Towards Performant Static Analysis of WebAssembly via Staging and Continuations

Guannan Wei, INRIA/ENS, Tufts University
with Dinghong Zhong and Alexander Bai

Dagstuhl Seminar 25241

Motivation

- Goal: fast, correct, automated static analysis of Wasm programs
  - Examples: symbolic execution, abstract interpretation, etc.
  - Useful for test case generation, verification, security auditing, etc.

Motivation

- The “reference” semantics/interpreter is useful for specifying and implementing analyses:
- Defines true dynamic behaviors (simulation, Galois connection, etc.)
- Analyses implemented as non-standard interpreters (collecting semantics, SMT invocation, etc.)

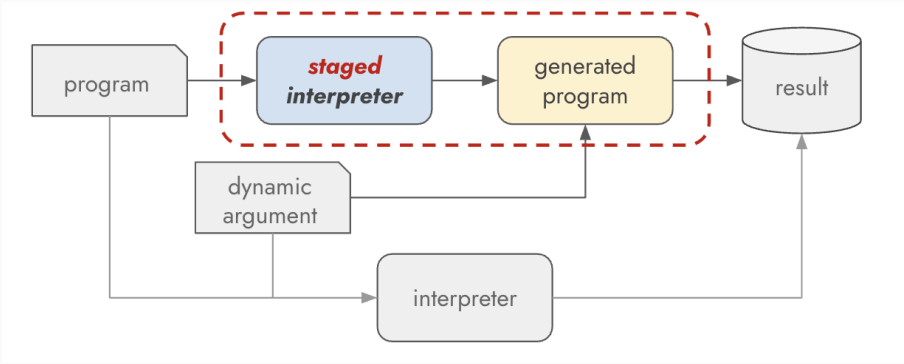
Improving Performance of Wasm Static Analysis

- Performance matters
- **Interpretation overhead**:
  - Traverse the program representation (AST, etc.) of multiple paths; fixpoint iteration; etc.
  - angr symbolic execution: 321,000x slower than native execution! (USENIX Sec '18)

Improving Performance of Wasm Static Analysis

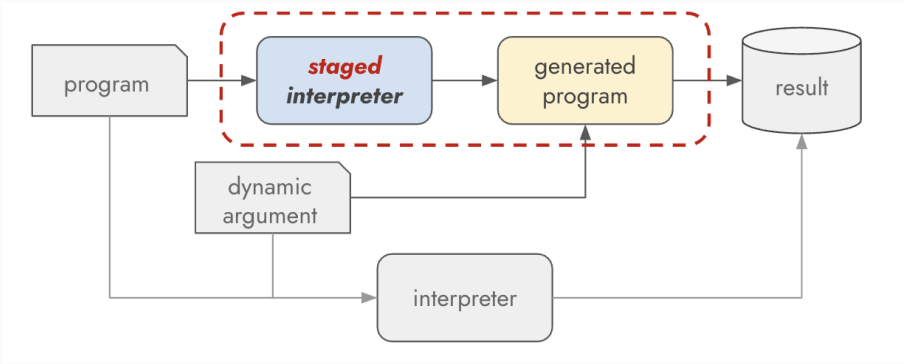
- Performance matters
- **Interpretation overhead**:
  - Traverse the program representation (AST, etc.) of multiple paths; fixpoint iteration; etc.
  - angr symbolic execution: 321,000x slower than native execution! (USENIX Sec '18)
- **Staging can help**: specialize (i.e. partially evaluate) the non-standard interpreter to a specific program
  - Generate a residual program and removes interpretation overhead

Improving Performance of Wasm Static Analysis via Staging



- 1st Futamura projection: a staged interpreter is a “compiler”

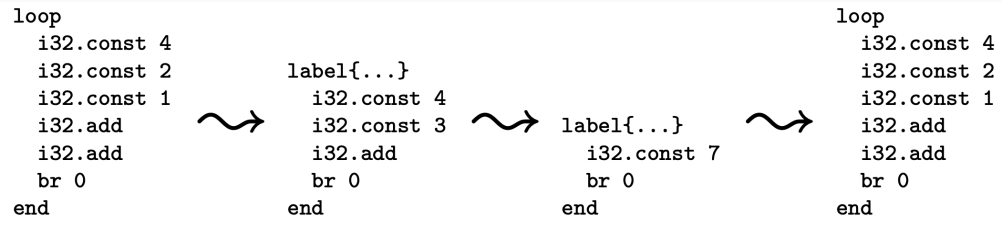
Improving Performance of Wasm Static Analysis via Staging



- 1st Futamura projection: a staged interpreter is a “compiler”
- But, not all interpreters can be well staged . . .

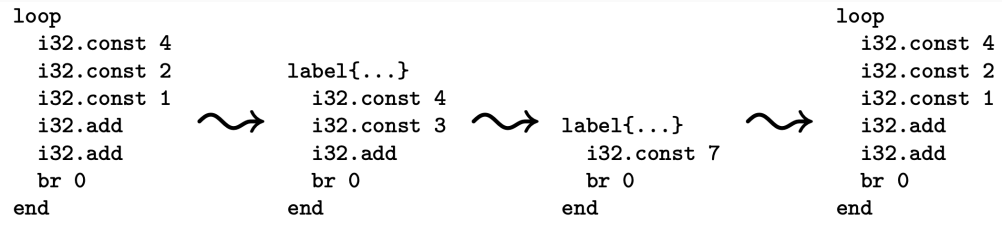
Administrative Instructions Disturb Specialization

- Using "administrative instructions" to represent evaluation context for structured control flow (blocks, loops, etc.)



Administrative Instructions Disturb Specialization

- Using "administrative instructions" to represent evaluation context for structured control flow (blocks, loops, etc.)



- Administrative instructions are not part of the original program (static binding time), but generated at "runtime" (dynamic binding time)
- Staging requires the whole program to be statically known

Non-Compositionality

- Compositionality: obtain the "meaning" of the larger program by composing the meaning of smaller programs
- Much easier for program reasoning and transformation (e.g. partial evaluation)

An Alternative CPS Semantics for Wasm

- An alternative to reduction semantics of Wasm:
  - “Rewriting” vs “denotation”
  - Rather than the first-order representation for control structures, we use continuation functions in the meta-language to represent control semantics

An Alternative CPS Semantics for Wasm

- An alternative to reduction semantics of Wasm:
  - "Rewriting" vs "denotation"
  - Rather than the first-order representation for control structures, we use continuation functions in the meta-language to represent control semantics
- A compositional control-flow semantics for core Wasm in *continuation-passing style* (CPS)

An Alternative CPS Semantics for Wasm

- An alternative to reduction semantics of Wasm:
  - "Rewriting" vs "denotation"
  - Rather than the first-order representation for control structures, we use continuation functions in the meta-language to represent control semantics
- A compositional control-flow semantics for core Wasm in *continuation-passing style* (CPS)
- Paper at Trends in Functional Programming 2025: Reconstructing Continuation-Passing Semantics for WebAssembly

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

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

- So far it is a standard "interpreter" in CPS, 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: $[\![\cdot]\!] : \text{List[Inst]} \to (\text{Stack} \times \text{Env} \times \text{Cont} \times \text{Trail}) \to \text{Ans}$

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

The CPS Semantics – Block

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

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

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

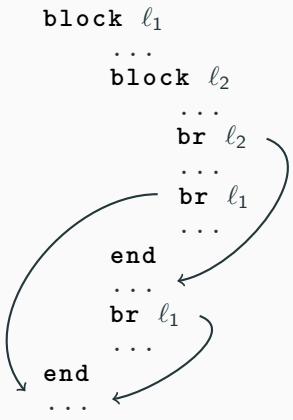
Wasm Control Flow - Blocks

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

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

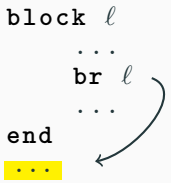
Wasm Control Flow - Blocks

- Blocks are structured and can be nested
- A block has a label (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

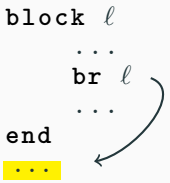
$$
\begin{aligned}
&[\![\text{block } (t^m \to t^n)\ es :: rest]\!](\sigma_{arg\ m} +\!\!+ \sigma, \rho, \kappa, \theta) = \\
&\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).[\![rest]\!](\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho_1, \kappa, \theta) \text{ in} \\
&\quad [\![es]\!](\sigma_{arg},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



- 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

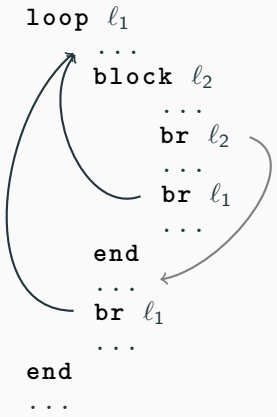
$$
\begin{aligned}
&[\![\text{block } (t^m \to t^n)\ es :: rest]\!](\sigma_{arg\ m} +\!\!+ \sigma, \rho, \kappa, \theta) = \\
&\quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1).[\![rest]\!](\lfloor \sigma_1 \rfloor_n +\!\!+ \sigma, \rho_1, \kappa, \theta) \text{ in} \\
&\quad [\![es]\!](\sigma_{arg},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



- 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

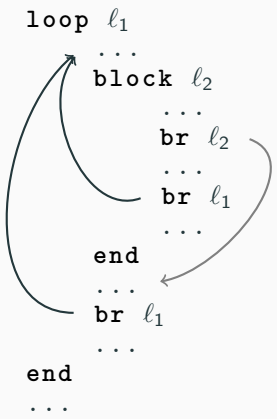
Wasm Control Flow – Loops

- 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

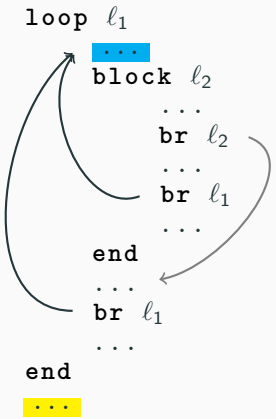
- Similar to blocks, loops also introduce a label as jump target
- But branching to that label will jump back to the beginning of the loop!
- If no branching happens, the loop finishes
- **Idea**: we need to remember two different kinds of continuations for loops!



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



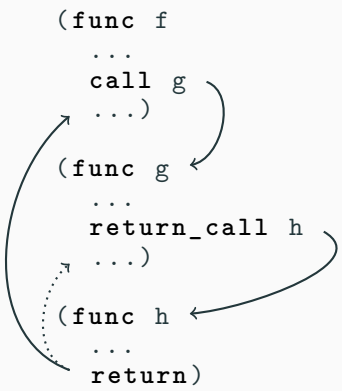
Call and Return

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

- 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)

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



Tail Call

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

- Instead of constructing new continuation with *rest*, using the return continuation from the current context
- So that when return from the function body, we discard the current frame/context
- Equational reasoning to calculate:

$$[\![\text{return\_call } x :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta) = [\![\text{call } x :: \text{return} :: \mathit{rest}]\!](\sigma, \rho, \kappa, \theta)$$

Extending the CPS Semantics

- WasmFX-style effect handlers
  - cont.new, resume, suspend
  - Augment the semantics with another trail of continuations
- ...

Interpreter

- Now we have demonstrated the core CPS semantics

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

  - Trail nicely gives semantics for `block`, `loop`, `br`, `call`, and `return`
  - Compositional and tail recursive
- Can be easily implemented as a concise definitional interpreter for Wasm

From Interpreter to Compiler

- A **staged** concrete Wasm interpreter:

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

  - Trail is eliminated at compile/staging-time (in contrast to Rep[List[Cont]])

From Interpreter to Compiler

- A **staged** concrete Wasm interpreter:

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

  - Trail is eliminated at compile/staging-time (in contrast to Rep[List[Cont]])
- Generating C/C++ code using Scala/LMS, no fancy optimizations yet
- Preliminary result: 7-12x speedup vs our Scala CPS interpreter on some micro benchmarks

From Interpreter to Compiler

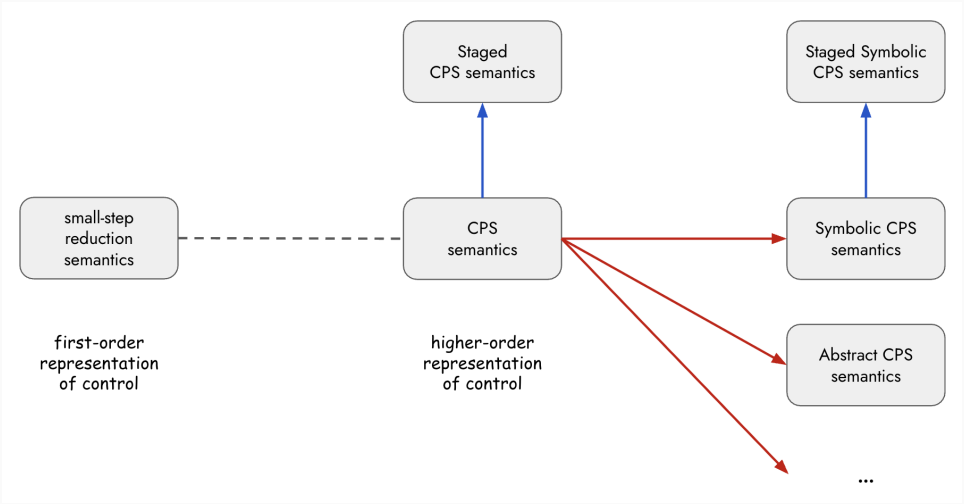
- A **staged** concrete Wasm interpreter:

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

  - Trail is eliminated at compile/staging-time (in contrast to Rep[List[Cont]])
- Generating C/C++ code using Scala/LMS, no fancy optimizations yet
- Preliminary result: 7-12x speedup vs our Scala CPS interpreter on some micro benchmarks
- Next step: specialize a symbolic interpreter $[\![\cdot]\!]^{\mathbb{S}}_{\uparrow\downarrow}$ and generate C/C++ code to perform symbolic execution

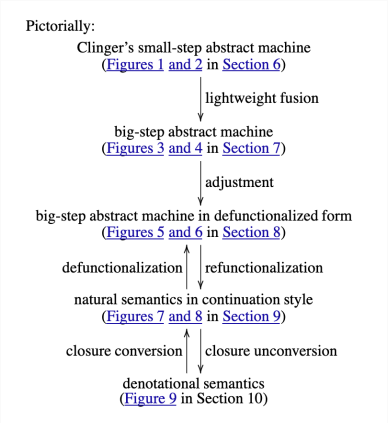
Bigger Picture

- CPS semantics is more friendly for staging and spcifying executable analysis



Bigger Picture

- Olivier Danvy's agenda on interderivable semantic specifications
- Example: Towards Compatible and Interderivable Semantic Specifications for the Scheme Programming Language



Conclusion

- SpecTec seems to be a good playground for interderivable semantics
  - Choose one and automatically derive the others?
- Performant static analysis
  - Staging + compositional interpreter with continuation + analysis semantics