Reconstructing Continuation-Passing Semantics for WebAssembly

---

Guannan Wei [1, 2] Alexander Bai [2, 3] Dinghong Zhong [4] Jiatai Zhang [2]

Wasm Research Day
Feb 11, 2025

[1]INRIA/ENS-PSL, [2]Tufts University, [3]MPI-SWS, [4]Unaffliated

WebAssembly

- 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

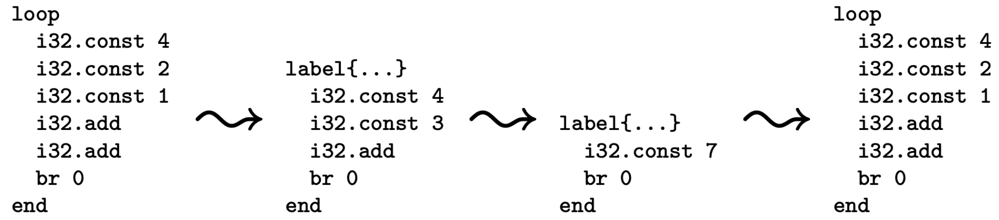
- A stack-based, low-level, fast IR for the web, now supported in major browsers
- Official formalized semantics
  - Small-step reduction dynamic semantics
  - Static type system that constrains the shape of the stack
  - Soundness and safety
- Many work-in-progress new features, e.g., effect handlers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

Wasm's Reference Small-step Semantics

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

Wasm's Reference Small-step Semantics

- Reduction: $s; v^*; e^* \to s; v^*; e^*$
- Use explicit "administrative instructions" to represent evaluation context



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

$\rightsquigarrow$

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

$\rightsquigarrow$

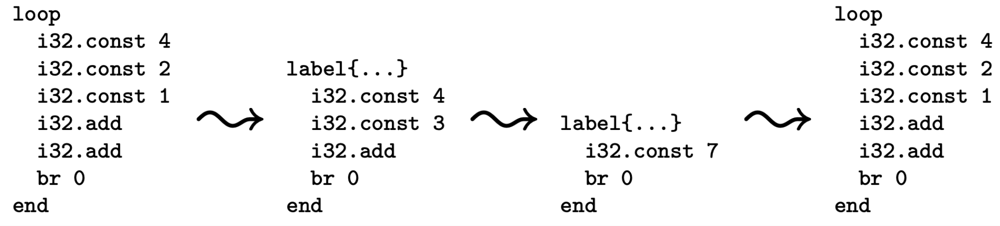
```
label{...}
  i32.const 7
  br 0
end
```

$\rightsquigarrow$

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

Wasm's Reference Small-step Semantics

- Reduction: $s; v^*; e^* \to s; v^*; e^*$
- Use explicit "administrative 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
```

- Standard approach in formalizing the semantics, straightforward to translate to an implementation of interpreters

Why Do Want an Alternative?

- **Expensive and verbose** administrative instructions
  - Time: searching on the stack in deeply nested frames/labels
  - Space: duplication of syntactic constructs

Why Do Want an Alternative?

- **Expensive and verbose** administrative instructions
  - Time: searching on the stack in deeply nested frames/labels
  - Space: duplication of syntactic constructs
- The reduction semantics is **not compositional**
  - Compositionality: obtain the “meaning” of the larger program by composing the meaning of smaller programs
  - Compositionality makes it easier for program reasoning and transformation (e.g. partial evaluation)

This Work

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

Syntax of μWasm

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

Syntax of μWasm

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

Syntax of μWasm

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

Syntax of μWasm

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

Syntax of μWasm

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

Semantics Definition

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

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

- So far it is a standard CPS “interpreter”, well-known from the 70s

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

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

The CPS Semantics – Local Registers

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

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

The CPS Semantics – Block and Branch

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

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

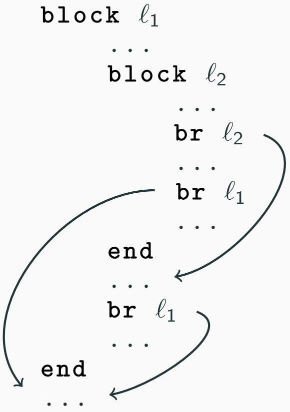
Wasm Control Flow - Blocks

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

```
block $\ell_1$
    ...
    block $\ell_2$
        ...
        br $\ell_2$
        ...
        br $\ell_1$
        ...
    end
    ...
    br $\ell_1$
    ...
end
...
```

Wasm Control Flow - Blocks

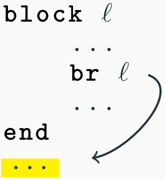
- Blocks are structured and can be nested
- A block has a label (either named or nameless as de Bruijn indices)
- The label serves as a branch target, jumping to the instruction after the block
- **Idea**: we need to remember the “escaping continuation” of every block introduced in the scope



```
block ℓ1
    ...
    block ℓ2
        ...
        br ℓ2
        ...
        br ℓ1
        ...
    end
    ...
    br ℓ1
    ...
end
...
```

The CPS Semantics – Block and Branch

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

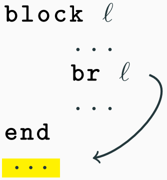


```
block ℓ
      ...
      br ℓ
      ...
end
...
```

- 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

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

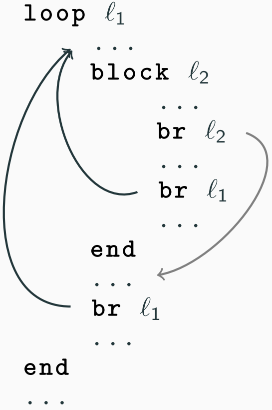


```
block ℓ
      ...
      br ℓ
      ...
end
...
```

- 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

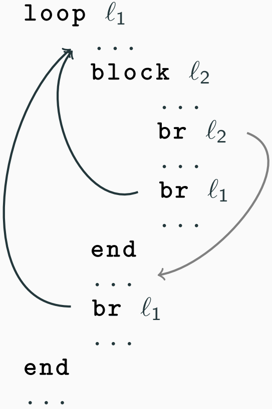
- Similar to blocks, loops also introduce a label as jump target
- But branching to that label will jump back to the beginning of the loop!
- If no branching happens, the loop finishes



```
loop ℓ1
  ...
  block ℓ2
    ...
    br ℓ2
    ...
    br ℓ1
    ...
  end
  ...
  br ℓ1
  ...
end
...
```

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 remember two different kinds of continuations for loops!

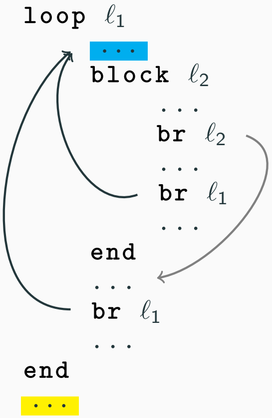


```
loop ℓ1
    ...
    block ℓ2
        ...
        br ℓ2
        ...
        br ℓ1
        ...
    end
    ...
    br ℓ1
    ...
end
...
```

The CPS Semantics – Loops

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

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



```
loop ℓ1
    ...
    block ℓ2
        ...
        br ℓ2
        ...
        br ℓ1
        ...
    end
    ...
    br ℓ1
    ...
end
...
```

Call and Return

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

- 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

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

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

What is it good for?

- 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?
  - **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)

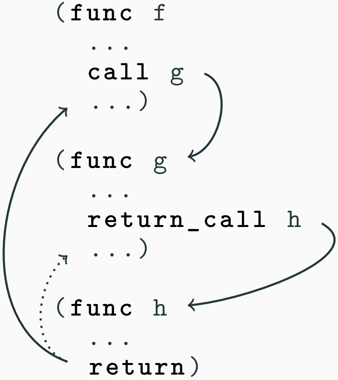
Extension 1: Tail Call

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

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

Extension 1: Tail Call

- Since `h` is a tail call, it returns to the caller of `g`
- The rest computation after `return_call h` in `g` is discarded
- Can be considered as first return, then call



```
(func f
  ...
  call g
  ...)

(func g
  ...
  return_call h
  ...)

(func h
  ...
  return)
```

Extension 1: Tail Call - CPS Semantics

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

- 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`

Equational Reasoning for Tail Call

- Justifying the semantics of `return_call` by calculating it from the semantics of `return` and `call`
- Now let's pretend a call is made at a tail position:

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

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

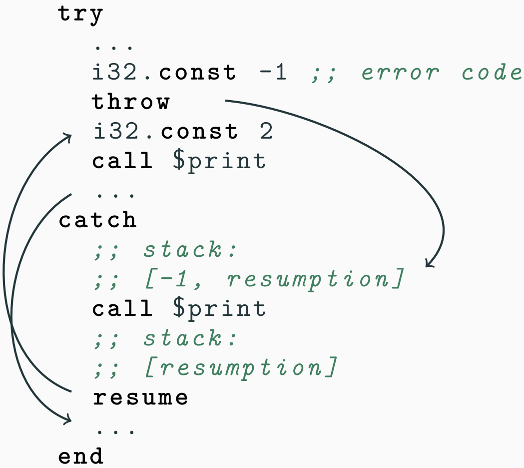
$$\begin{aligned}
&\quad \text{let } \{\text{type} : t^{m} \to t^{n}, \text{locals} : ts, \text{body} : es\} \coloneqq \text{lookupFunc}(x) \text{ in}\\
&\quad \text{let } \rho_1 \coloneqq \text{buildEnv}(\sigma_{arg}, ts) \text{ in}\\
&\quad \text{let } \kappa_1 \coloneqq \lambda(\sigma_1, \rho_1).\theta.\text{last}(\lfloor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

Extension 2: Try-Catch-Resume

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

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

Example



```
try
  ...
  i32.const -1 ;; error code
  throw
  i32.const 2
  call $print
  ...
catch
  ;; stack:
  ;; [-1, resumption]
  call $print
  ;; stack:
  ;; [resumption]
  resume
  ...
end
```

- Resumption continuation is a proper value on the stack
- The resumable continuation is delimited within the `try` block
- When `try` block finishes, the control flow continues to the instruction after `resume`
- How do we express this behavior?

Semantics for resumable exception

- Extend continuations with meta-continuations [1]

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

---

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

Semantics for resumable exception

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

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

- Staging the interpreter for partial evaluation
  - Turn the interpreter into a code generator
- 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
- Implementation: https://github.com/Generative-Program-Analysis/wasm-cps
- Paper to appear in the proceedings of Trends in Functional Programming 2025