Reconstructing Continuation-Passing Semantics for WebAssembly

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

Trends in Functional Programming
Jan 15 2025

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

WebAssembly (Wasm)

- A stack-based low-level IR for the web (supported in most major browsers)

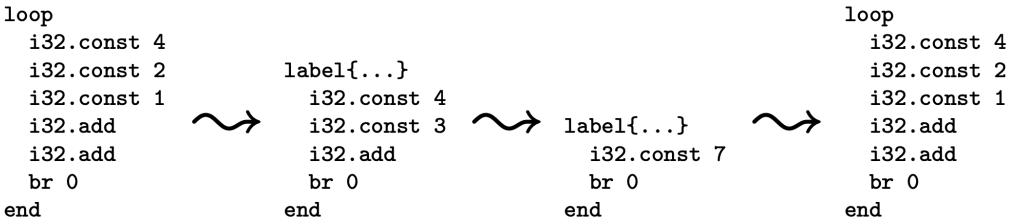
WebAssembly (Wasm)

- A stack-based low-level IR for the web (supported in most 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)

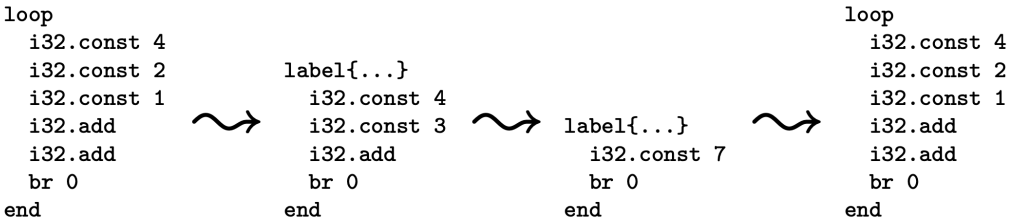
- A stack-based low-level IR for the web (supported in most major browsers)
- Official formalized semantics
  - Small-step reduction dynamic semantics
  - Static type system that constrains the shape of the stack
  - Soundness and safety
- Many work-in-progress new features, e.g., effect handlers (WasmFX)

Wasm's Reference Small-step Semantics



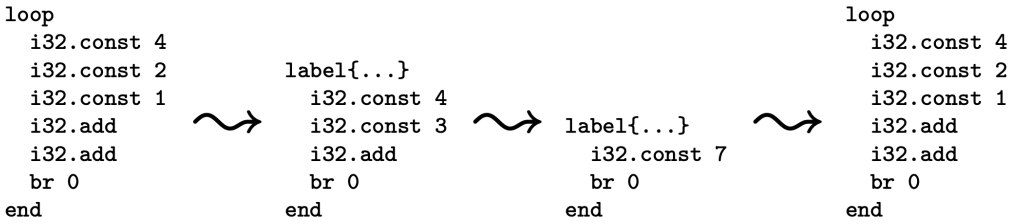
- Explicit and verbose "adminstrative instructions" to represent evaluation context

Wasm's Reference Small-step Semantics



- Explicit and verbose "adminstrative instructions" to represent evaluation context
- Mixes control and value stack, causing inefficiencies in deeply nested frames/labels

Wasm's Reference Small-step Semantics



- Explicit and verbose "adminstrative instructions" to represent evaluation context
- Mixes control and value stack, causing inefficiencies in deeply nested frames/labels
- *Not compositional* and *not tail-recursive*, not ideal for mechanical program transformations

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 Wasm in CPS
  - Implemented as a big-step interpreter
  - Or, can be viewed as a CPS transformer

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 Wasm in CPS
  - Implemented as a big-step interpreter
  - Or, can be viewed as a CPS transformer
- Extensions for control constructs
  - (Hypothetical) Structured loops, try/catch, and resumable exceptions
  - (Wasm Proposals): tail calls, effect handlers (ongoing)

Syntax of μWasm

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

Semantics Definition

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

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

First Glimpse of the CPS Semantics

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

$$
\begin{aligned}
[\![\mathit{nil}]\!](\sigma, \rho, \kappa, \theta) &= \kappa(\sigma, \rho) \\
[\![\mathsf{nop} :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](\sigma, \rho, \kappa, \theta)
\end{aligned}
$$

First Glimpse of the CPS Semantics

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

$$
\begin{aligned}
[\![\mathit{nil}]\!](\sigma, \rho, \kappa, \theta) &= \kappa(\sigma, \rho) \\
[\![\mathsf{nop} :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](\sigma, \rho, \kappa, \theta) \\
[\![t.\mathsf{const}\ c :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](c :: \sigma, \rho, \kappa, \theta)
\end{aligned}
$$

First Glimpse of the CPS Semantics

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

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

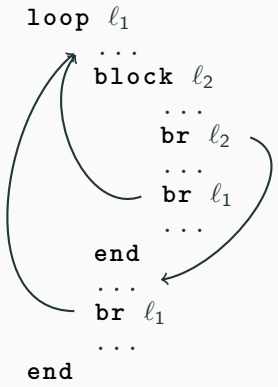
First Glimpse of the CPS Semantics

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

$$
\begin{aligned}
[\![\mathit{nil}]\!](\sigma, \rho, \kappa, \theta) &= \kappa(\sigma, \rho) \\
[\![\mathsf{nop} :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](\sigma, \rho, \kappa, \theta) \\
[\![t.\mathsf{const}\ c :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](c :: \sigma, \rho, \kappa, \theta) \\
[\![t.\mathsf{add} :: \mathit{rest}]\!](v_1 :: v_2 :: \sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](v_1 + v_2 :: \sigma, \rho, \kappa, \theta) \\
[\![\mathsf{local.get}\ x :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](\rho(x) :: \sigma, \rho, \kappa, \theta) \\
[\![\mathsf{local.set}\ x :: \mathit{rest}]\!](v :: \sigma, \rho, \kappa, \theta) &= [\![\mathit{rest}]\!](\sigma, \rho[x \mapsto v], \kappa, \theta)
\end{aligned}
$$

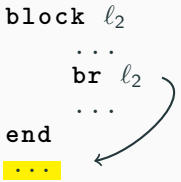
Wasm Control Flow

- What's interesting about Wasm's control semantics?



Control Flow Instructions

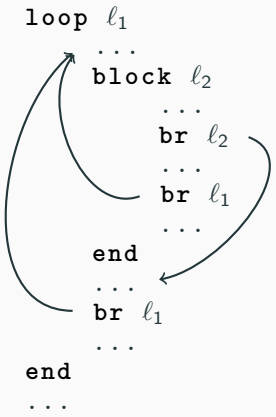
$$[\![\text{block } (t^m \to t^n)\ es :: rest]\!](\sigma_{arg\ m} +\!\!+ \sigma, \rho, \kappa, \theta) =$$
$$\text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).[\![rest]\!](\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho_1, \kappa, \theta) \text{ in}$$
$$[\![es]\!](\sigma_{arg}, \rho, \kappa_1, \kappa_1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



- The only exit point (br or fall-through) this block creates is *rest*
- ℓ is the de Bruijn index of the target label of the block, so $\theta(\ell)$ is the corresponding escaping continuation

Control Flow Instructions

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



Call and Return

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

What is it good for?

- Now we have demonstrated the core CPS semantics

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

- The semantics is 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 (effect handlers, ongoing)

Extension 1: Tail Call

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

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

Equational Reasoning for Tail Call

- Let's calculate the semantics of return_call from Call then Return

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

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

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

Extension 2: Try-Catch-Resume

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

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

- From continuations to 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

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

Implementation

```
type Stack = List[Value]
type Env = Map[Int, Value]
type Cont[A] = List[Value] ⇒ A
type Trail = List[Cont[Ans]]

def eval[Ans](insts: List[Inst], stack: Stack, env: Env,
              k: Cont[Ans], trail: Trail): Ans =
  insts match
    case Nil ⇒ k(stack, env)
    case Binary(op) ⇒
      val v2 :: v1 :: newStack = stack
      val result = evalBinOp(op, v1, v2)
      eval(rest, result :: newStack, env, k, trail)
    ... // more instructions
```

Ongoing & Future Work

- Validating against the official Wasm test suite
- Interderivation between the big step / CPS / small step semantics [2][3]
- Mechanization
- WasmFX support

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

Summary

- An alternative to reduction semantics of Wasm using CPS
- A compositional and tail recursive semantics
  - Implemented as a big-step interpreter
  - Or, can be viewed as a CPS transformer
- Extensions for control constructs
  - (Hypothetical) Structured loops, try/catch, and resumable exceptions
  - (Wasm Proposals): tail calls, effect handlers (ongoing)