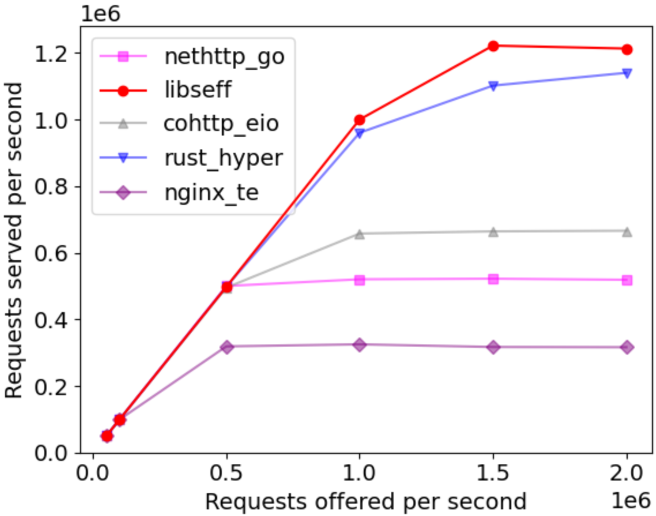
Web server benchmark (1 OS thread)



Web server benchmark (8 OS thread)



Web server benchmark (16 OS thread)



Effect handlers in Cangjie

Cangjie is a new general-purpose programming language developed at Huawei

From the documentation: "Cangjie embraces a multi-paradigm approach, supporting functional, imperative, and object-oriented programming styles"

Exploits OO interfaces much like the **cpp-effects** C++ library does

Effect handlers implemented on top of existing pre-emptive concurrency features

Potential applications include **dependency injection** and **reactive programming**

Presented at PLDI 2024 and recent coffee house tech talks by Magnus Morton and Mario Alvarez-Picallo

Example: cooperative lightweight threads in Cangjie

```
class Yield <: Command<Unit> {}
class Fork <: Command<Unit> {
  Fork(let fn: () -> Unit) {}
}

func mainThread() {
  println("M1")
  perform Fork({ =>
    println("A1"); perform Yield(); println("A2")
  })
  println("M2")
  perform Fork({ =>
    println("B1"); perform Yield(); println("B2")
  })
  println("M3")
}
```

Example: cooperative lightweight threads in Cangjie

```
func cooperate(threads: List<() -> Unit>) {
  match (threads) {
    case Nil => ()
    case Cons(head, rest) =>
      try {
        head ()
      } handle (_: Yield, next: Resumption<Unit, Unit>) {
        cooperate(rest.append({ => resume next }))
      } handle (f: Fork, next: Resumption<Unit, Unit>) {
        cooperate(rest.append(f.fn).append({ => resume next }))
      } finally {
        cooperate(rest)
      }
  }
}
```

Example: cooperative lightweight threads in Cangjie

```
func main() {
  cooperate(Cons(mainThread, Nil))
}
```

Example: cooperative lightweight threads in Cangjie

```
func main() {
  cooperate(Cons(mainThread, Nil))
}
```

M1 A1 M2 A2 B1 M3 B2

EPOCH

EPOCH: effectful programming on capability hardware

Sam Lindley

Ian Stark

Brian Campbell

Wilmer Ricciotti

- **Effect Handlers**: powerful high-level programming abstraction with strong properties
- **Implementation**: through libraries, program transformation, compilation
- **Challenge**: all those good properties get translated away
- **Opportunity**: advances in CHERI CPU architecture with hardware capabilities
- **Goal**: use capability hardware to directly express effect handlers

Two strands of work: **implementations** and **foundations**

Capability implementation

Ported existing C/C++ effect handler libraries to use CHERI

- **libmpeff**/**libmprompt**
- **cpp-effects** [Ghica et al, OOPSLA 2022], C++ library based on **boost.context**
- **libseff** [Alvarez-Picallo et al, OOPSLA 2024], C library

Added CHERI support to Koka

All run on CHERI hardware with capabilities for all pointers demonstrating:

- memory protection
- control flow integration

Handlers as compartment boundaries

Can we use capabilities with handlers to

- constrain effects?
- recover from failure?

Investigating the use of capabilities to restrict external calls to libraries and OS, where

- handlers control the effects available
- handlers can use this to recover from crashes

Plan: experiment with an old version of a common library (e.g. for image decoding) to ensure safe recovery from a known bug

AsmFX

- Typical implementations of effect handlers:
  - first-class functions
  - closures
  - continuations
  - prompts
- These **need to be translated** further for real compilation to CPU architectures
- What is the simplest abstraction over an instruction set we need to implement effect handlers?
- Can we actually implement some of those abstractions on top of effect handlers?

Source language: a first-order functional language with handlers

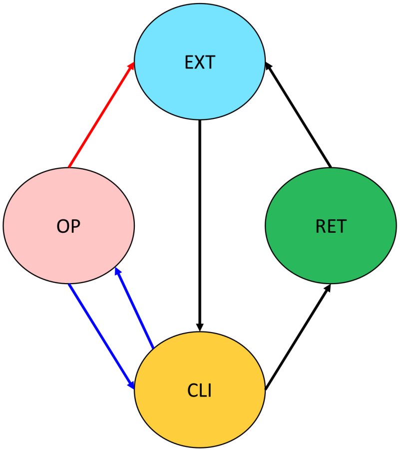
```
let res = handle
{
    // ...
    op(n);
    // ...
    0
}
with
{
    return(x) ->
        // ...
    op(n) k when n >= 0 ->
        // ...
        resume k()
    op(n) k when n < 0 ->
        // ...
        42
}
```

External code

Client code

Handler (return) code

Handler (operation) code



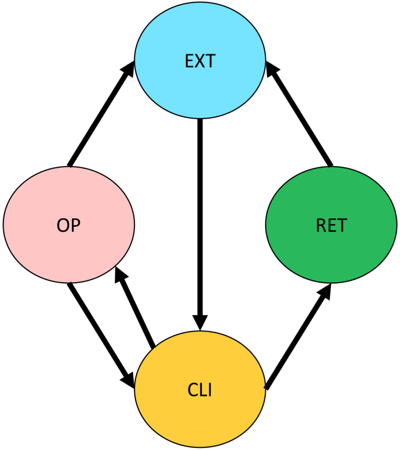
AsmFX: assembly language with effect context manipulating instructions

```
ext:
    enter cli, H
    ; ...
```

```
cli:
    ; ...
    do #op
    ; ...
    return
```

```
H.ret:
    ; ...
    exit
```

```
H.op:
    ; ...
    reenter k
    ; ...
    exit
```



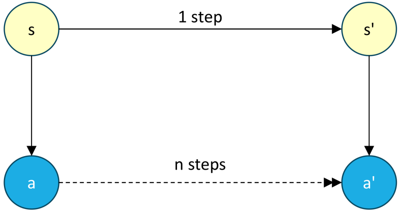
Compilation soundness

A source language configuration $s = (M, \gamma, \kappa)$

- Computation M to evaluate
- Environment γ
- Continuation κ

An AsmFX configuration $a = (\Xi, \Theta, C)$

- Memory Ξ (holding the code, read only)
- Register file Θ (holding the data)
- Effect context C (stack of handlers)



A validity relation $a \vDash s$ encodes the correspondence between source code and its compiled memory image.

ReactFX

ReactFX: reactive programming with effects and handlers

Project due to start in September 2025 and will fund one PhD student for 3.5 years

- **Foundations**: unify synchronous effects from the programming language with asynchronous events from the environment
- **Implementations**: experiment with research languages, e.g., Links, Koka, OCaml
- **Case studies**: ReactJS-style web applications, spreadsheets, etc.
- **Effect typing**: exploit for optimisation and modularity
- **Incremental updates**: use effect handlers to abstract over incremental updates to virtual DOM
- **Pre-emptive concurrency**: synergy with Cangjie implementation of effect handlers