Reconstructing Continuation-Passing Semantics for WebAssembly

Guannan Wei [1, 2] Alexander Bai [1] Dinghong Zhong [3] Jiatai Zhang [1]

WebAssembly Workshop @ POPL 2025
Jan 20 2025, Denver, CO

[1]Tufts University, [2]INRIA/ENS-PSL, [3]Unaffliated

WebAssembly (Wasm)

- A stack-based, low-level, fast IR for the web, now supported in major browsers
- Official formalized semantics
    - Small-step reduction dynamic semantics
    - Static type system that constrains the shape of the stack
    - Soundness and safety

WebAssembly (Wasm)

- A stack-based, low-level, fast IR for the web, now supported in major browsers
- Official formalized semantics
  - Small-step reduction dynamic semantics
  - Static type system that constrains the shape of the stack
  - Soundness and safety
- Many work-in-progress new features, e.g., effect handlers (WasmFX)

Wasm's Reference Small-step Semantics

- Reduction: $s; v^*; e^* \rightarrow s; v^*; e^*$

Wasm's Reference Small-step Semantics

- Reduction: $s; v^*; e^* \to s; v^*; e^*$
- Explicit and verbose "adminstrative instructions" to represent evaluation context

<pre>loop
  i32.const 4
  i32.const 2
  i32.const 1
  i32.add
  i32.add
  br 0
end</pre>
	$\rightsquigarrow$	<pre>label{...}
		  i32.const 4
		  i32.const 3
		  i32.add
		  br 0
		end</pre>
			$\rightsquigarrow$	<pre>label{...}
				  i32.const 7
				  br 0
				end</pre>
					$\rightsquigarrow$	<pre>loop
						  i32.const 4
						  i32.const 2
						  i32.const 1
						  i32.add
						  i32.add
						  br 0
						end</pre>

Wasm's Reference Small-step Semantics

- Reduction: $s; v^*; e^* \to s; v^*; e^*$
- Explicit and verbose "adminstrative instructions" to represent evaluation context

```
loop
  i32.const 4
  i32.const 2
  i32.const 1
  i32.add
  i32.add
  br 0
end
```

⇝

```
label{...}
  i32.const 4
  i32.const 3
  i32.add
  br 0
end
```

⇝

```
label{...}
  i32.const 7
  br 0
end
```

⇝

```
loop
  i32.const 4
  i32.const 2
  i32.const 1
  i32.add
  i32.add
  br 0
end
```

- Duplication and searching on the stack causes inefficiencies in deeply nested frames/labels

Wasm's Reference Small-step Semantics

- Reduction: $s; v^*; e^* \to s; v^*; e^*$
- Explicit and verbose "adminstrative instructions" to represent evaluation context

```
loop                                                      loop
  i32.const 4                                               i32.const 4
  i32.const 2      label{...}                               i32.const 2
  i32.const 1        i32.const 4                            i32.const 1
  i32.add     ⇝      i32.const 3   ⇝  label{...}       ⇝    i32.add
  i32.add            i32.add            i32.const 7         i32.add
  br 0               br 0               br 0                br 0
end                end                end                 end
```

- Duplication and searching on the stack causes inefficiencies in deeply nested frames/labels
- The reduction semantics is **not compositional** and **not tail-recursive**

This Work

- An alternative to reduction semantics of Wasm:
  - Rather than the first-order representation for control structures, we use CPS in the meta-language to represent control semantics

This Work

- An alternative to reduction semantics of Wasm:
  - Rather than the first-order representation for control structures, we use CPS in the meta-language to represent control semantics
- A compositional and tail recursive semantics for core Wasm in CPS
  - Implemented as a big-step interpreter
  - Or, can be viewed as a CPS transformer

Syntax of μWasm

$$
\begin{aligned}
\ell \in \mathsf{Label} &= \mathbb{N} \\
x \in \mathsf{Identifier} &= \mathbb{N} \\
t \in \mathsf{ValueType} &::= \mathsf{i32} \mid \mathsf{i64} \mid \ldots \\
\mathit{ft} \in \mathsf{FunctionType} &::= t^* \to t^* \\
e \in \mathsf{Instruction} &::= \mathsf{nop} \mid t.\mathsf{const}\ c \mid t.\{\mathsf{add}, \mathsf{sub}, \mathsf{eq}, \ldots\} \\
&\quad \mid \mathsf{local.get}\ x \mid \mathsf{local.set}\ x \\
&\quad \mid \mathsf{block}\ \mathit{ft}\ \mathit{es} \mid \mathsf{loop}\ \mathit{ft}\ \mathit{es} \mid \mathsf{if}\ \mathit{ft}\ \mathit{es}\ \mathit{es} \\
&\quad \mid \mathsf{br}\ \ell \mid \mathsf{call}\ x \mid \mathsf{return} \\
\mathit{es} \in \mathsf{Instructions} &= \mathsf{List}[\mathsf{Instruction}] \\
f \in \mathsf{Function} &::= \mathsf{func}\ x\ \{\mathsf{type} : \mathit{ft}, \mathsf{locals} : t^*, \mathsf{body} : \mathit{es}\} \\
m \in \mathsf{Module} &::= \mathsf{module}\ f^*
\end{aligned}
$$

Syntax of μWasm

$$
\begin{aligned}
\ell \in \text{Label} &= \mathbb{N} \\
x \in \text{Identifier} &= \mathbb{N} \\
t \in \text{ValueType} &::= \text{i32} \mid \text{i64} \mid \ldots \\
ft \in \text{FunctionType} &::= t^* \rightarrow t^* \\
e \in \text{Instruction} &::= \text{nop} \mid t.\text{const } c \mid t.\{\text{add}, \text{sub}, \text{eq}, \ldots\} \\
&\mid \text{local.get } x \mid \text{local.set } x \\
&\mid \text{block } ft \ es \mid \text{loop } ft \ es \mid \text{if } ft \ es \ es \\
&\mid \text{br } \ell \mid \text{call } x \mid \text{return} \\
es \in \text{Instructions} &= \text{List[Instruction]} \\
f \in \text{Function} &::= \text{func } x \ \{\text{type} : ft, \text{locals} : t^*, \text{body} : es\} \\
m \in \text{Module} &::= \text{module } f^*
\end{aligned}
$$

Syntax of μWasm

$$
\begin{aligned}
\ell \in \text{Label} &= \mathbb{N} \\
x \in \text{Identifier} &= \mathbb{N} \\
t \in \text{ValueType} &::= \text{i32} \mid \text{i64} \mid \ldots \\
\mathit{ft} \in \text{FunctionType} &::= t^* \rightarrow t^* \\
e \in \text{Instruction} &::= \text{nop} \mid t.\text{const } c \mid t.\{\text{add}, \text{sub}, \text{eq}, \ldots\} \\
&\mid \text{local.get } x \mid \text{local.set } x \\
&\mid \text{block } \mathit{ft}\ \mathit{es} \mid \text{loop } \mathit{ft}\ \mathit{es} \mid \text{if } \mathit{ft}\ \mathit{es}\ \mathit{es} \\
&\mid \text{br } \ell \mid \text{call } x \mid \text{return} \\
\mathit{es} \in \text{Instructions} &= \text{List[Instruction]} \\
f \in \text{Function} &::= \text{func } x\ \{\text{type} : \mathit{ft}, \text{locals} : t^*, \text{body} : \mathit{es}\} \\
m \in \text{Module} &::= \text{module } f^*
\end{aligned}
$$

Syntax of μWasm

$$
\begin{aligned}
\ell \in \text{Label} &= \mathbb{N} \\
x \in \text{Identifier} &= \mathbb{N} \\
t \in \text{ValueType} &::= \text{i32} \mid \text{i64} \mid \ldots \\
\mathit{ft} \in \text{FunctionType} &::= t^* \to t^* \\
e \in \text{Instruction} &::= \text{nop} \mid t.\text{const}\ c \mid t.\{\text{add}, \text{sub}, \text{eq}, \ldots\} \\
&\quad \mid \text{local.get}\ x \mid \text{local.set}\ x \\
&\quad \mid \text{block}\ \mathit{ft}\ \mathit{es} \mid \text{loop}\ \mathit{ft}\ \mathit{es} \mid \text{if}\ \mathit{ft}\ \mathit{es}\ \mathit{es} \\
&\quad \mid \text{br}\ \ell \mid \text{call}\ x \mid \text{return} \\
\mathit{es} \in \text{Instructions} &= \text{List[Instruction]} \\
f \in \text{Function} &::= \text{func}\ x\ \{\text{type} : \mathit{ft}, \text{locals} : t^*, \text{body} : \mathit{es}\} \\
m \in \text{Module} &::= \text{module}\ f^*
\end{aligned}
$$

Syntax of μWasm

$$
\begin{array}{rcl}
\ell \in \text{Label} & = & \mathbb{N} \\
x \in \text{Identifier} & = & \mathbb{N} \\
t \in \text{ValueType} & ::= & \text{i32} \mid \text{i64} \mid \dots \\
\mathit{ft} \in \text{FunctionType} & ::= & t^* \to t^* \\
e \in \text{Instruction} & ::= & \text{nop} \mid t.\text{const}\ c \mid t.\{\text{add}, \text{sub}, \text{eq}, \dots\} \\
& & \mid \text{local.get}\ x \mid \text{local.set}\ x \\
& & \mid \text{block}\ \mathit{ft}\ \mathit{es} \mid \text{loop}\ \mathit{ft}\ \mathit{es} \mid \text{if}\ \mathit{ft}\ \mathit{es}\ \mathit{es} \\
& & \mid \text{br}\ \ell \mid \text{call}\ x \mid \text{return} \\
\mathit{es} \in \text{Instructions} & = & \text{List[Instruction]} \\
f \in \text{Function} & ::= & \text{func}\ x\ \{\text{type} : \mathit{ft}, \text{locals} : t^*, \text{body} : \mathit{es}\} \\
m \in \text{Module} & ::= & \text{module}\ f^*
\end{array}
$$

Semantics Definition

Evaluation function: $[\![\cdot]\!] : \text{List[Inst]} \to (\text{Stack} \times \text{Env} \times \text{Cont}) \to \text{Ans}$

$$
\begin{aligned}
v &\in \text{Value} = \mathbb{Z} \\
\sigma &\in \text{Stack} = \text{List[Value]} \\
\rho &\in \text{Env} \quad = \text{List[Value]} \\
\kappa &\in \text{Cont} \ \ = \text{Stack} \times \text{Env} \to \text{Ans}
\end{aligned}
$$

Semantics Definition

Evaluation function: $[\![\cdot]\!] : \mathsf{List[Inst]} \to (\mathsf{Stack} \times \mathsf{Env} \times \mathsf{Cont} \times \mathsf{Trail}) \to \mathsf{Ans}$

$$
\begin{aligned}
v &\in \mathsf{Value} = \mathbb{Z} \\
\sigma &\in \mathsf{Stack} = \mathsf{List[Value]} \\
\rho &\in \mathsf{Env} = \mathsf{List[Value]} \\
\kappa &\in \mathsf{Cont} = \mathsf{Stack} \times \mathsf{Env} \to \mathsf{Ans} \\
\theta &\in \mathsf{Trail} = \mathsf{List[Cont]}
\end{aligned}
$$

The CPS Semantics – Empty List of Inst

Evaluation function: $[\![\cdot]\!] : \text{List[Inst]} \to (\text{Stack} \times \text{Env} \times \text{Cont} \times \text{Trail}) \to \text{Ans}$

$$[\![\mathit{nil}]\!](\sigma, \rho, \kappa, \theta) = \kappa(\sigma, \rho)$$

The CPS Semantics – Stack Manipulation

Evaluation function: $[\![\cdot]\!] : \text{List[Inst]} \to (\text{Stack} \times \text{Env} \times \text{Cont} \times \text{Trail}) \to \text{Ans}$

$$
\begin{aligned}
[\![\text{nop} :: \textit{rest}]\!](\sigma, \rho, \kappa, \theta) &= [\![\textit{rest}]\!](\sigma, \rho, \kappa, \theta) \\
[\![t.\text{const } c :: \textit{rest}]\!](\sigma, \rho, \kappa, \theta) &= [\![\textit{rest}]\!](c :: \sigma, \rho, \kappa, \theta) \\
[\![t.\text{add} :: \textit{rest}]\!](v_1 :: v_2 :: \sigma, \rho, \kappa, \theta) &= [\![\textit{rest}]\!](v_1 + v_2 :: \sigma, \rho, \kappa, \theta)
\end{aligned}
$$

The CPS Semantics – Local Registers

Evaluation function: $[\![\cdot]\!] : \mathsf{List[Inst]} \to (\mathsf{Stack} \times \mathsf{Env} \times \mathsf{Cont} \times \mathsf{Trail}) \to \mathsf{Ans}$

$$
\begin{aligned}
[\![\mathsf{local.get}\ x :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](\rho(x) :: \sigma, \rho, \kappa, \theta) \\
[\![\mathsf{local.set}\ x :: \mathit{rest}]\!](v :: \sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](\sigma, \rho[x \mapsto v], \kappa, \theta)
\end{aligned}
$$

The CPS Semantics – Block and Branch

Evaluation function: $[\![\cdot]\!] : \mathsf{List[Inst]} \to (\mathsf{Stack} \times \mathsf{Env} \times \mathsf{Cont} \times \mathsf{Trail}) \to \mathsf{Ans}$

$$
\begin{aligned}
[\![\mathsf{block}\ (t^m \to t^n)\ \mathit{es} :: \mathit{rest}]\!](\sigma_{\mathit{arg}\ m} +\!\!+\ \sigma, \rho, \kappa, \theta) &= ???\\
[\![\mathsf{br}\ \ell :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) &= ???
\end{aligned}
$$

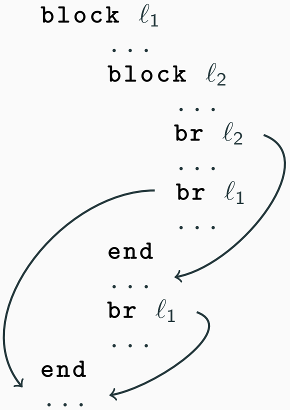
Wasm Control Flow - Blocks

- Blocks are structured and can be nested
- A block has a label (either named or nameless as de Bruijn indices)

$$
\begin{array}{l}
\texttt{block}\ \ell_1 \\
\quad \ldots \\
\quad \texttt{block}\ \ell_2 \\
\quad\quad \ldots \\
\quad\quad \texttt{br}\ \ell_2 \\
\quad\quad \ldots \\
\quad\quad \texttt{br}\ \ell_1 \\
\quad\quad \ldots \\
\quad \texttt{end} \\
\quad \ldots \\
\quad \texttt{br}\ \ell_1 \\
\quad \ldots \\
\texttt{end} \\
\ldots
\end{array}
$$

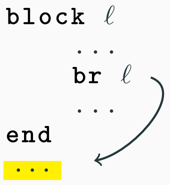
Wasm Control Flow - Blocks

- Blocks are structured and can be nested
- A block has a label (either named or nameless as de Bruijn indices)
- The label serves as a branch target, jumping to the instruction after the block
- **Idea**: we need to remember the "escaping continuation" of every block introduced in the scope



The CPS Semantics – Block and Branch

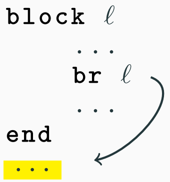
$$\llbracket \text{block } (t^m \to t^n)\ es :: rest \rrbracket(\sigma_{arg\ m} +\!\!+\ \sigma, \rho, \kappa, \theta) =$$
$$\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).\llbracket rest \rrbracket(\lfloor \sigma_1 \rfloor_n +\!\!+\ \sigma, \rho_1, \kappa, \theta) \text{ in}$$
$$\quad \llbracket es \rrbracket(\sigma_{arg}, \rho, \kappa_1, \kappa_1 :: \theta)$$
$$\llbracket \text{br } \ell :: rest \rrbracket(\sigma, \rho, \kappa, \theta) =$$
$$\quad \theta(\ell)(\sigma, \rho)$$



- The new continuation κ_1 is shared as ordinary continuation and escape/branch continuation
- ℓ is the de Bruijn index of the target label of the block, so $\theta(\ell)$ is the corresponding escaping continuation

The CPS Semantics – Block and Branch

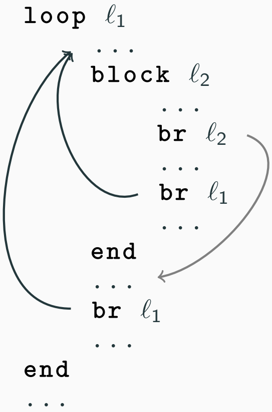
$$\begin{aligned}
&[\![\text{block } (t^m \to t^n)\ es :: rest]\!](\sigma_{arg\ m} +\!\!+\ \sigma, \rho, \kappa, \theta) = \\
&\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).[\![rest]\!](\lfloor \sigma_1 \rfloor_n +\!\!+\ \sigma, \rho_1, \kappa, \theta) \text{ in} \\
&\quad [\![es]\!](\sigma_{arg}, \rho, \kappa_1, \kappa_1 :: \theta) \\
&[\![\text{br } \ell :: rest]\!](\sigma, \rho, \kappa, \theta) \qquad\qquad\qquad\qquad = \\
&\quad \theta(\ell)(\sigma, \rho)
\end{aligned}$$



- The new continuation κ_1 is shared as ordinary continuation and escape/branch continuation
- ℓ is the de Bruijn index of the target label of the block, so $\theta(\ell)$ is the corresponding escaping continuation

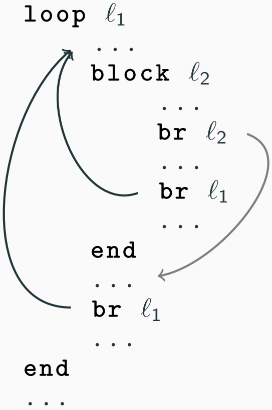
Wasm Control Flow – Loops

- Similar to blocks, loops also introduce a label as jump target
- But branching to that label will jump back to the beginning of the loop!
- If no branching happens, the loop finishes



Wasm Control Flow – Loops

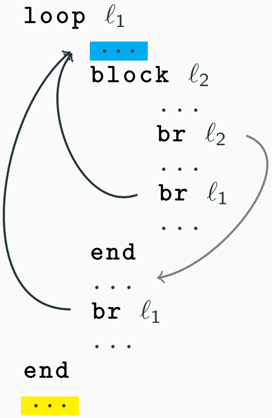
- Similar to blocks, loops also introduce a label as jump target
- But branching to that label will jump back to the beginning of the loop!
- If no branching happens, the loop finishes
- **Idea**: we need to have two different kinds of continuations for loops!



The CPS Semantics – Loops

$$
\begin{aligned}
&[\![\text{loop } (t^m \to t^n) \; es :: rest]\!](\sigma_{arg\ m} +\!\!+ \sigma, \rho, \kappa, \theta) = \\
&\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).[\![rest]\!](\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho_1, \kappa, \theta) \text{ in} \\
&\quad \text{fix } \kappa_2 := \lambda(\sigma_2, \rho_2).[\![es]\!](\lfloor \sigma_2 \rfloor_m, \rho_2, \kappa_1, \kappa_2 :: \theta) \text{ in} \\
&\quad \kappa_2(\sigma_{arg}, \rho)
\end{aligned}
$$

- κ_2 is both the body of the loop and the branch continuation
- Therefore defined recursively and appended to the trail



Call and Return

$$
\begin{aligned}
&[\![\mathsf{call}\ x :: \mathit{rest}]\!](\sigma_{\mathit{arg}\ m} +\!\!+\ \sigma, \rho, \kappa, \theta) = \\
&\qquad \mathsf{let}\ \{\mathsf{type} : t^m \to t^n, \mathsf{locals} : \mathit{ts}, \mathsf{body} : \mathit{es}\} := \mathsf{lookupFunc}(x)\ \mathsf{in} \\
&\qquad \mathsf{let}\ \rho_1 := \mathsf{buildEnv}(\sigma_{\mathit{arg}}, \mathit{ts})\ \mathsf{in} \\
&\qquad \mathsf{let}\ \kappa_1 := \lambda(\sigma_1, \rho_1).[\![\mathit{rest}]\!](\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho, \kappa, \theta)\ \mathsf{in} \\
&\qquad [\![\mathit{es}]\!]([], \rho_1, \kappa_1, [\kappa_1]) \\
&[\![\mathsf{return} :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) \qquad\qquad = \theta.\mathsf{last}(\sigma, \rho)
\end{aligned}
$$

- Discard the current trail, and install a new singleton trail containing the return continuation
- The last continuation in the trail is always the return continuation (function body is also a block, implicitly)

What is it good for?

- Now we have demonstrated the core CPS semantics

$$[\![\cdot]\!] : \text{List[Inst]} \to (\text{Stack} \times \text{Env} \times \text{Cont} \times \text{Trail}) \to \text{Ans}$$

- Trail nicely gives semantics for `block`, `loop`, `br`, `call`, and `return`
- Compositional and tail recursive

What is it good for?

- Now we have demonstrated the core CPS semantics

$$[\![\cdot]\!] : \mathsf{List}[\mathsf{Inst}] \to (\mathsf{Stack} \times \mathsf{Env} \times \mathsf{Cont} \times \mathsf{Trail}) \to \mathsf{Ans}$$

- Trail nicely gives semantics for `block`, `loop`, `br`, `call`, and `return`
- Compositional and tail recursive
- What is it good for?
  - **Specify new extensions**
  - **Equational reasoning**
  - **Run Wasm programs: interpreter**
  - Transform Wasm programs: partial evaluator
  - ...

Extending μWasm

- Structured loops
- Tail calls
- Exceptions
- Resumable exceptions
- WasmFX-style effect handlers (ongoing)

Extension 1: Tail Call

- Modeled after the current tail call proposal for WebAssembly
- Explicitly enables tail-call optimization

$$e \in \text{Instruction} ::= \cdots \mid \text{return\_call}\ x$$

Extension 1: Tail Call

$$
\begin{aligned}
&[\![\text{return\_call } x :: \mathit{rest}]\!](\sigma_{\mathit{arg}\ m} +\!\!+\ \sigma, \rho, \kappa, \theta) = \\
&\quad \text{let } \{\text{type} : t^m \to t^n, \text{locals} : \mathit{ts}, \text{body} : \mathit{es}\} := \text{lookupFunc}(x) \text{ in} \\
&\quad \text{let } \rho_1 := \text{buildEnv}(\sigma_{\mathit{arg}}, \mathit{ts}) \text{ in} \\
&\quad [\![\mathit{es}]\!]([], \rho_1, \boxed{\theta.\text{last}}, \boxed{[\theta.\text{last}]})
\end{aligned}
$$

- Instead constructing new continuations as in ordinary call, using the return continuation from the current context
- So that when return from the function body, we discard the current frame/context

Equational Reasoning for Tail Call

- Justifying the semantics of `return_call` by calculating it from the semantics of `return` and `call`

$$
\begin{aligned}
& [\![\text{call} \quad x :: \text{return} :: \mathit{rest}]\!](\sigma_{arg\ m} +\!\!+\ \sigma, \rho, \kappa, \theta) \\
= \ & \{\text{unfold call } x\} \\
& \text{let } \{\text{type} : t^m \to t^n, \text{locals} : ts, \text{body} : es\} := \text{lookupFunc}(x) \text{ in} \\
& \text{let } \rho_1 := \text{buildEnv}(\sigma_{arg}, ts) \text{ in} \\
& \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).[\![\text{return} :: \mathit{rest}]\!](\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho, \kappa, \theta) \text{ in} \\
& [\![es]\!]([], \rho_1, \kappa_1, [\kappa_1])
\end{aligned}
$$

$$
\begin{aligned}
& [\![\text{call } x :: \text{return} :: \mathit{rest}]\!](\sigma_{arg\ m} +\!\!+\, \sigma, \rho, \kappa, \theta) \\
= & \{\text{unfold call } x\} \\
& \text{let } \{\text{type} : t^m \to t^n, \text{locals} : \mathit{ts}, \text{body} : \mathit{es}\} := \text{lookupFunc}(x) \text{ in} \\
& \text{let } \rho_1 := \text{buildEnv}(\sigma_{arg}, \mathit{ts}) \text{ in} \\
& \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).[\![\text{return} :: \mathit{rest}]\!](\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho, \kappa, \theta) \text{ in} \\
& [\![\mathit{es}]\!]([], \rho_1, \kappa_1, [\kappa_1]) \\
= & \{\text{unfold return}\} \\
& \text{let } \{\text{type} : t^m \to t^n, \text{locals} : \mathit{ts}, \text{body} : \mathit{es}\} := \text{lookupFunc}(x) \text{ in} \\
& \text{let } \rho_1 := \text{buildEnv}(\sigma_{arg}, \mathit{ts}) \text{ in} \\
& \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).\ \theta.\text{last}(\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho) \text{ in} \\
& [\![\mathit{es}]\!]([], \rho_1, \kappa_1, [\kappa_1])
\end{aligned}
$$

$$
\begin{aligned}
&\text{let } \{\text{type} : t^m \to t^n, \text{locals} : ts, \text{body} : es\} := \text{lookupFunc}(x) \text{ in} \\
&\text{let } \rho_1 := \text{buildEnv}(\sigma_{arg}, ts) \text{ in} \\
&\text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).\theta.\text{last}(\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho) \text{ in} \\
&[\![es]\!]([], \rho_1, \kappa_1, [\kappa_1]) \\
= &\{\kappa_1 \text{ is } \eta\text{-equivalent to } \theta.\text{last, inlining } \kappa_1\} \\
&\text{let } \{\text{type} : t^m \to t^n, \text{locals} : ts, \text{body} : es\} := \text{lookupFunc}(x) \text{ in} \\
&\text{let } \rho_1 := \text{buildEnv}(\sigma_{arg}, ts) \text{ in} \\
&[\![es]\!]([], \rho_1, \theta.\text{last}, [\theta.\text{last}]) \\
= &\{\text{definition of return\_call}\} \\
&[\![\text{return\_call } x :: rest]\!](\sigma_{arg}\ {}_m\!+\!\!+\ \sigma, \rho, \kappa, \theta)
\end{aligned}
$$

Extension 2: Try-Catch-Resume

- A hypothetical extension of resumable exceptions
- Or, effect handlers with unlabeled single operation

$$e \in \text{Instruction} ::= \cdots \mid \text{try } es_1 \text{ catch } es_2 \mid \text{throw} \mid \text{resume}$$

Example

```
1  try
2    i32.const 1
3    call $print
4    i32.const -1 ;; error code
5    throw
6    i32.const 2
7    call $print
8  catch
9    ;; stack: [-1, resumption]
10   call $print
11   resume ;; back to line 6
12 end
```

- The resumable continuation is delimited (line 6-7)
- How do we express this behavior?

Semantics for resumable exception

- Extend continuations with meta-continuations [1]

$$
\begin{aligned}
\kappa \in \mathsf{Cont} &= \mathsf{Stack} \times \mathsf{Env} \times \mathsf{MCont} \to \mathsf{Ans} \\
m \in \mathsf{MCont} &= \mathsf{Stack} \times \mathsf{Env} \to \mathsf{Ans} \\
\gamma \in \mathsf{Handler} &= \mathsf{Stack} \times \mathsf{Env} \times \mathsf{Cont} \times \mathsf{MCont} \to \mathsf{Ans} \\
v, r \in \mathsf{Value} &::= \cdots \mid \mathsf{Stack} \times \mathsf{Env} \times \mathsf{MCont} \times \mathsf{Handler} \to \mathsf{Ans}
\end{aligned}
$$

[1] Danvy, O., Filinski, A.: Abstracting control.

Semantics for resumable exception

$$
\begin{aligned}
&[\![\text{try } es_1 \text{ catch } es_2 :: rest]\!](\sigma, \rho, \kappa, \theta, m, \gamma) = \\
&\quad \text{let } m_1 := \lambda.(\sigma_1, \rho_1).[\![rest]\!](\sigma_1, \rho_1, \kappa, \theta, m, \gamma) \\
&\quad \text{let } \gamma_1 := \lambda(\sigma_1, \rho_1, \kappa_1, m_1).[\![es_2]\!](\sigma_1, \rho_1, \kappa_1, [], m_1, \gamma) \text{ in} \\
&\quad [\![es_1]\!]([], \rho, \kappa_0, \theta, m_1, \gamma_1) \\
&[\![\text{throw} :: rest]\!](v :: \sigma, \rho, \kappa, \theta, m, \gamma) = \\
&\quad \text{let } r := \lambda(\sigma_1, \rho_1, m_1, \gamma_1).[\![rest]\!](\sigma_1, \rho_1, \kappa, \theta, m_1, \gamma_1) \text{ in} \\
&\quad \gamma([v, r], \rho, \kappa_0, m) \\
&[\![\text{resume} :: rest]\!](r :: \sigma, \rho, \kappa, \theta, m, \gamma) = \\
&\quad \text{let } m_1 := \lambda.(\sigma_1, \rho_1).[\![rest]\!](\sigma_1, \rho_1, \kappa, \theta, m, \gamma) \\
&\quad r([], \rho, m_1, \gamma)
\end{aligned}
$$

Ongoing Work

- Towards CPS Semantics of WasmFX
  - Working implementation for WasmFX's sheep handler semantics
  - Use another trail of continuations (instead of meta-continuations)
  - Formalization work-in-progress
- Implementation (in Scala) and validating against the official Wasm test suite

Future Work

- Interderivation and mechanization of semantics
  - Correspondence the big-step / CPS / small-step semantics [2,3]
  - SpecTec
  - Mechanization in theorem provers
  - . . .

[2] Danvy, O., Millikin, K.: Refunctionalization at Work.
[3] Danvy, O., Nielsen, L.R.: Defunctionalization at work.

Summary

- A CPS semantics for Wasm
  - Use a stack of continuations for `block`, `loop`, `br`, `call`, and `return`
  - Compositional and tail recursive
  - Can be implemented as a big-step interpreter or CPS transformer
- Possible extensions
  - (Hypothetical) Structured loops, try/catch, and resumable exceptions
  - (Wasm Proposals): tail calls, WasmFX
- Paper to appear in the proceedings of Trends in Functional Programming 2025