Effect Handlers All the Way Down

Daniel Hillerström

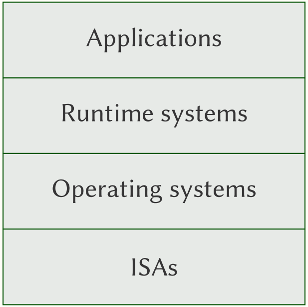
Computing Systems Laboratory
Zurich Research Center
Huawei Technologies, Switzerland

July 4, 2024

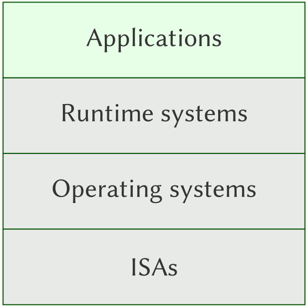
Global Software Technology Summit
Edinburgh, Scotland, United Kingdom

`https://dhil.net`

The Software Stack



The Software Stack: Applications



Control idioms

. . .

Non-local control idioms are pervasive

- Async/await (e.g. C++, C#, Dart, JavaScript, Rust, Swift)
- Coroutines (e.g. C++, Kotlin, Python, Swift)
- Lightweight threads (e.g. Erlang, Go, Haskell, Java, Swift)
- Yield-style generators (e.g. C#, Dart, Haskell, JavaScript, Kotlin, Python)

Control idioms

OCaml JS . . .

Non-local control idioms are pervasive

- Async/await (e.g. C++, C#, Dart, JavaScript, Rust, Swift)
- Coroutines (e.g. C++, Kotlin, Python, Swift)
- Lightweight threads (e.g. Erlang, Go, Haskell, Java, Swift)
- Yield-style generators (e.g. C#, Dart, Haskell, JavaScript, Kotlin, Python)

Instances of a general phenomenon

- First-class continuations (e.g. Haskell, Java, OCaml, Scheme)

Control idioms

. . .

Non-local control idioms are pervasive

- Async/await (e.g. C++, C#, Dart, JavaScript, Rust, Swift)
- Coroutines (e.g. C++, Kotlin, Python, Swift)
- Lightweight threads (e.g. Erlang, Go, Haskell, Java, Swift)
- Yield-style generators (e.g. C#, Dart, Haskell, JavaScript, Kotlin, Python)

Instances of a general phenomenon

- First-class continuations (e.g. Haskell, Java, OCaml, Scheme)

Structured programming with continuations

- Effect handlers are a structured facility for programming with continuations
- Composable control idioms as user-defined libraries
- Seamless interoperability with native effects

Yield-style generators (1)

Effect interface

$$\mathsf{Gen} = \{\mathsf{Yield} : \mathsf{Int} \twoheadrightarrow \mathsf{Void}\}$$

Yield-style generators (1)

Effect interface

$$\text{Gen} = \{\text{Yield} : \text{Int} \twoheadrightarrow \text{Void}\}$$

Integer generator — effectful function

$$\begin{aligned}&\text{ints} : \text{Int} \to \text{Void}!\text{Gen}\\ &\text{ints}\ i = \textbf{do}\ \text{Yield}\ i; \text{ints}\ (i + 1)\end{aligned}$$

Yield-style generators (1)

Effect interface

$$\text{Gen} = \{\text{Yield} : \text{Int} \twoheadrightarrow \text{Void}\}$$

Integer generator — effectful function

$$\begin{aligned}&\text{ints} : \text{Int} \to \text{Void}!\text{Gen}\\&\text{ints}\ i = \mathbf{do}\ \text{Yield}\ i; \text{ints}\ (i+1)\end{aligned}$$

Accumulator — linear handler

$$\begin{aligned}&\text{sumUp} : \text{Int} \to \text{Void}!\text{Gen} \Rightarrow \text{Int}\\&\text{sumUp}\ n = \{\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\}\end{aligned}$$

Yield-style generators (1)

Effect interface

$$\mathrm{Gen} = \{\mathrm{Yield} : \mathrm{Int} \twoheadrightarrow \mathrm{Void}\}$$

Integer generator — effectful function

$$\begin{aligned}&\mathrm{ints} : \mathrm{Int} \to \mathrm{Void}!\mathrm{Gen}\\&\mathrm{ints}\ i = \mathbf{do}\ \mathrm{Yield}\ i; \mathrm{ints}\ (i+1)\end{aligned}$$

Accumulator — linear handler

$$\begin{aligned}&\mathrm{sumUp} : \mathrm{Int} \to \mathrm{Void}!\mathrm{Gen} \Rightarrow \mathrm{Int}\\&\mathrm{sumUp}\ n = \{\ \langle\!\langle \mathrm{Yield}\ i \twoheadrightarrow r \rangle\!\rangle\ \mapsto\ \mathbf{if}\ n > 0\ \mathbf{then}\ \mathrm{decr}\ n; i + r\ \langle\rangle\\&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{else}\ 0\\&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \}\end{aligned}$$

Yield-style generators (1)

Effect interface

$$\text{Gen} = \{\text{Yield} : \text{Int} \twoheadrightarrow \text{Void}\}$$

Integer generator — effectful function

$$\begin{aligned}&\text{ints} : \text{Int} \to \text{Void}!\text{Gen}\\ &\text{ints}\ i = \mathbf{do}\ \text{Yield}\ i; \text{ints}\ (i+1)\end{aligned}$$

Accumulator — linear handler

$$\begin{aligned}&\text{sumUp} : \text{Int} \to \text{Void}!\text{Gen} \Rightarrow \text{Int}\\ &\text{sumUp}\ n = \{\ \langle\!\langle \text{Yield}\ i \twoheadrightarrow r \rangle\!\rangle \mapsto \mathbf{if}\ n > 0\ \mathbf{then}\ \text{decr}\ n; i + r\ \langle\rangle\\ &\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{else}\ 0\\ &\qquad\qquad\quad\ \mathbf{return}\ \langle\rangle \mapsto 0\ \}\end{aligned}$$

Yield-style generators (1)

Effect interface

$$\text{Gen} = \{\text{Yield} : \text{Int} \twoheadrightarrow \text{Void}\}$$

Integer generator — effectful function

$$\begin{aligned} &\text{ints} : \text{Int} \to \text{Void}!\text{Gen} \\ &\text{ints}\ i = \mathbf{do}\ \text{Yield}\ i; \text{ints}\ (i+1) \end{aligned}$$

Accumulator — linear handler

$$\begin{aligned} &\text{sumUp} : \text{Int} \to \text{Void}!\text{Gen} \Rightarrow \text{Int} \\ &\text{sumUp}\ n = \{\ \langle\!\langle \text{Yield}\ i \twoheadrightarrow r \rangle\!\rangle \mapsto \mathbf{if}\ n > 0\ \mathbf{then}\ \text{decr}\ n; i + r\ \langle\rangle \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ 0 \\ &\qquad\qquad\qquad \mathbf{return}\ \langle\rangle \mapsto 0\ \} \end{aligned}$$

Example

$$\text{sumUp}\ 10\ (\text{ints}\ 0) \rightsquigarrow 55$$

Lightweight threads

Effect interface

$$\mathsf{Lwt} = \{\mathsf{Fork} : (\mathsf{Void}!\mathsf{Lwt}) \twoheadrightarrow \mathsf{Void}, \mathsf{Interrupt} : \mathsf{Void} \twoheadrightarrow \mathsf{Void}\}$$

Lightweight threads

Effect interface

$$\text{Lwt} = \{\text{Fork} : (\text{Void}!\text{Lwt}) \twoheadrightarrow \text{Void}, \text{Interrupt} : \text{Void} \twoheadrightarrow \text{Void}\}$$

Scheduler — escaping continuation

$$\begin{array}{l} \text{schedule} : \text{Queue} \to \alpha!\text{Lwt} \Rightarrow \text{Void} \\ \text{schedule}\ q = \{ \\ \\ \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \end{array}$$

Lightweight threads

Effect interface

$$\mathsf{Lwt} = \{\mathsf{Fork} : (\mathsf{Void}!\mathsf{Lwt}) \twoheadrightarrow \mathsf{Void}, \mathsf{Interrupt} : \mathsf{Void} \twoheadrightarrow \mathsf{Void}\}$$

Scheduler — escaping continuation

$$\begin{aligned}
&\mathsf{schedule} : \mathsf{Queue} \to \alpha!\mathsf{Lwt} \Rightarrow \mathsf{Void} \\
&\mathsf{schedule}\ q = \{ \\
&\qquad \langle\!\langle \mathsf{Fork}\ f \twoheadrightarrow r \rangle\!\rangle \ \mapsto\ \mathsf{runNext}\ ((\mathsf{schedule}\ q\ f) :: (r\ \langle\rangle) :: q) \\
&\qquad\qquad\qquad\qquad \}
\end{aligned}$$

Lightweight threads

Effect interface

$$\mathrm{Lwt} = \{\mathrm{Fork} : (\mathrm{Void}!\mathrm{Lwt}) \twoheadrightarrow \mathrm{Void}, \mathrm{Interrupt} : \mathrm{Void} \twoheadrightarrow \mathrm{Void}\}$$

Scheduler — escaping continuation

$$\begin{aligned}
&\mathrm{schedule} : \mathrm{Queue} \to \alpha!\mathrm{Lwt} \Rightarrow \mathrm{Void}\\
&\mathrm{schedule}\ q = \{\\
&\qquad \langle\!\langle \mathrm{Fork}\ f \twoheadrightarrow r \rangle\!\rangle \ \mapsto\ \mathrm{runNext}\ ((\mathrm{schedule}\ q\ f) :: (r\ \langle\rangle) :: q)\\
&\quad \langle\!\langle \mathrm{Interrupt}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \ \mapsto\ \mathrm{runNext}\ ((r\ \langle\rangle) :: q)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \}
\end{aligned}$$

Lightweight threads

Effect interface

$$\text{Lwt} = \{\text{Fork} : (\text{Void}!\text{Lwt}) \twoheadrightarrow \text{Void}, \text{Interrupt} : \text{Void} \twoheadrightarrow \text{Void}\}$$

Scheduler — escaping continuation

$$\begin{aligned}
&\text{schedule} : \text{Queue} \to \alpha!\text{Lwt} \Rightarrow \text{Void} \\
&\text{schedule}\ q = \{ \\
&\quad \langle\!\langle \text{Fork}\ f \twoheadrightarrow r \rangle\!\rangle \mapsto \text{runNext}\ ((\text{schedule}\ q\ f) :: (r\ \langle\rangle) :: q) \\
&\quad \langle\!\langle \text{Interrupt}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto \text{runNext}\ ((r\ \langle\rangle) :: q) \\
&\quad \textbf{return}\ _ \mapsto \text{runNext}\ q\ \}
\end{aligned}$$

Lightweight threads

Effect interface

$$\mathsf{Lwt} = \{\mathsf{Fork} : (\mathsf{Void}!\mathsf{Lwt}) \twoheadrightarrow \mathsf{Void}, \mathsf{Interrupt} : \mathsf{Void} \twoheadrightarrow \mathsf{Void}\}$$

Scheduler — escaping continuation

$$\begin{array}{l}
\mathsf{schedule} : \mathsf{Queue} \to \alpha!\mathsf{Lwt} \Rightarrow \mathsf{Void} \\
\mathsf{schedule}\ q = \{ \\
\qquad \begin{array}{rcl}
\langle\!\langle \mathsf{Fork}\ f \twoheadrightarrow r \rangle\!\rangle & \mapsto & \mathsf{runNext}\ ((\mathsf{schedule}\ q\ f) :: (r\ \langle\rangle) :: q) \\
\langle\!\langle \mathsf{Interrupt}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle & \mapsto & \mathsf{runNext}\ ((r\ \langle\rangle) :: q) \\
\mathbf{return}\ _ & \mapsto & \mathsf{runNext}\ q\ \}
\end{array}
\end{array}$$

Scheduling policy — regular function

$$\begin{array}{l}
\mathsf{runNext} : \mathsf{Queue} \to \mathsf{Void} \\
\mathsf{runNext}\ q = \mathbf{case}\ \mathsf{pop}\ q\ \{\ \mathsf{None} \mapsto \langle\rangle \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Some}\ r \mapsto r\ \langle\rangle\ \}
\end{array}$$

Lightweight threads

Effect interface

$$\text{Lwt} = \{\text{Fork} : (\text{Void}!\text{Lwt}) \twoheadrightarrow \text{Void}, \text{Interrupt} : \text{Void} \twoheadrightarrow \text{Void}\}$$

Scheduler — escaping continuation

$$\begin{aligned}
&\text{schedule} : \text{Queue} \to \alpha!\text{Lwt} \Rightarrow \text{Void} \\
&\text{schedule}\ q = \{ \\
&\qquad \langle\!\langle \text{Fork}\ f \twoheadrightarrow r \rangle\!\rangle \mapsto \text{runNext}\ ((\text{schedule}\ q\ f) :: (r\ \langle\rangle) :: q) \\
&\qquad \langle\!\langle \text{Interrupt}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto \text{runNext}\ ((r\ \langle\rangle) :: q) \\
&\qquad \mathbf{return}\ _ \mapsto \text{runNext}\ q\ \}
\end{aligned}$$

Scheduling policy — regular function

$$\begin{aligned}
&\text{runNext} : \text{Queue} \to \text{Void} \\
&\text{runNext}\ q = \mathbf{case}\ \text{pop}\ q\ \{\ \text{None} \mapsto \langle\rangle \\
&\qquad\qquad\qquad\qquad\qquad\quad \text{Some}\ r \mapsto r\ \langle\rangle\ \}
\end{aligned}$$

Example

$$\begin{aligned}
&\text{task}\ n\ \langle\rangle = \mathbf{do}\ \text{Yield}\ n; \mathbf{do}\ \text{Interrupt}\ \langle\rangle; \text{task}\ (n+1)\ \langle\rangle \\
&\text{sumUp}\ 10\ (\text{schedule}\ []\ (\mathbf{do}\ \text{Fork}\ (\text{task}\ 10); \mathbf{do}\ \text{Fork}\ (\text{task}\ (-10)))) \leadsto 20
\end{aligned}$$

Lightweight threads

Effect interface

$$\mathsf{Lwt} = \{\mathsf{Fork} : (\mathsf{Void}!\mathsf{Lwt}) \twoheadrightarrow \mathsf{Void}, \mathsf{Interrupt} : \mathsf{Void} \twoheadrightarrow \mathsf{Void}\}$$

Scheduler — escaping continuation

$$\begin{aligned}
&\mathsf{schedule} : \mathsf{Queue} \to \alpha!\mathsf{Lwt} \Rightarrow \mathsf{Void}\\
&\mathsf{schedule}\ q = \{\\
&\qquad \langle\!\langle \mathsf{Fork}\ f \twoheadrightarrow r \rangle\!\rangle \mapsto \mathsf{runNext}\ ((\mathsf{schedule}\ q\ f) :: (r\ \langle\rangle) :: q)\\
&\qquad \langle\!\langle \mathsf{Interrupt}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto \mathsf{runNext}\ ((r\ \langle\rangle) :: q)\\
&\qquad \mathbf{return}\ _ \mapsto \mathsf{runNext}\ q\ \}
\end{aligned}$$

Scheduling policy — regular function

$$\begin{aligned}
&\mathsf{runNext} : \mathsf{Queue} \to \mathsf{Void}\\
&\mathsf{runNext}\ q = \mathbf{case}\ \mathsf{pop}\ q\ \{\ \mathsf{None} \mapsto \langle\rangle\\
&\qquad\qquad\qquad\qquad\quad \mathsf{Some}\ r \mapsto r\ \langle\rangle\ \}
\end{aligned}$$

Example

$$\begin{aligned}
&\mathsf{task}\ n\ \langle\rangle = \mathbf{do}\ \mathsf{Yield}\ n; \mathbf{do}\ \mathsf{Interrupt}\ \langle\rangle; \mathsf{task}\ (n+1)\ \langle\rangle\\
&\mathsf{schedule}\ []\ (\mathsf{sumUp}\ 10\ (\mathbf{do}\ \mathsf{Fork}\ (\mathsf{task}\ 10); \mathbf{do}\ \mathsf{Fork}\ (\mathsf{task}\ (-10)))) \rightsquigarrow \langle\rangle
\end{aligned}$$

Lightweight threads

Effect interface

$$\text{Lwt} = \{\text{Fork} : (\text{Void!Lwt}) \twoheadrightarrow \text{Void}, \text{Interrupt} : \text{Void} \twoheadrightarrow \text{Void}\}$$

Scheduler — escaping continuation

$$\begin{aligned}
&\text{schedule} : \text{Queue} \to \alpha!\text{Lwt} \Rightarrow \text{Void} \\
&\text{schedule}\ q = \{ \\
&\quad \langle\!\langle \text{Fork}\ f \twoheadrightarrow r \rangle\!\rangle \ \mapsto\ \text{runNext}\ ((\text{schedule}\ q\ f) :: (r\ \langle\rangle) :: q) \\
&\quad \langle\!\langle \text{Interrupt}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \ \mapsto\ \text{runNext}\ ((r\ \langle\rangle) :: q) \\
&\quad \textbf{return}\ _ \ \mapsto\ \text{runNext}\ q\ \}
\end{aligned}$$

Scheduling policy — regular function

$$\begin{aligned}
&\text{runNext} : \text{Queue} \to \text{Void} \\
&\text{runNext}\ q = \textbf{case}\ \text{pop}\ q\ \{\ \text{None} \mapsto \langle\rangle \\
&\qquad\qquad\qquad\qquad\qquad\ \ \text{Some}\ r \mapsto r\ \langle\rangle\ \}
\end{aligned}$$

Example

$$\begin{aligned}
&\text{task}\ n\ \langle\rangle = \textbf{do}\ \text{Yield}\ n; \textbf{do}\ \text{Interrupt}\ \langle\rangle; \text{task}\ (n+1)\ \langle\rangle \\
&\text{schedule}\ []\ (\text{sumUp}\ 10\ (\textbf{do}\ \text{Fork}\ (\text{task}\ 10); \textbf{do}\ \text{Fork}\ (\text{task}\ (-10)))) \rightsquigarrow \langle\rangle
\end{aligned}$$

Applications

Applications in the wild

- Concurrency
  - Direct-style asynchronous I/O (e.g. OCaml)
  - Efficient user interface rendering (e.g. Facebook's React)
- Distribution
  - Content-addressed programming (e.g. Unison)
  - Build systems and deployment (e.g. srclang)
- Modular interpretation
  - Probabilistic programming (e.g. Pyro)
  - Flexible data manipulation (e.g. GitHub's SEMANTIC)

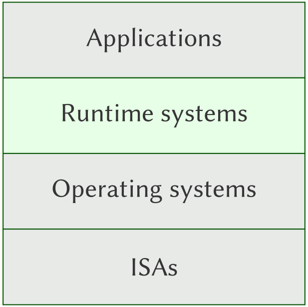
Applications

Applications in the wild

- Concurrency
  - Direct-style asynchronous I/O (e.g. OCaml)
  - Efficient user interface rendering (e.g. Facebook's React)
- Distribution
  - Content-addressed programming (e.g. Unison)
  - Build systems and deployment (e.g. srclang)
- Modular interpretation
  - Probabilistic programming (e.g. Pyro)
  - Flexible data manipulation (e.g. GitHub's SEMANTIC)

We have only scratched the surface for what effect handler oriented programming has to offer!

The Software Stack: Runtime systems

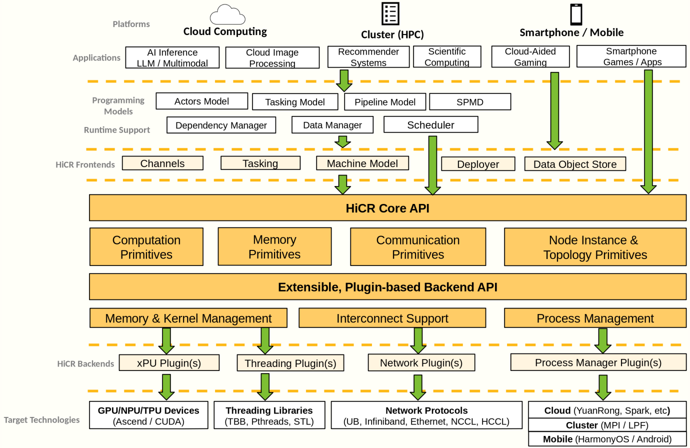


HiSilicon Common Runtime (HiCR)

(joint work with Sergio M. Martin, Kiril Dichev, Luca Terracciano, Orestis Korakitis, and Albert-Jan Yzelmann)

Executive summary

- A unified API for building portable runtime systems, providing seamless and efficient access to new technologies



(figure courtesy of Sergio M. Martin)

HiSilicon Common Runtime (HiCR)

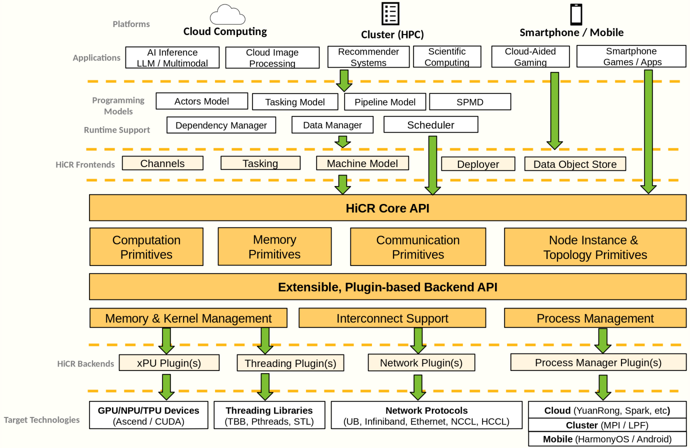
(joint work with Sergio M. Martin, Kiril Dichev, Luca Terracciano, Orestis Korakitis, and Albert-Jan Yzelmann)

Executive summary

- A unified API for building portable runtime systems, providing seamless and efficient access to new technologies

Key components

- **Core**: target-agnostic API for low-level operations (e.g. memcpy)



(figure courtesy of Sergio M. Martin)

HiSilicon Common Runtime (HiCR)

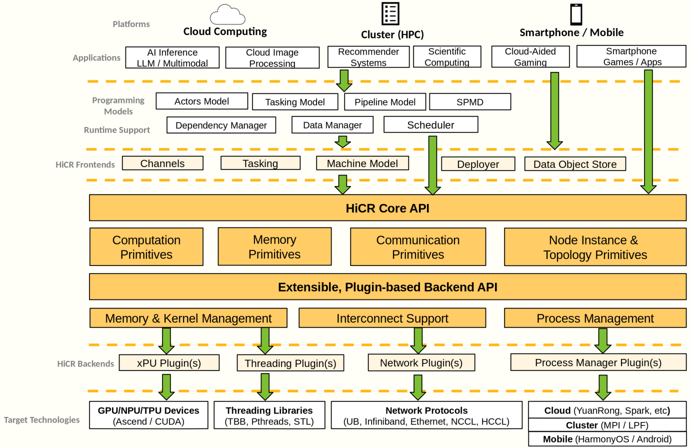
(joint work with Sergio M. Martin, Kiril Dichev, Luca Terracciano, Orestis Korakitis, and Albert-Jan Yzelmann)

Executive summary

- A unified API for building portable runtime systems, providing seamless and efficient access to new technologies

Key components

- **Core**: target-agnostic API for low-level operations (e.g. memcpy)
- **Frontend**: Higher-level building blocks for applications



(figure courtesy of Sergio M. Martin)

HiSilicon Common Runtime (HiCR)

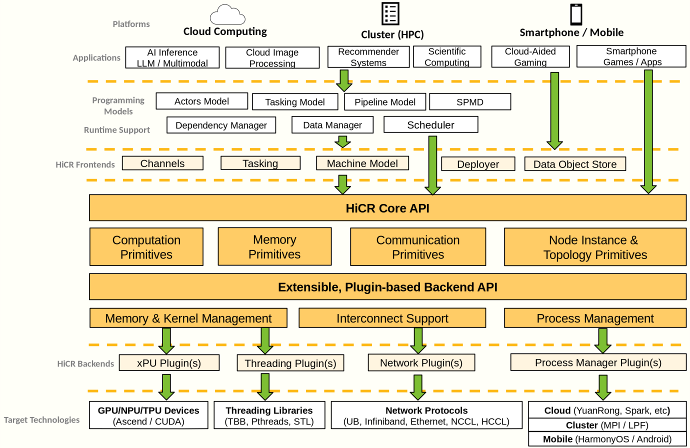
(joint work with Sergio M. Martin, Kiril Dichev, Luca Terracciano, Orestis Korakitis, and Albert-Jan Yzelmann)

Executive summary

- A unified API for building portable runtime systems, providing seamless and efficient access to new technologies

Key components

- **Core**: target-agnostic API for low-level operations (e.g. memcpy)
- **Frontend**: Higher-level building blocks for applications
- **Backend**: Extensible plugin-based API for integrating computational fabrics

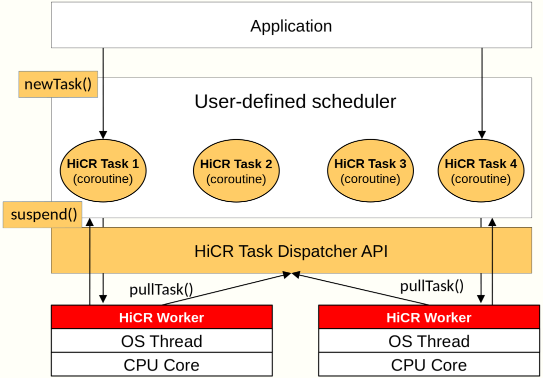


(figure courtesy of Sergio M. Martin)

Application-specific scheduling

Customised task scheduling

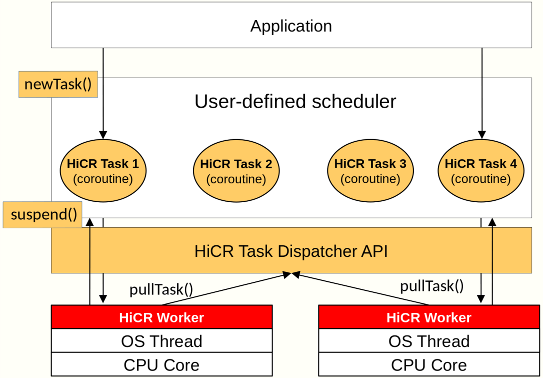
- Domain-tailored task scheduler
- HiCR coroutines interact with their environment
  - **`newTask`**: spawn new coroutine
  - **`suspend`**: yield control



Application-specific scheduling

Customised task scheduling

- Domain-tailored task scheduler **aka handler**
- HiCR coroutines interact with their environment **via effects**
  - **`newTask`**: spawn new coroutine
  - **`suspend`**: yield control



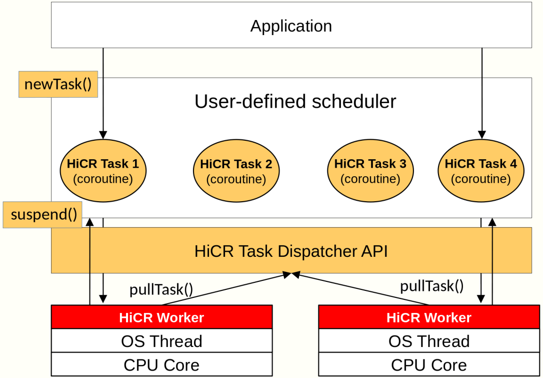
Application-specific scheduling

Customised task scheduling

- Domain-tailored task scheduler **aka handler**
- HiCR coroutines interact with their environment **via effects**
  - **`newTask`**: spawn new coroutine
  - **`suspend`**: yield control

Handlers yield benefits

- It just works™
- Seamless composition with builtin effects (e.g. exceptions)
- Anecdotally, easy fancy control programming



More effects in HiCR

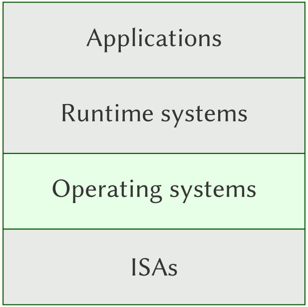
Growing demand for effect handlers

- Bespoke frontend for effect handlers
- Core building blocks for effect handlers
- Custom stack allocation policies
- Support for stackless coroutines

Liberating parallel programming

- No support for colocation (e.g. needed for multi-tenant systems)
- Virtualise parallel resources (e.g. ‘virtual memory’ for parallel programming)

The Software Stack: Operating systems



Effect handlers as composable microkernels

System calls and kernels

A **system call** is an abstract operation, whose implementation is provided by the **kernel**.

Effect handlers as composable microkernels

System calls and kernels

A **system call** is an abstract operation, whose implementation is provided by the **kernel**.

Interpretation via handlers

System calls *as* effectful operations
and
Kernels *as* effect handlers

Effect handlers as composable microkernels

System calls and kernels

A **system call** is an abstract operation, whose implementation is provided by the **kernel**.

Interpretation via handlers

System calls *as* effectful operations
and
Kernels *as* effect handlers

Composable interactive microkernels

- Domain-specific interactive (micro)kernels
- Virtualisation of operating system facilities
- Composability inherited from handlers

Dynamic multi user environments

Effect interface

$$\text{Env} = \{\text{Get} : \text{String} \twoheadrightarrow \text{String}\}$$

Environment — tail-resumptive handler

$$\begin{aligned}
&\text{env} : \text{List}\ (\text{String} \times \text{String}) \to \alpha!\text{Env} \Rightarrow \alpha \\
&\text{env}\ \mathit{xs} = \{\ \mathbf{return}\ \mathit{ans} \quad \mapsto \mathit{ans} \\
&\qquad\qquad\quad \langle\!\langle \text{Get}\ x \twoheadrightarrow r \rangle\!\rangle \mapsto r\ (\text{assoc}\ x\ \mathit{xs})\}
\end{aligned}$$

Dynamic multi user environments

Effect interface

$$\mathrm{Env} = \{\mathrm{Get} : \mathrm{String} \twoheadrightarrow \mathrm{String}\}$$

Environment — tail-resumptive handler

$$\begin{aligned}
&\mathrm{env} : \mathrm{List}\ (\mathrm{String} \times \mathrm{String}) \to \alpha!\mathrm{Env} \Rightarrow \alpha \\
&\mathrm{env}\ \mathit{xs} = \{\ \mathbf{return}\ \mathit{ans} \quad \mapsto \mathit{ans} \\
&\qquad\qquad\quad \langle\!\langle \mathrm{Get}\ x \twoheadrightarrow r \rangle\!\rangle \mapsto r\ (\mathrm{assoc}\ x\ \mathit{xs})\}
\end{aligned}$$

Effect interface

$$\mathrm{Session} = \{\mathrm{Su} : \mathrm{User} \twoheadrightarrow \mathrm{Void}\}$$

Dynamic multi user environments

Effect interface

$$\text{Env} = \{\text{Get} : \text{String} \twoheadrightarrow \text{String}\}$$

Environment — tail-resumptive handler

$$\begin{aligned}
&\text{env} : \text{List}\ (\text{String} \times \text{String}) \to \alpha!\text{Env} \Rightarrow \alpha\\
&\text{env}\ \mathit{xs} = \{\ \textbf{return}\ \mathit{ans} \mapsto \mathit{ans}\\
&\qquad\qquad \langle\!\langle \text{Get}\ x \twoheadrightarrow r \rangle\!\rangle \mapsto r\ (\text{assoc}\ x\ \mathit{xs})\}
\end{aligned}$$

Effect interface

$$\text{Session} = \{\text{Su} : \text{User} \twoheadrightarrow \text{Void}\}$$

Shell — shadowing handler

$$\begin{aligned}
&\text{shell} : \text{List}\ (\text{User} \times \text{List}\ (\text{String} \times \text{String})) \to (\alpha!\text{Env} \oplus \text{Session}) \Rightarrow \alpha\\
&\text{shell}\ \mathit{es} = \text{env}\ [\,] \circ \{\ \textbf{return}\ \mathit{ans} \mapsto \mathit{ans}\\
&\qquad\qquad \langle\!\langle \text{Su}\ \mathit{user} \twoheadrightarrow r \rangle\!\rangle \mapsto \text{env}\ (\text{assoc}\ \mathit{user}\ \mathit{es})(r\ \langle\rangle)\}
\end{aligned}$$

Dynamic multi user environments

Effect interface

$$\text{Env} = \{\text{Get} : \text{String} \twoheadrightarrow \text{String}\}$$

Environment — tail-resumptive handler

$$\begin{aligned}
&\text{env} : \text{List}\ (\text{String} \times \text{String}) \to \alpha!\text{Env} \Rightarrow \alpha \\
&\text{env}\ \mathit{xs} = \{\ \mathbf{return}\ \mathit{ans} \quad \mapsto \mathit{ans} \\
&\qquad\qquad\quad \langle\!\langle \text{Get}\ x \twoheadrightarrow r \rangle\!\rangle \mapsto r\ (\text{assoc}\ x\ \mathit{xs})\}
\end{aligned}$$

Effect interface

$$\text{Session} = \{\text{Su} : \text{User} \twoheadrightarrow \text{Void}\}$$

Shell — shadowing handler

$$\begin{aligned}
&\text{shell} : \text{List}\ (\text{User} \times \text{List}\ (\text{String} \times \text{String})) \to (\alpha!\text{Env} \oplus \text{Session}) \Rightarrow \alpha \\
&\text{shell}\ \mathit{es} = \text{env}\ [] \circ \{\ \mathbf{return}\ \mathit{ans} \qquad \mapsto \mathit{ans} \\
&\qquad\qquad\qquad\qquad \langle\!\langle \text{Su}\ \mathit{user} \twoheadrightarrow r \rangle\!\rangle \mapsto \text{env}\ (\text{assoc}\ \mathit{user}\ \mathit{es})(r\ \langle\rangle)\}
\end{aligned}$$

Example

$$\begin{aligned}
&\text{shell}\ \langle [\langle \text{Alice}, [\langle \texttt{"USER"}, \texttt{"Alice"} \rangle] \rangle, \langle \text{Bob}, \ldots \rangle], \\
&\qquad \lambda\langle\rangle.\langle \mathbf{do}\ \text{Su Alice}; \mathbf{do}\ \text{Get}\ \texttt{"USER"}, \mathbf{do}\ \text{Su Bob}; \mathbf{do}\ \text{Get}\ \texttt{"USER"} \rangle\rangle \\
&\rightsquigarrow \langle \texttt{"Alice"}, \texttt{"Bob"} \rangle
\end{aligned}$$

Process duplication via UNIX fork

$$\texttt{let}\ \mathit{pid} \leftarrow \mathrm{fork}\ \langle\rangle\ \texttt{in}$$
$$\texttt{if}\ \mathit{pid} = 0\ \texttt{then}\ \textit{child continuation}$$
$$\texttt{else}\ \textit{parent continuation}$$

Note, fork causes execution of both continuations, i.e. it returns twice!

Timesharing with UNIX fork

Effect interface

$$\text{Timeshare} = \{\text{Fork} : \text{Void} \to \text{Int}, \text{Interrupt} : \text{Void} \twoheadrightarrow \text{Void}, \text{Wait} : \text{Int} \twoheadrightarrow \text{Int}\}$$

Timesharing with UNIX fork

Effect interface

$$\text{Timeshare} = \{\text{Fork} : \text{Void} \to \text{Int}, \text{Interrupt} : \text{Void} \twoheadrightarrow \text{Void}, \text{Wait} : \text{Int} \twoheadrightarrow \text{Int}\}$$

Scheduler — multi-shot handler

$\text{tmshare} : \text{State} \to \alpha!\text{Timeshare} \Rightarrow \text{List}\ (\text{Int} \times \alpha)$
$\text{tmshare}\ \mathit{st} = \{$

$\}$

Timesharing with UNIX fork

Effect interface

$$\text{Timeshare} = \{\text{Fork} : \text{Void} \to \text{Int}, \text{Interrupt} : \text{Void} \twoheadrightarrow \text{Void}, \text{Wait} : \text{Int} \twoheadrightarrow \text{Int}\}$$

Scheduler — multi-shot handler

$$\begin{aligned}
&\text{tmshare} : \text{State} \to \alpha!\text{Timeshare} \Rightarrow \text{List}\ (\text{Int} \times \alpha)\\
&\text{tmshare}\ \mathit{st} = \{\\
&\qquad \langle\!\langle \text{Fork}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto \ \mathbf{let}\ \mathit{pid} \leftarrow \text{incr}\ \mathit{st}.\mathit{next_pid}\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad\quad \text{runNext}\ \langle \mathit{st}\ \mathbf{with}\ \mathit{rq} = \langle \mathit{st}.\mathit{cur}, \lambda\langle\rangle.r\ 0\rangle :: \langle \lambda\langle\rangle.r\ \mathit{pid}\rangle :: \mathit{st}.\mathit{rq}\rangle\\
&\}
\end{aligned}$$

Timesharing with UNIX fork

Effect interface

$$\mathsf{Timeshare} = \{\mathsf{Fork} : \mathsf{Void} \to \mathsf{Int}, \mathsf{Interrupt} : \mathsf{Void} \twoheadrightarrow \mathsf{Void}, \mathsf{Wait} : \mathsf{Int} \twoheadrightarrow \mathsf{Int}\}$$

Scheduler — multi-shot handler

$$\begin{array}{l}
\mathsf{tmshare} : \mathsf{State} \to \alpha!\mathsf{Timeshare} \Rightarrow \mathsf{List}\ (\mathsf{Int} \times \alpha) \\
\mathsf{tmshare}\ st = \{ \\
\qquad \langle\!\langle \mathsf{Fork}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \ \mapsto\ \mathbf{let}\ pid \leftarrow \mathsf{incr}\ st.next_pid\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{runNext}\ \langle st\ \mathbf{with}\ rq = \langle st.cur, \lambda\langle\rangle.r\ 0\rangle :: \langle \lambda\langle\rangle.r\ pid\rangle :: st.rq\rangle \\
\}
\end{array}$$

Timesharing with UNIX fork

Effect interface

$$\text{Timeshare} = \{\text{Fork} : \text{Void} \to \text{Int}, \text{Interrupt} : \text{Void} \twoheadrightarrow \text{Void}, \text{Wait} : \text{Int} \twoheadrightarrow \text{Int}\}$$

Scheduler — multi-shot handler

$$\begin{aligned}
&\text{tmshare} : \text{State} \to \alpha!\text{Timeshare} \Rightarrow \text{List}\ (\text{Int} \times \alpha)\\
&\text{tmshare}\ st = \{\\
&\quad \langle\!\langle \text{Fork}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto \mathbf{let}\ pid \leftarrow \text{incr}\ st.next_pid\ \mathbf{in}\\
&\qquad\qquad \text{runNext}\ \langle st\ \mathbf{with}\ rq = \langle st.cur, \lambda\langle\rangle.r\ 0\rangle :: \langle \lambda\langle\rangle.r\ pid\rangle :: st.rq\rangle\\
&\quad \langle\!\langle \text{Interrupt}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto \text{runNext}\ \langle st\ \mathbf{with}\ rq = \langle st.cur, \lambda\langle\rangle.r\ \langle\rangle\rangle :: st.rq\rangle\\
&\}
\end{aligned}$$

Timesharing with UNIX fork

Effect interface

$$\text{Timeshare} = \{\text{Fork} : \text{Void} \to \text{Int}, \text{Interrupt} : \text{Void} \twoheadrightarrow \text{Void}, \text{Wait} : \text{Int} \twoheadrightarrow \text{Int}\}$$

Scheduler — multi-shot handler

$$
\begin{aligned}
&\text{tmshare} : \text{State} \to \alpha!\text{Timeshare} \Rightarrow \text{List}\,(\text{Int} \times \alpha)\\
&\text{tmshare}\ st = \{\\
&\qquad \langle\!\langle \text{Fork}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto \mathbf{let}\ pid \leftarrow \text{incr}\ st.next\_pid\ \mathbf{in}\\
&\qquad\qquad \text{runNext}\ \langle st\ \mathbf{with}\ rq = \langle st.cur, \lambda\langle\rangle.r\ 0\rangle :: \langle \lambda\langle\rangle.r\ pid\rangle :: st.rq\rangle\\
&\qquad \langle\!\langle \text{Interrupt}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle \mapsto \text{runNext}\ \langle st\ \mathbf{with}\ rq = \langle st.cur, \lambda\langle\rangle.r\ \langle\rangle\rangle :: st.rq\rangle\\
&\qquad \langle\!\langle \text{Wait}\ pid \twoheadrightarrow r \rangle\!\rangle \mapsto \text{runNext}\ \langle st\ \mathbf{with}\ bq = \langle pid, \langle st.cur, \lambda\langle\rangle.r\ \langle\rangle\rangle\rangle :: st.bq\rangle\\
&\}
\end{aligned}
$$

Timesharing with UNIX fork

Effect interface

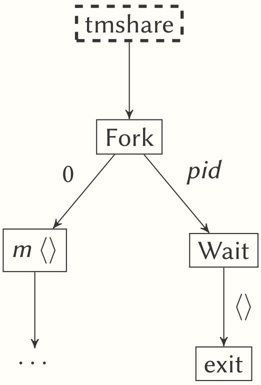
$$\mathsf{Timeshare} = \{\mathsf{Fork} : \mathsf{Void} \to \mathsf{Int}, \mathsf{Interrupt} : \mathsf{Void} \twoheadrightarrow \mathsf{Void}, \mathsf{Wait} : \mathsf{Int} \twoheadrightarrow \mathsf{Int}\}$$

Scheduler — multi-shot handler

$$\begin{array}{l}
\mathsf{tmshare} : \mathsf{State} \to \alpha!\mathsf{Timeshare} \Rightarrow \mathsf{List}\ (\mathsf{Int} \times \alpha) \\
\mathsf{tmshare}\ \mathit{st} = \{ \\
\qquad \begin{array}{rcl}
\langle\!\langle \mathsf{Fork}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle & \mapsto & \mathbf{let}\ \mathit{pid} \leftarrow \mathsf{incr}\ \mathit{st.next_pid}\ \mathbf{in} \\
& & \mathsf{runNext}\ \langle \mathit{st}\ \mathbf{with}\ \mathit{rq} = \langle \mathit{st.cur}, \lambda\langle\rangle.r\ 0\rangle :: \langle \lambda\langle\rangle.r\ \mathit{pid}\rangle :: \mathit{st.rq}\rangle \\
\langle\!\langle \mathsf{Interrupt}\ \langle\rangle \twoheadrightarrow r \rangle\!\rangle & \mapsto & \mathsf{runNext}\ \langle \mathit{st}\ \mathbf{with}\ \mathit{rq} = \langle \mathit{st.cur}, \lambda\langle\rangle.r\ \langle\rangle\rangle :: \mathit{st.rq}\rangle \\
\langle\!\langle \mathsf{Wait}\ \mathit{pid} \twoheadrightarrow r \rangle\!\rangle & \mapsto & \mathsf{runNext}\ \langle \mathit{st}\ \mathbf{with}\ \mathit{bq} = \langle \mathit{pid}, \langle \mathit{st.cur}, \lambda\langle\rangle.r\ \langle\rangle\rangle\rangle :: \mathit{st.bq}\rangle \\
\mathbf{return}\ \mathit{ans} & \mapsto & \mathbf{let}\ \langle \mathit{rq}', \mathit{bq}'\rangle = \mathsf{pop}\ \mathit{st.cur}\ \mathit{st.bq}\ \mathbf{in} \\
& & \mathsf{runNext}\ \langle \mathit{st}\ \mathbf{with}\ \mathit{rq} = \mathit{st.rq} +\!\!+\ \mathit{rq}', \mathit{bq} = \mathit{bq}', \mathit{done} = \langle \mathit{st.cur}, \mathit{ans}\rangle :: \mathit{st.done}\rangle\}
\end{array}
\end{array}$$

Semantics for init

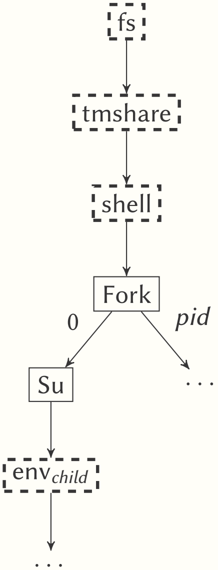
$$\text{init} : (\text{Void} \to \alpha!\text{Timeshare}) \to \alpha$$
$$\begin{aligned}\text{init}\ m = {} & \textbf{let}\ \mathit{pid} \leftarrow \textbf{do}\ \text{Fork}\ \langle\rangle\ \textbf{in} \\ & \textbf{if}\ \mathit{pid} = 0\ \textbf{then}\ m\ \langle\rangle \\ & \textbf{else}\ \textbf{do}\ \text{Wait}\ \mathit{pid}; \text{exit}\ 0\end{aligned}$$



Composition confers functionality

Functionality through composition

- Suppose we have a file system handler fs
- Then (fs ∘ tmshare ∘ env) yields a basic operating system!
- Every process share the same filesystem…



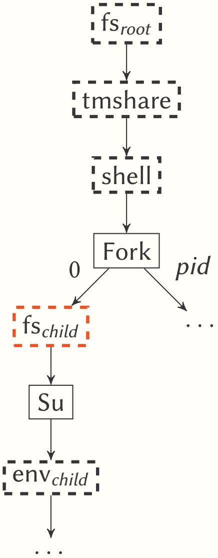
Composition confers functionality

Functionality through composition

- Suppose we have a file system handler fs
- Then (fs ∘ tmshare ∘ env) yields a basic operating system!
- Every process share the same filesystem...

Sandboxing through composition

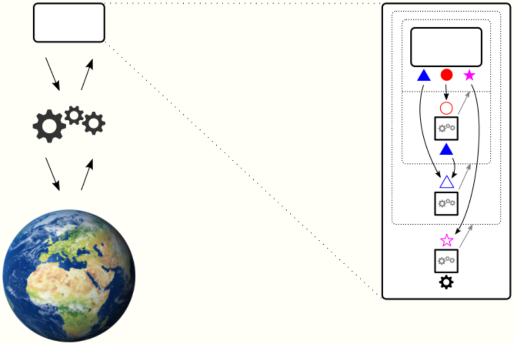
- Suppose we want each process to have its own filesystem?
- Easy: fs ∘ tmshare ∘ env ∘ fs



The future

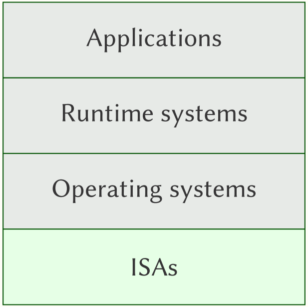
(figure courtesy of Sam Lindley)

The future



(figure courtesy of Sam Lindley)

The Software Stack: ISAs



WebAssembly: neither web nor assembly (Haas et al. 2017)

What is Wasm?

- A virtual instruction set architecture
- An abstraction of the hardware
- Secure, sandboxed execution environment

Code format

- A Wasm “program” is a structured module
- Designed for streaming compilation
- The term language is **statically typed** and block-structured
- Control flow is **structured** (i.e. all CFGs are reducible)

WebAssembly: neither web nor assembly (Haas et al. 2017)

What is Wasm?

- A virtual instruction set architecture
- An abstraction of the hardware
- Secure, sandboxed execution environment

Code format

- A Wasm "program" is a structured module
- Designed for streaming compilation
- The term language is **statically typed** and block-structured
- Control flow is **structured** (i.e. all CFGs are reducible)

Problem

How do I compile my control idioms to Wasm?

WebAssembly: neither web nor assembly (Haas et al. 2017)

What is Wasm?

- A virtual instruction set architecture
- An abstraction of the hardware
- Secure, sandboxed execution environment

Code format

- A Wasm “program” is a structured module
- Designed for streaming compilation
- The term language is **statically typed** and block-structured
- Control flow is **structured** (i.e. all CFGs are reducible)

Problem

How do I compile my control idioms to Wasm?

Solution

A **global restructuring scheme** for programs (e.g. CPS, Asyncify)

Asyncify is the current state-of-the-art (1)

```
(func $doSomething (param $arg i32) (result i32)
  (call $foo
    (call $bar (local.get $arg))))
```

Asyncify is the current state-of-the-art (2)

```
(func $doSomething (param $arg i32) (result i32)
  (call $foo
    (call $bar (local.get $arg))))
```

can `$bar` suspend?

Asyncify is the current state-of-the-art (2)

can `$foo` suspend?

```
(func $doSomething (param $arg i32) (result i32)
  (call $foo
    (call $bar (local.get $arg))))
```

can `$bar` suspend?

Asyncify is the current state-of-the-art (2)

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))                ;; test rewind state
    (then (local.set $arg                                               ;; store local $arg
            (i32.load offset=4 (global.get $asyncify_heap_ptr)))
          (local.set $call_idx                                          ;; continuation point
            (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
                (then (br $call_bar))                                    ;; restore $call_bar
                (else (br $restore_foo))))
          (else (br $call_bar))))                                        ;; regular $call_bar
      (local.set $ret (call $bar (local.get 0)))
      (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32)    ;; test unwind state
            (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
                (i32.store offset=8 (global.get $asyncify_heap_ptr (i32.const 0))
                (return (i32.const 0)))  ...))))))
```

Asyncify is the current state-of-the-art (2)

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))                 ;; test rewind state
    (then (local.set $arg                                                 ;; store local $arg
            (i32.load offset=4 (global.get $asyncify_heap_ptr)))
          (local.set $call_idx                                            ;; continuation point
            (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
                  (then (br $call_bar))                                    ;; restore $call_bar
                  (else (br $restore_foo))))
          (else (br $call_bar))))                                          ;; regular $call_bar
      (local.set $ret (call $bar (local.get 0)))
      (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32)    ;; test unwind state
            (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
                  (i32.store offset=8 (global.get $asyncify_heap_ptr (i32.const 0))
                  (return (i32.const 0)))  ...))))))
```

Asyncify is the current state-of-the-art (2)

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))                          ;; test rewind state
    (then (local.set $arg                                                         ;; store local $arg
            (i32.load offset=4 (global.get $asyncify_heap_ptr)))
          (local.set $call_idx                                                    ;; continuation point
            (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
                  (then (br $call_bar))                                            ;; restore $call_bar
                  (else (br $restore_foo))))
          (else (br $call_bar))))                                                  ;; regular $call_bar
      (local.set $ret (call $bar (local.get 0)))
      (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32)    ;; test unwind state
            (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
                  (i32.store offset=8 (global.get $asyncify_heap_ptr (i32.const 0))
                  (return (i32.const 0)))  ...))))))
```

Asyncify is the current state-of-the-art (2)

```
(func $doSomething (param $arg i32) (result i32)
  (local $call_idx i32)
  (local $ret i32)
  (if (i32.eq (global.get $asyncify_mode) (i32.const 2))                   ;; test rewind state
    (then (local.set $arg                                                   ;; store local $arg
            (i32.load offset=4 (global.get $asyncify_heap_ptr)))
          (local.set $call_idx                                              ;; continuation point
            (i32.load offset=8 (global.get $asyncify_heap_ptr)))
    (else))
  (block $call_foo (result i32)
    (block $restore_foo (result i32)
      (block $call_bar (result i32)
        (local.get $arg)
        (if (i32.eq (global.get $asyncify_mode) (i32.const 2)) (result i32)
          (then (if (i32.eq (local.get $call_idx) (i32.const 0))
                  (then (br $call_bar))                                      ;; restore $call_bar
                  (else (br $restore_foo))))
          (else (br $call_bar))))                                            ;; regular $call_bar
      (local.set $ret (call $bar (local.get 0)))
      (if (i32.eq (global.get $asyncify_mode) (i32.const 1)) (result i32)    ;; test unwind state
            (then (i32.store offset=4 (global.get $asyncify_heap_ptr) (local.get $arg))
                  (i32.store offset=8 (global.get $asyncify_heap_ptr (i32.const 0))
                  (return (i32.const 0)))  ...))))))
```

A basis for stack switching

Can we do better?

A basis for stack switching

Can we do better?

- Yes! Effect handlers!
- Intuition: a continuation is a handle to a stack
- Aligns with the design restrictions of Wasm

The WasmFX instruction set extension (1)

(joint work with Arjun Guha, Andreas Rossberg, Daan Leijen, Frank Emrich, KC Sivaramakrishnan, Luna Phipps-Costin, Matija Pretnar, and Sam Lindley)

Types

- **`cont`** $\$ft$

Tags

- **`tag`** $\$tag : [\sigma^*] \to [\tau^*]$

Core instructions

- **`cont.new`** $\$ct : [(\textbf{ref}\ \$ft)] \to [(\textbf{ref}\ \$ct)]$
- **`suspend`** $\$tag : [\sigma^*] \to [\tau^*]$
- **`resume`** $\$ct\ (\textbf{tag}\ \$tag\ \$h) : [\sigma^* (\textbf{ref}\ \$ct)] \to [\tau^*]$

We call this extension **WasmFX**

The WasmFX instruction set extension (1)

(joint work with Arjun Guha, Andreas Rossberg, Daan Leijen, Frank Emrich, KC Sivaramakrishnan, Luna Phipps-Costin, Matija Pretnar, and Sam Lindley)

Types

- **`cont`** $\$ft$

  where $\$ft : [\sigma^*] \to [\tau^*]$

Tags

- **`tag`** $\$tag : [\sigma^*] \to [\tau^*]$

Core instructions

- **`cont.new`** $\$ct : [(\texttt{ref}\ \$ft)] \to [(\texttt{ref}\ \$ct)]$
- **`suspend`** $\$tag : [\sigma^*] \to [\tau^*]$
- **`resume`** $\$ct\ (\texttt{tag}\ \$tag\ \$h) : [\sigma^*(\texttt{ref}\ \$ct)] \to [\tau^*]$

We call this extension **WasmFX**

The WasmFX instruction set extension (1)

(joint work with Arjun Guha, Andreas Rossberg, Daan Leijen, Frank Emrich, KC Sivaramakrishnan, Luna Phipps-Costin, Matija Pretnar, and Sam Lindley)

Types

- **`cont`** $\$ft$

Tags

- **`tag`** $\$tag : [\sigma^*] \to [\tau^*]$

Core instructions

- **`cont.new`** $\$ct : [(\texttt{ref}\ \$ft)] \to [(\texttt{ref}\ \$ct)]$
- **`suspend`** $\$tag : [\sigma^*] \to [\tau^*]$
- **`resume`** $\$ct\ (\texttt{tag}\ \$tag\ \$h) : [\sigma^*(\texttt{ref}\ \$ct)] \to [\tau^*]$

We call this extension **WasmFX**

The WasmFX instruction set extension (1)

(joint work with Arjun Guha, Andreas Rossberg, Daan Leijen, Frank Emrich, KC Sivaramakrishnan, Luna Phipps-Costin, Matija Pretnar, and Sam Lindley)

Types

- $\texttt{cont}\ \$\mathit{ft}$

Tags

- $\texttt{tag}\ \$\mathit{tag} : [\sigma^*] \to [\tau^*]$

Core instructions

- $\texttt{cont.new}\ \$\mathit{ct} : [(\texttt{ref}\ \$\mathit{ft})] \to [(\texttt{ref}\ \$\mathit{ct})]$

  $$\begin{array}{rl} \text{where} & \$\mathit{ft} : [\sigma^*] \to [\tau^*] \\ \text{and} & \$\mathit{ct} : \texttt{cont}\ \$\mathit{ft} \end{array}$$

- $\texttt{suspend}\ \$\mathit{tag} : [\sigma^*] \to [\tau^*]$
- $\texttt{resume}\ \$\mathit{ct}\ (\texttt{tag}\ \$\mathit{tag}\ \$h) : [\sigma^*(\texttt{ref}\ \$\mathit{ct})] \to [\tau^*]$

We call this extension **WasmFX**

The WasmFX instruction set extension (1)

(joint work with Arjun Guha, Andreas Rossberg, Daan Leijen, Frank Emrich, KC Sivaramakrishnan, Luna Phipps-Costin, Matija Pretnar, and Sam Lindley)

Types

- $\texttt{cont}\ \$\mathit{ft}$

Tags

- $\texttt{tag}\ \$\mathit{tag} : [\sigma^*] \to [\tau^*]$

Core instructions

- $\texttt{cont.new}\ \$\mathit{ct} : [(\texttt{ref}\ \$\mathit{ft})] \to [(\texttt{ref}\ \$\mathit{ct})]$
- $\texttt{suspend}\ \$\mathit{tag} : [\sigma^*] \to [\tau^*]$

  where $\$\mathit{tag} : [\sigma^*] \to [\tau^*]$

- $\texttt{resume}\ \$\mathit{ct}\ (\texttt{tag}\ \$\mathit{tag}\ \$h) : [\sigma^*(\texttt{ref}\ \$\mathit{ct})] \to [\tau^*]$

We call this extension **WasmFX**

https://wasmfx.dev

The WasmFX instruction set extension (1)

(joint work with Arjun Guha, Andreas Rossberg, Daan Leijen, Frank Emrich, KC Sivaramakrishnan, Luna Phipps-Costin, Matija Pretnar, and Sam Lindley)

Types

- $\texttt{cont}\ \$\mathit{ft}$

Tags

- $\texttt{tag}\ \$\mathit{tag} : [\sigma^*] \to [\tau^*]$

Core instructions

- $\texttt{cont.new}\ \$\mathit{ct} : [(\texttt{ref}\ \$\mathit{ft})] \to [(\texttt{ref}\ \$\mathit{ct})]$
- $\texttt{suspend}\ \$\mathit{tag} : [\sigma^*] \to [\tau^*]$
- $\texttt{resume}\ \$\mathit{ct}\ (\texttt{tag}\ \$\mathit{tag}\ \$h) : [\sigma^*(\texttt{ref}\ \$\mathit{ct})] \to [\tau^*]$

$$\begin{array}{l} \text{where } \{\$\mathit{tag}_i : [\sigma_i^*] \to [\tau_i^*] \text{ and } \$h_i : [\sigma_i^*\ (\texttt{ref null}\ \$\mathit{ct}_i)] \text{ and} \\ \qquad \$\mathit{ct}_i : \texttt{cont}\ \$\mathit{ft}_i \text{ and } \$\mathit{ft}_i : [\tau_i^*] \to [\tau^*]\}_i \\ \quad \text{and } \$\mathit{ct} : \texttt{cont}\ \$\mathit{ft} \\ \quad \text{and } \$\mathit{ft} : [\sigma^*] \to [\tau^*] \end{array}$$

We call this extension **WasmFX**

Example: Yield-style generators

```
(tag $gen (param i32))

(func $ints
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $ints
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sumUp (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k)
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $ints
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sumUp (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k)
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $ints
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sumUp (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k))
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $ints
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sumUp (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k)
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

Example: Yield-style generators

```
(tag $gen (param i32))

(func $ints
  (local $i i32) ;; zero-initialised local
  (loop $produce-next
    (suspend $gen (local.get $i))
    (local.set $i
      (i32.add (local.get $i)
               (i32.const 1)))
    (br $produce-next) ;; continue next
  )
)
```

```
(func $sumUp (param $upto i32) (param $k (cont [] -> []))
  (local $n i32) ;; current value
  (local $s i32) ;; accumulator
  (loop $consume-next
    (block $on_gen (result i32 (cont [] -> []))
      (resume (tag $gen $on_gen) (local.get $k)
      (call $print (local.get $s))
    ) ;; stack: [i32 (cont [] -> [])]
    (local.set $k) ;; save next continuation
    (local.set $n) ;; save current value
    (local.set $s (i32.add (local.get $s)
                           (local.get $n)))
    (br_if $consume-next
      (i32.lt_u (local.get $n) (local.get $upto)))
  )
  (call $print ((local.get $s)))
)
```

`(call $sumUp (i32.const 10) (cont.new (ref.func $ints)))` returns 55

The WasmFX instruction set extension (2)

(joint work with Arjun Guha, Andreas Rossberg, Daan Leijen, Frank Emrich, KC Sivaramakrishnan, Luna Phipps-Costin, Matija Pretnar, and Sam Lindley)

Types

- **`cont`** $\$\mathit{ft}$

Tags

- **`tag`** $\$\mathit{tag} : [\sigma^*] \to [\tau^*]$

Core instructions

- **`cont.new`** $\$\mathit{ct} : [(\mathbf{ref}\ \$\mathit{ft})] \to [(\mathbf{ref}\ \$\mathit{ct})]$
- **`suspend`** $\$\mathit{tag} : [\sigma^*] \to [\tau^*]$
- **`resume`** $\$\mathit{ct}\ (\mathbf{tag}\ \$\mathit{tag}\ \$h) : [\sigma^*(\mathbf{ref}\ \$\mathit{ct})] \to [\tau^*]$

We call this extension **WasmFX**

The WasmFX instruction set extension (2)

(joint work with Arjun Guha, Andreas Rossberg, Daan Leijen, Frank Emrich, KC Sivaramakrishnan, Luna Phipps-Costin, Matija Pretnar, and Sam Lindley)

Types

- **`cont`** $\$ft$

Tags

- **`tag`** $\$tag : [\sigma^*] \to [\tau^*]$

Core instructions

- **`cont.new`** $\$ct : [(\textbf{ref}\ \$ft)] \to [(\textbf{ref}\ \$ct)]$
- **`suspend`** $\$tag : [\sigma^*] \to [\tau^*]$
- **`resume`** $\$ct\ (\textbf{tag}\ \$tag\ \$h) : [\sigma^*(\textbf{ref}\ \$ct)] \to [\tau^*]$

Legend

- Spec'ed
- Reference impl.
- Wasmtime impl.

We call this extension **WasmFX**

Effect handlers for WebAssembly

Key properties provided by handlers

- A structured facility for non-local control flow
- Just works with builtin effects (e.g. exceptions, threads)
- Seamless interoperability with host
- Compatible with standard debuggers and profilers

Conclusions and future work

Summary

- Effect handlers provide a universal abstraction for non-local control
- Flexible and customisable control flow
- Compositional virtualisation of system effects

Future work

- More prominent applications!
- Implementation strategies
- Hardware support for effect handlers?

References

Plotkin, Gordon D. and Matija Pretnar (2013). "Handling Algebraic Effects". In: *Logical Methods in Computer Science* 9.4. DOI: 10.2168/LMCS-9(4:23)2013.

Haas, Andreas et al. (2017). "Bringing the web up to speed with WebAssembly". In: *PLDI*. ACM, pp. 185–200.

Hillerström, Daniel (2021). "Foundations for Programming and Implementing Effect Handlers". PhD thesis. The University of Edinburgh, Scotland, UK.

Thomson, Patrick et al. (2022). "Fusing industry and academia at GitHub (experience report)". In: *Proc. ACM Program. Lang.* 6.ICFP, pp. 496–511. DOI: 10.1145/3547639. URL: https://doi.org/10.1145/3547639.

Phipps-Costin, Luna et al. (2023). "Continuing WebAssembly with Effect Handlers". In: *Proc. ACM Program. Lang.* 7.OOPSLA2, pp. 460–485.