Mixing Transformation and Symbolic Execution with Continuation for WebAssembly

Guannan Wei, Dinghong Zhong, Alex Bai
Tufts University

OlivierFest at SPLASH/ICFP
Singapore, 2025

Motivation

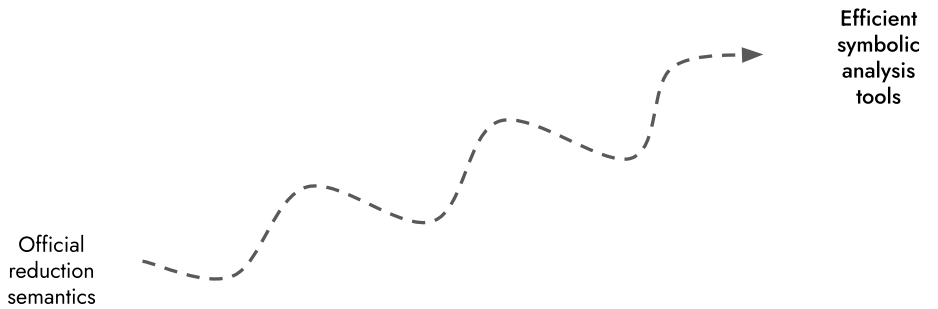
WebAssembly

- WebAssembly is a low-level IR aimed to be safe, portable, and efficient
- WebAssembly comes with an official formal specification

From official semantics to efficient symbolic analysis tools

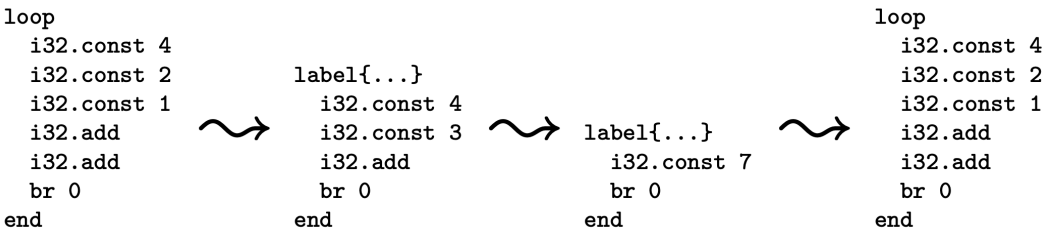
This talk

A few mechanical steps from official semantics to efficient symbolic analysis tools using continuations and staging.



Official Wasm reduction semantics

- Official specification: a small-step reduction semantics
  - Structured control flow (`block`, `loop`, `if`, etc.)
  - Administrative instructions (`label`, etc.) representing evaluation context



Downsides of using “administrative instructions”

- Code duplication and search over the context -> inefficient

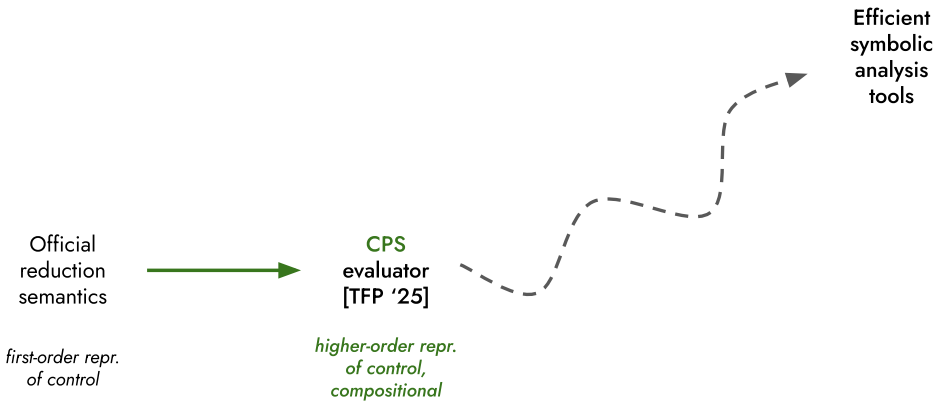
Downsides of using “administrative instructions”

- Code duplication and search over the context -> inefficient

But to use this semantics to derive faster implementations using partial evaluation:

- Not part of the source language -> binding-time conflation
- Non-compositional -> unfolding the interpreter doesn’t work straightforwardly

From official semantics to efficient symbolic analysis tools



Syntax of μWasm

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

Semantics Definition

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

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

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

$$[\![\text{add} :: \mathit{rest}]\!](v_1 :: v_2 :: \sigma, \rho, \kappa, \theta) = [\![\mathit{rest}]\!](v_1 + v_2 :: \sigma, \rho, \kappa, \theta)$$

The CPS Semantics – Block

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

$$[\![\text{block } (t^m \to t^n) \; es :: rest]\!](\sigma_{arg\ m} +\!\!+ \sigma, \rho, \kappa, \theta) = ???$$

$$[\![\text{br } \ell :: rest]\!](\sigma, \rho, \kappa, \theta) = ???$$

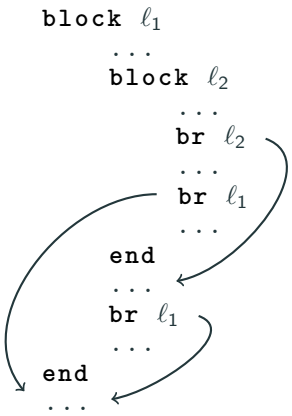
Wasm Control Flow - Blocks

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

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

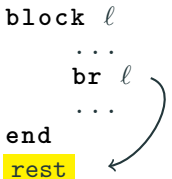
Wasm Control Flow - Blocks

- Blocks are structured and can be nested
- A block has a label (can be 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 each block introduced in the scope



The CPS Semantics – Block

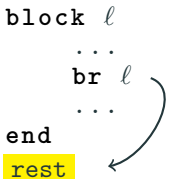
$$
\begin{aligned}
&[\![\text{block } (t^m \to t^n)\ es :: rest]\!](\sigma_{arg\ m} +\!\!+ \sigma, \rho, \kappa, \theta) = \\
&\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).[\![rest]\!](\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho_1, \kappa, \theta) \text{ in} \\
&\quad [\![es]\!](\sigma_{arg},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



- 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

The CPS Semantics – Block

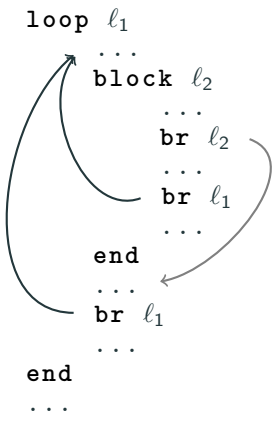
$$
\begin{aligned}
&[\![\text{block } (t^m \to t^n) \; es :: rest]\!](\sigma_{arg\ m} +\!\!+ \sigma, \rho, \kappa, \theta) = \\
&\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).[\![rest]\!](\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho_1, \kappa, \theta) \text{ in} \\
&\quad [\![es]\!](\sigma_{arg},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



- 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

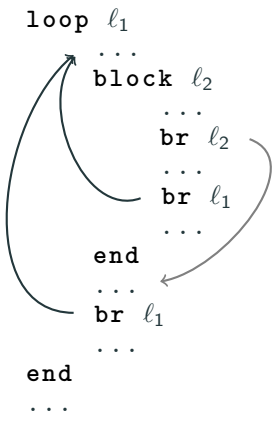
Wasm Control Flow – Loops

- 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

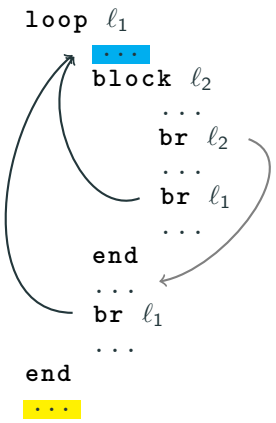
- Similar to blocks, loops also introduce a label as jump target
- But branching to that label will jump back to the beginning of the loop!
- If no branching happens, the loop finishes
- **Idea**: we need to remember two different continuations for loops!



The CPS Semantics – Loops

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

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



Call and Return

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

- 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)

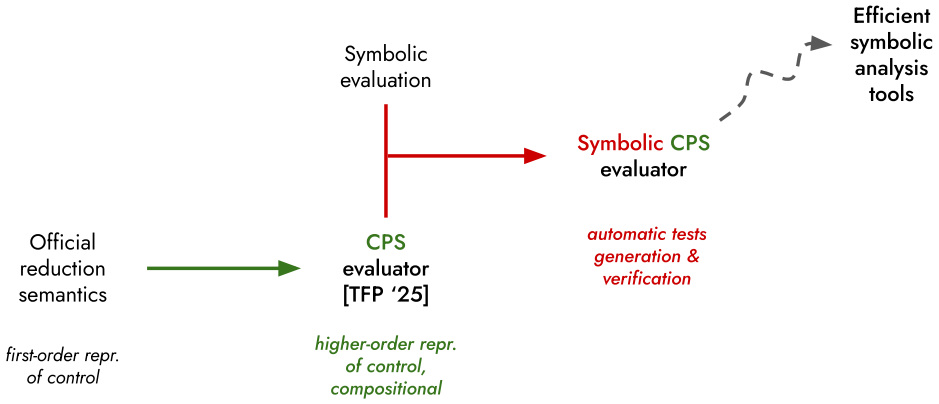
Definitional Interpreter for WebAssembly

- Now we have an evaluator in continuation-passing style with a trail:

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

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

From official semantics to efficient symbolic analysis tools

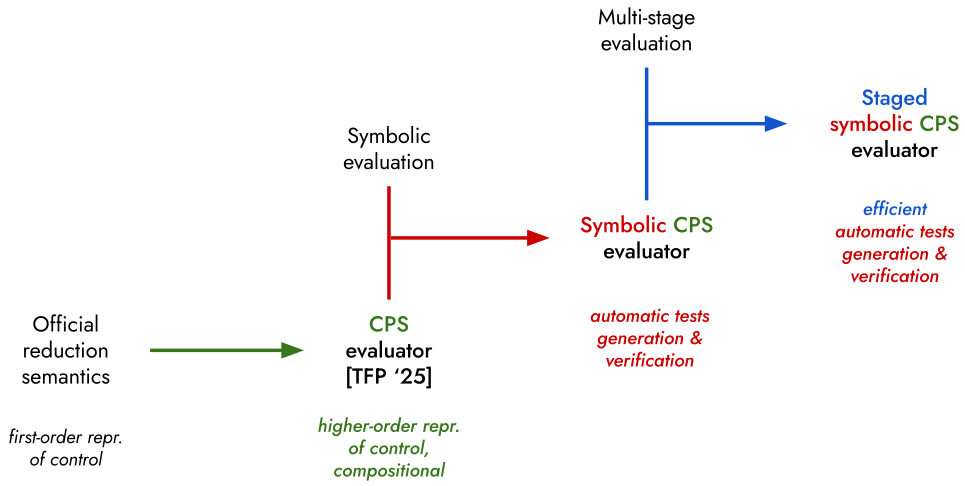


From interpreter to symbolic interpreter

- Symbolic evaluator maintains symbolic expressions on stack/env and path conditions (PC):

$$\begin{aligned}[\![\cdot]\!]_S : \mathsf{List}[\mathsf{Inst}] \to \quad & (\mathsf{Stack} \times \mathsf{Env} \\ & \times \mathsf{SymStack} \times \mathsf{SymEnv} \times \mathsf{PC} \\ & \times \mathsf{Cont} \times \mathsf{List}[\mathsf{Cont}]) \to \mathsf{Ans}\end{aligned}$$

From official semantics to efficient symbolic analysis tools



From symbolic interpreter to staged symbolic interpreter

- A two-stage symbolic interpreter
  - Staging removes interpretation overhead
  - Trail list is eliminated at compile/staging-time (vs. Rep[List[Cont]])

$$[\![\cdot]\!]_S^{\downarrow\uparrow} : \mathsf{List}[\mathsf{Inst}] \to (\mathsf{Rep}[\mathsf{Stack}] \times \mathsf{Rep}[\mathsf{Env}] \times \mathsf{Rep}[\mathsf{SymStack}] \times \mathsf{Rep}[\mathsf{SymEnv}] \times \mathsf{Rep}[\mathsf{PC}] \times \mathsf{Rep}[\mathsf{Cont}] \times \mathsf{List}[\mathsf{Rep}[\mathsf{Cont}]]) \to \mathsf{Rep}[\mathsf{Ans}]$$

From symbolic interpreter to staged symbolic interpreter

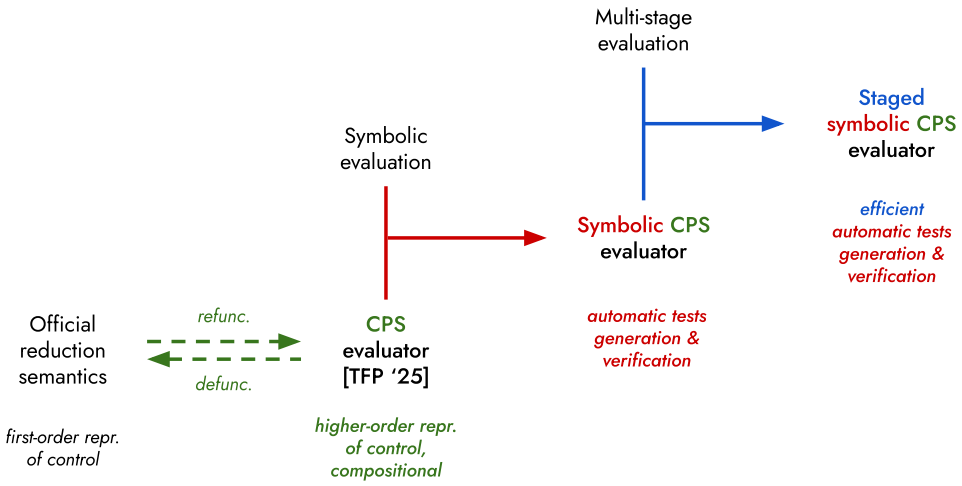
- A two-stage symbolic interpreter
  - Staging removes interpretation overhead
  - Trail list is eliminated at compile/staging-time (vs. Rep[List[Cont]])

$$[\![\cdot]\!]_S^{\downarrow\uparrow} : \text{List[Inst]} \to \begin{array}{l}(\text{Rep[Stack]} \times \text{Rep[Env]} \\ \times \text{Rep[SymStack]} \times \text{Rep[SymEnv]} \times \text{Rep[PC]} \\ \times \text{Rep[Cont]} \times \text{List[Rep[Cont]]}) \to \text{Rep[Ans]}\end{array}$$

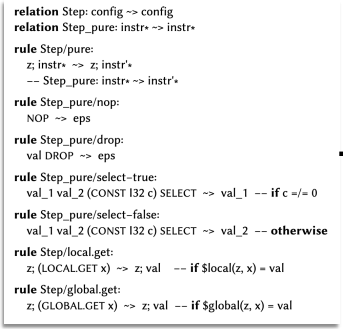
- Thanks to CPS:
  - Staging the interpreter is straightforward (unfolding)
  - Enables snapshot-reuse optimizations by remembering the continuation
- Talk at WebAssembly Workshop on Thursday

17:05 **20m** ☆ **Efficient Concolic Execution of WebAssembly by Compilation and Snapshot Reuse**
Talk Dinghong Zhong Tufts University, Alexander Bai New York University, Guannan Wei Tufts University

Functional correspondence



New opportunities for the (inter-)derivation of semantic artifacts



```
relation Step: config ~> config
relation Step_pure: instr* ~> instr*

rule Step/pure:
  z; instr* ~> z; instr'*
  -- Step_pure: instr* ~> instr'*

rule Step_pure/nop:
  NOP ~> eps

rule Step_pure/drop:
  val DROP ~> eps

rule Step_pure/select-true:
  val_1 val_2 (CONST I32 c) SELECT ~> val_1 -- if c =/= 0

rule Step_pure/select-false:
  val_1 val_2 (CONST I32 c) SELECT ~> val_2 -- otherwise

rule Step/local.get:
  z; (LOCAL.GET x) ~> z; val -- if $local(z, x) = val

rule Step/global.get:
  z; (GLOBAL.GET x) ~> z; val -- if $global(z, x) = val
```

Official reduction semantics

first-order repr. of control

refunc.

defunc.

SpecTec [Youn et al., PLDI '24]

CPS evaluator [TFP '25]

higher-order repr. of control, compositional

Multi-stage evaluation

Can we automate this derivation?

Symbolic evaluation

Symbolic CPS evaluator

automatic tests generation & verification

Staged symbolic CPS evaluator

efficient automatic tests generation & verification

Conclusion

WebAssembly CPS semantics + symbolic evaluation + staging:

- CPS semantics eliminates administrative instructions
- Symbolic evaluation maintains symbolic expressions and path conditions
- Amenable to be partially evaluated/staged due to compositionality
- Result: efficient symbolic analysis tools

Ongoing/future work:

- Automating the derivation with SpecTec specification as input