Scheme on WebAssembly

It is happening!

7 September 2024 – Scheme ’24

Andy Wingo

Igalia, S.L.

Agenda

Context: Where can users run Scheme?

How to run Scheme on WebAssembly

Hoot: Guile on WebAssembly (with a scenic detour)

Getting Scheme to the user in 2024

Native executables: Fine, but waning as a distribution paradigm

HTTP: On the internet, nobody knows your web server is a Scheme

Browsers: On the client, JS is king

Servers: Docker, K8s; but, ugh

Browsers are special

Users install dozens of programs a day

Largely on mobile, often low-powered

Network + mobile CPU: strong size constraint

Executable format: JS source code

Scheme to JS?

JS virtual machines are amazing

- World-class garbage collectors
- Tiered compilers: low latency *and* high throughput
- Ubiquitous, evergreen
- Well-resourced

Can != Should

JS is not a great compile target

- Monkeypatching
- Unpredictable performance
- Poor numerics
- No tail calls
- Limited stack size
- Single-threaded
- Emitting source is gross

Meanwhile, in C++-land

2013: C++ to JS with `asm.js`

2017: C++ to WebAssembly

Typed functions operating on *linear memory*

Capabilities explicitly granted by *host functions* (e.g. `gl.attachShader`)

GC, tail calls always planned, but not present initially

Scheme to Wasm?

Not at first!

Choice was between:

- Nice virtual machine, no GC
- Gross source language, great GC

If you care about users, JS beat Wasm as a compile target

...until now

gc pause

WasmGC is now in Firefox, Chrome, and Safari (preview)

Tail calls too

It’s wassembly time!

Let us connive:

- What kind of code should we contrive
- Getting there from here: Hoot compiler deep dive

Scheme to WasmGC

```
      any     extern      func
       |
       eq
    /  |   \
i31 struct  array
```

The unitype: `(ref eq)`

Immediate values in `(ref i31)`

- fixnums with 30-bit range
- chars, bools, etc

Explicit nullability: `(ref null eq)` vs `(ref eq)`

Object representation

```
(rec
 (type $heap-object
   (sub
    (struct (field $hash i32))))
 (type $pair
   (sub $heap-object
     (struct (field $hash i32)
             (field $car (ref eq))
             (field $cdr (ref eq)))))
 ...)
```

WasmGC allows subtyping on structs, functions, arrays

Structural type equivalance, but types in rec group distinct

Working with values

```
(func $cons (param $car (ref eq))
            (param $cdr (ref eq))
            (result (ref $pair))
  (struct.new $pair
    (i32.const 0)
    (local.get $car)
    (local.get $cdr)))

(func $%car (param $pair (ref $pair))
            (result (ref eq))
  (struct.get $pair $car (local.get $pair)
```

Dynamic type checks

```
(func $car (param $obj (ref eq))
           (result (ref eq))
  (block $not-pair
    (return_call $%car
      (br_on_cast_fail $not-pair
                       (ref eq)
                       (ref $pair)
        (local.get $obj))))
  (call $type-error)
  (unreachable))
```

Varargs

```
(list 'hey)        ;; => (hey)
(list 'hey 'icfp) ;; => (hey icfp)
```

Wasm functions strongly typed

```
(func $list (param ???) (result (ref eq))
  ???)
```

Varargs workaround

Uniform function type

```
(type $argv (array (ref eq)))

(type $kvarargs
  (func (param $nargs i32)
        (param $arg0 (ref eq))
        (param $arg1 (ref eq))
        (param $arg2 (ref eq))
        (param $argv (ref null $argv))
        (result (ref eq)))) ;; *
```

```
(func $checked-car (param $nargs i32)
                   (param $arg0 (ref eq))
                   (param $arg1 (ref eq))
                   (param $arg2 (ref eq))
                   (param $argv (ref null $argv))
                   (result (ref eq))
  (block $bad-arity
    (br_if $bad-arity
           (i32.ne (local.get $nargs)
                   (i32.const 2)))
    (return_call $car (local.get $arg1)))
  (call $throw-wrong-number-of-arguments)
  (unreachable))
```

Multiple values?

Do the same kind of `$kvarargs` treatment as calls?

This is getting a little silly

Layering dynamic typing over static typing is more overhead than direct implementation in VM

Let’s get back to this later

Prompts

Guile uses prompts for lightweight threads/fibers, exceptions

“Bring your whole self”

Idea: CPS-convert to stack-allocate return continuations

Delimited continuations are then just stack slices

Stack allocation of return continuations

To make a non-tail-call:

- Push live-out vars on stacks (one stack per top type)
- Push return continuation on stack
- Tail-call callee

```
(type $kvarargs
  (func (param $nargs i32)
        (param $arg0 (ref eq))
        (param $arg1 (ref eq))
        (param $arg2 (ref eq))
        (param $argv (ref null $argv))))
```

Returns are tail calls

```
(func $values (param $nargs i32)
              (param $arg0 (ref eq))
              (param $arg1 (ref eq))
              (param $arg2 (ref eq))
              (param $argv (ref null $argv
  (return_call_ref
    (call $pop-return-stack)
    (local.get $nargs)
    (local.get $arg0)
    (local.get $arg1)
    (local.get $arg2)
    (local.get $argv)))
```

After return, continuation pops state from stacks

Prompts for free

`abort-to-prompt`:

- Pop stack slice to reified continuation object
- Tail-call new top of stack: prompt handler

Calling a reified continuation:

- Push stack slice
- Tail-call new top of stack

No need to wait for effect handlers proposal; you can have it all now!

Hoot!

Hoot is Guile for WebAssembly

- Whole-program compilation, single binary
- Same source language, different implementation
- Shared front-end and middle-end

Goal: Get Spritely’s Goblins on the web

https://davexunit.itch.io/cirkoban

keyboard:

arrows -> move

Z -> undo

touchscreen:

dpad -> move

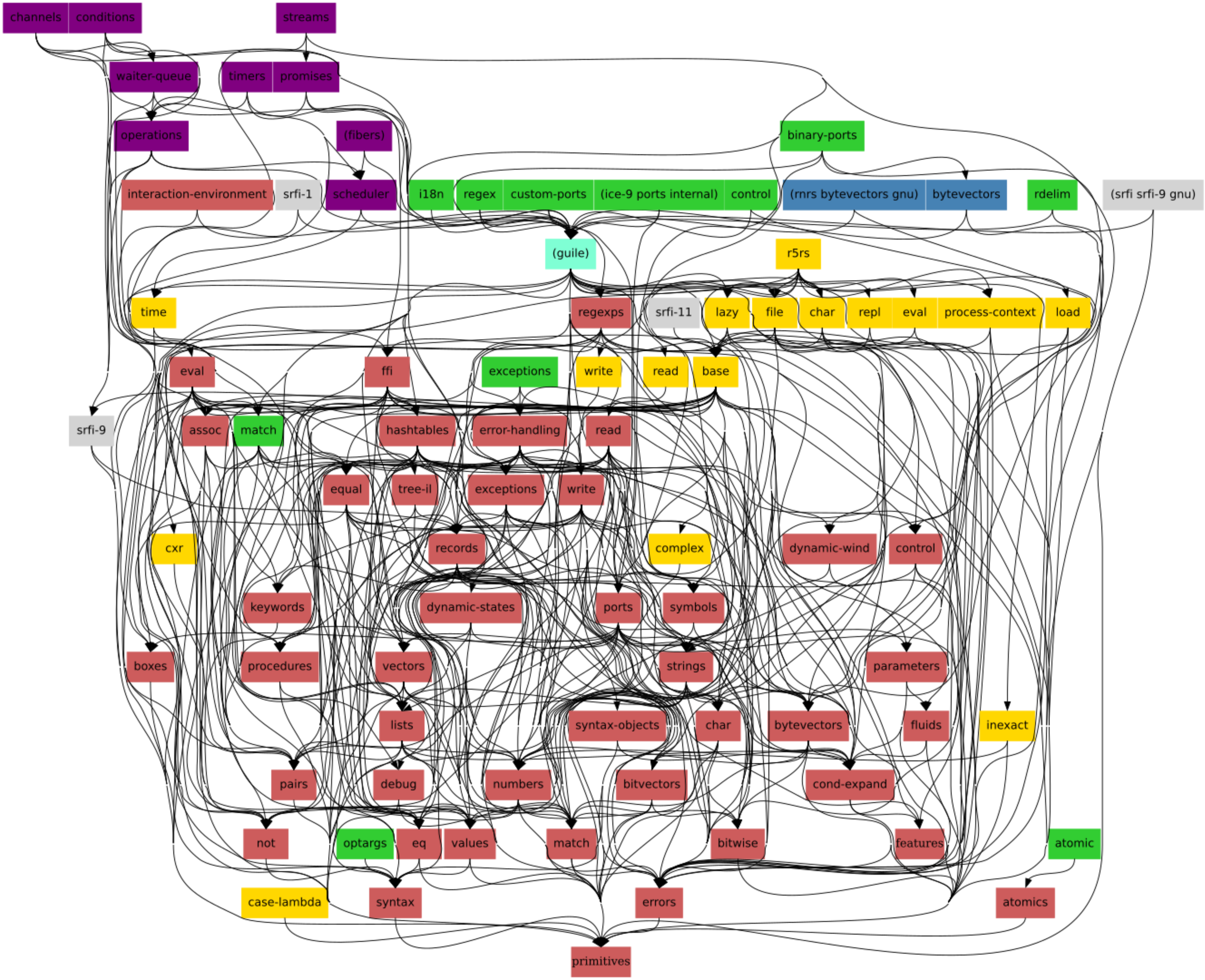
A -> undo

Hoot compiler pipeline

Front-end

- `library-group` makes a big `letrec*` (Hoot)
- `fix-letrec*` sorts SCCs to nested `let` and `fix` (Guile)
- `peval` inlines, eliminates dead code, high-level constant folding, some algebraic reduction (Guile)

`https://wingolog.org/archives/2024/05/22/growing-a-bootie`



Modules can use syntax from their imports

Syntax runs on host, uses imported bindings

Modules live on host, residualized for target

Scenic detour

Middle-end

- Let’s go for a walk

Guile’s middle end

Middle-end spans gap between high-level source code (AST) and low-level machine code

Programs in middle-end expressed in intermediate language

CPS Soup is the language of Guile’s middle-end

How to lower?

High-level:

```
(+ 1 (if x 42 69))
```

Low-level:

```
  cmpi $x, #f
  je L1
  movi $t, 42
  j L2
L1:
  movi $t, 69
L2:
  addi $t, 1
```

How to get from here to there?

1970s

Control-flow graph (CFG)

```
graph := array<block>
block := tuple<preds, succs, insts>
inst  := goto B
       | if x then BT else BF
       | z = const C
       | z = add x, y
       ...
```

```
BB0: if x then BB1 else BB2
BB1: t = const 42; goto BB3
BB2: t = const 69; goto BB3
BB3: t2 = addi t, 1; ret t2
```

Assignment, not definition

1980s

Static single assignment (SSA) CFG

```
graph := array<block>
block := tuple<preds, succs, phis, insts>
phi   := z := φ(x, y, ...)
inst  := z := const C
       | z := add x, y
       ...
BB0: if x then BB1 else BB2
BB1: v0 := const 42; goto BB3
BB2: v1 := const 69; goto BB3
BB3: v2 := φ(v0,v1); v3:=addi t,1; ret v3
```

Phi is phony function: `v2` is `v0` if coming from first predecessor, or `v1` from second predecessor

2003: MLton

Refinement: phi variables are basic block args

```
graph := array<block>
block := tuple<preds, succs, args, insts>
```

Inputs of phis implicitly computed from preds

```
BB0(a0): if a0 then BB1() else BB2()
BB1(): v0 := const 42; BB3(v0)
BB2(): v1 := const 69; BB3(v1)
BB3(v2): v3 := addi v2, 1; ret v3
```

Scope and dominators

```
BB0(a0): if a0 then BB1() else BB2()
BB1(): v0 := const 42; BB3(v0)
BB2(): v1 := const 69; BB3(v1)
BB3(v2): v3 := addi v2, 1; ret v3
```

What vars are “in scope” at BB3? `a0` and `v2`.

Not `v0`; not all paths from BB0 to BB3 define `v0`.

`a0` always defined: its definition *dominates* all uses.

BB0 dominates BB3: All paths to BB3 go through BB0.

Refinement: Control tail

Often nice to know how a block ends (e.g. to compute phi input vars)

```
graph := array<block>
block := tuple<preds, succs, args, insts,
                     control>
control := if v then L1 else L2
         | L(v, ...)
         | switch(v, L1, L2, ...)
         | ret v
```

Refinement: DRY

Block successors directly computable from control

Predecessors graph is inverse of successors graph

```
graph := array<block>
block := tuple<args, insts, control>
```

Can we simplify further?

Basic blocks are annoying

Ceremony about managing insts; array or doubly-linked list?

Nonuniformity: “local” vs “global” transformations

Optimizations transform graph A to graph B; mutability complicates this task

- Desire to keep A in mind while making B
- Bugs because of spooky action at a distance

Basic blocks, phi vars redundant

Blocks: label with args sufficient; “containing” multiple instructions is superfluous

Unify the two ways of naming values: every var is a phi

```
graph := array<block>
block := tuple<args, inst>
inst  := L(expr)
       | if v then L1() else L2()
       ...
expr  := const C
       | add x, y
       ...
```

Arrays annoying

Array of blocks implicitly associates a label with each block

Optimizations add and remove blocks; annoying to have dead array entries

Keep labels as small integers, but use a map instead of an array

`graph := map<label, block>`

This is CPS soup

```
graph := map<label, cont>
cont  := tuple<args, term>
term  := continue to L
           with values from expr
       | if v then L1() else L2()
       ...
expr  := const C
       | add x, y
       ...
```

SSA is CPS

No explicit scope tree: implicit property of control flow

CPS soup in Guile

Compilation unit is intmap of label to cont

```
cont := $kargs names vars term
      | ...
term := $continue k src expr
      | ...
expr := $const C
      | $primcall 'add #f (a b)
      | ...
```

Conventionally, entry point is lowest-numbered label

CPS soup

```
term := $continue k src expr
      | $branch kf kt src op param args
      | $switch kf kt* src arg
      | $prompt k kh src escape? tag
      | $throw src op param args
```

Expressions can have effects, produce values

```
expr := $const val
      | $primcall name param args
      | $values args
      | $call proc args
      | ...
```

Kinds of continuations

Guile functions untyped, can have multiple return values

Error if too few values, possibly truncate too many values, possibly cons as rest arg...

Calling convention: contract between val producer & consumer

- both on call and return side

Continuation of `$call` unlike that of `$const`

The conts

```
cont := $kfun src meta self ktail kentry
      | $kclause arity kbody kalternate
      | $kargs names syms term
      | $kreceive arity kbody
      | $ktail
```

$kclause, $kreceive very similar

Continue to $ktail: return

$call and return (and $throw, $prompt) exit first-order flow graph

High and low

CPS bridges AST (Tree-IL) and target code

High-level: vars in outer functions in scope

Closure conversion between high and low

Low-level: Explicit closure representations; access free vars through closure

Optimizations at all levels

Optimizations before and after lowering

Some exprs only present in one level

Some high-level optimizations can merge functions (higher-order to first-order)

Practicalities

Intmap, intset: Clojure-style persistent functional data structures

Program: `intmap<label,cont>`

Optimization: `program→program`

Identify functions: `(program,label)→intset<label>`

Edges: `intmap<label,intset<label>>`

Compute succs: `(program,label)→edges`

Compute preds: `edges→edges`

Flow analysis

```
A[k] = meet(A[p] for p in preds[k])
         - kill[k] + gen[k]
```

Compute available values at labels:

- A: **`intmap<label,intset<val>>`**
- meet: **`intmap-intersect<intset-intersect>`**
- -, +: **`intset-subtract`**, **`intset-union`**
- kill[k]: values invalidated by cont because of side effects
- gen[k]: values defined at k

Persistent data structures FTW

- meet: `intmap-intersect<intset-intersect>`
- -, +: `intset-subtract`, `intset-union`

Naïve: O(nconts * nvals)

Structure-sharing: O(nconts * log(nvals))

CPS soup: strengths

Relatively uniform, orthogonal

Facilitates functional transformations and analyses, lowering mental load: “I just have to write a function from foo to bar; I can do that”

Encourages global optimizations

Some kinds of bugs prevented by construction (unintended shared mutable state)

We get the SSA optimization literature

CPS soup: weaknesses

Pointer-chasing, indirection through intmaps

Heavier than basic blocks: more control-flow edges

Names bound at continuation only; phi predecessors share a name

Over-linearizes control, relative to sea-of-nodes

Overhead of re-computation of analyses

CPS soup: summary

CPS soup is SSA, distilled

Labels and vars are small integers

Programs map labels to conts

Conts are the smallest labellable unit of code

Conts can have terms that continue to other conts

Back to the middle-end

- Lower to CPS Soup (Guile)
- DCE, simplification, CSE, loop peeling, flow-sensitive folding, contification (Guile)
- Closure optimization, unboxing, LICM, and so on (Guile)

The back-end

- “Tailify” (the true CPS conversion) (Hoot)
- Compute dominator tree (Hoot)
- Apply “Beyond Relooper” to go from CPS Soup CFG to Wasm control-flow (ICFP 2022; Hoot)
- Every CPS Soup variable is a wasm function-local variable

Back of the back-end

Result is a `<wasm>` object: Hoot has a whole Wasm toolchain

- Text parser, serializer
- Binary parser, serializer
- Linker, various transformations
- Validator, virtual machine (!!!)

Result is not browser-specific: same module can run on Hoot VM, in browser, on Node

Hoot status

Full R7RS, except `environment`, and `eval` doesn't have a working macro expander yet

Partial R6RS

Partial Guile

A work in progress, but becoming good, actually

`https://spritely.institute/news/guile-hoot-v050-released.html`

~ fin ~

Wasm is finally here for us!

Hoot’s future: supporting all of Guile, and maybe merging into Guile

Steal Hoot’s wasm toolkit!

Let’s ship Scheme everywhere!

`https://spritely.institute/hoot`