# Scheme + Wasm + GC = MVP

Hoot Scheme-to-Wasm compiler update

11 Oct 2023 – Wasm CG

Andy Wingo

Igalia, S.L. / Spritely Institute

# Agenda

Scheme-to-Wasm/GC update

`https://github.com/WebAssembly/meetings/blob/main/gc/2023/presentations/2023-04-18-wingo-scheme.pdf`

Goal: not marketing, not research, but feedback / feedforward

- Refresher
- Status
- Observations

# Refresher (1/4)

Source language: Scheme (Guile flavor)

Unityped: `(ref eq)`

Immediates: `(ref i31)`

- Fixnums, chars, oddballs (e.g. `#t`, `'()`)

(Most) everything else in a big `rec` group

```
(rec
  (type $heap-object
    (sub
      (struct
        (field $hash (mut i32)))))
  ...)
```

# Refresher (2/4)

Default calling convention:

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

(More precise for known calls)

Callees check incoming arity

Args 3-7 in globals

Args 8-*n* in a table; shrug!

# Refresher (3/4)

CPS-conversion: All calls are tail calls

Stack-allocated return continuations

Non-tail calls push live vars and push return `(ref $kvarargs)`

Return: Pop and `return_call_ref $kvarargs`

Return continuations check arity and pop live vars to restore

# Refresher (4/4)

Non-tail calls **push** live vars and **push** return `(ref $kvarargs)`

*Three stacks*: Raw in linear memory, `(ref eq)` in a table, `(ref $kvarargs)` in a table

Also a stack for dynamic bindings (prompts, fluids, dynwinds)

Suboptimal! But, it gives us delimited continuations

Would happily switch to stack switching

# Status

Current status: MVP.

(Almost) R7RS-small plus Guile extensions (delimited continuations, optargs, etc)

Specific updates:

- Toolchain
- Compiler
- Runtime
- Testing

# Status: Toolchain

Text (Guile datums) and binary reader / writer

Static linker: Provide defs missing in `a.wasm` from `b.wasm`

Transforms: Lower-globals, lower-stringref, symbolify, restackify (in progress)

Partial validator and interpreting VM

Benefit: Experiments (stringref), expressiveness (lower-globals), serendipity (inline-wasm)

# Status: Compiler

Started on fork of Guile, but merged new backend abstraction; now using mainline

Source language: “CPS soup”; SSA-like

“Beyond relooper” FTW

Whole-program rather than separate compilation + dynamic linking

# Status: Runtime

Deployment model: Open-world, Wasm only

Hosts (can) use separate, generic `reflect.wasm` to inspect, access, and create values, call functions

Host facilities: Bignums, weak maps, f64 to/from string, sqrt/sin/cos/etc, string upcase/downcase, host string to/from wtf8

# Status: Testing

Two hosts: JS (V8) and Hoot (Guile)

Hoot VM useful for tighter edit/compile/debug cycle

Stepping, trace, inspect locals, backtrace, dump

Hoot stringref: Guile strings

Complete enough, but not full Wasm. Probably will fill out later

# Observations

This is where I try to be useful

But first…

Thank you!!!

Tail calls landing at the same time

My cup overfloweth

# Observations

On retargetting an existing compiler

On stringref

On globals

On the utility of wat

On integers

On debugging

On the component model

On the future

# On retargetting (1)

SSA-like IR just fine; beyond relooper great

Propagate high-level types all the way through the middle-end

Guile: IR needed slight expansion for explicit returns

WasmGC is a 32-bit target: offsets, fixnums; what would 64-bit look like?

# On retargetting (2)

Calls: Sometimes you know the callee and can have a special calling convention (e.g. that doesn't check arity)

Returns: Harder to know return arity, but sometimes possible; however return stack is generic (`ref $kvarargs`)

Had to add special pass to pessimize returns by trampolining through `$kvarargs`

Stack switching would be a big win

# On retargetting (3)

Wasm stack allocation is not like register allocation: unlimited in number, heterogeneous in type

Hoot punts: one local per intermediate value. No stack data flow between IR ops

Restackify pass in the works

# On retargetting (3)

Non-nullable unitype + explicit intermediate values: joins need special treatment

Currently: Eagerly initialize `(ref eq)` joins to `(ref.i31 (i32.const 0))`

Restackify will mitigate: block result values

# On retargetting (4)

## Multi-byte access to `(array i8)` excruciating

```
(('s16-ref ann obj ptr idx)
 `(,(local.get ptr)
   ,(local.get idx)
   (i32.wrap_i64)
   (array.get_u $raw-bytevector)
   ,(local.get ptr)
   ,(local.get idx)
   (i32.wrap_i64)
   (i32.const 1)
   (i32.add)
   (array.get_s $raw-bytevector)
   (i32.const 8)
   (i32.shl)
   (i32.or)
   (i64.extend_i32_s)))
```

`(i32.load16_s $bytes)`?

# On stringref

```
(sub $heap-object
  (struct $string
    (field $hash (mut i32))
    (field $str (ref string))))
```

`string.const` for literals.

`string.as_iter` + `stringview_iter.advance` for `string-length`.

Same plus `stringview_iter.next` for `string-ref`.

Hash: `stringview_iter.next` loop.

Lots of `string.const` for debugging

# On stringref (2)

Textual I/O: “port” with WTF-8 buffer

String ports are string builders

Per-char read (`fgetc`, but for codepoints) currently uses stringref, but probably should decode bytes directly using DFA

# On stringref (3)

`string-copy` uses `stringview_iter.slice`

String comparison uses `string.compare`; not just Java :P

`string->utf8` uses `string.measure_wtf8`, `string.encode_wtf8_array`; reverse is `string.new_lossy_utf8_array`

# On stringref (4)

To ship, we must remove stringref

Solution: Pass to replace with `(array i8)` WTF-8 internally

Wrap outgoing: `(func (param (ref $wtf8)) (result (ref extern)))`

Wrap incoming: `(func (param (ref extern)) (result (ref $wtf8)))`

If we have to choose: UTF-8.

*But we would rather use the host's strings*

```
function wtf8_to_string(wtf8) {
    let { as_iter, iter_next } = wtf8_helper.exports;
    let codepoints = [];
    let iter = as_iter(wtf8);
    for (let cp = iter_next(iter); cp != -1; cp = iter_next(it
        codepoints.push(cp);
    return String.fromCodePoint(...codepoints);
}

function string_to_wtf8(str) {
    let { string_builder, builder_push_codepoint, finish_build
        wtf8_helper.exports;
    let builder = string_builder()
    for (let cp of str)
        builder_push_codepoint(builder, cp.codePointAt(0));
    return finish_builder(builder);
}
```

# On stringref (6)

Lowering to `(array i8)` not quite transparent

Imports: WTF-8 from other Scheme wasm modules, externref from host; similar with exports

`string.const` values to `array.new_data` globals; not const :-((

Can’t be in elem sections

# On stringref (7)

What about JS Builtin Strings? Can help, but less appropriate for use case.

Missing: `encodeWtf8Array` / `toWtf8Array`, UTF-8 variants (WTF-8, lossy), measuring WTF-8 size

Literals: Either `array.new_data` + `fromWtf8Array` or generate JavaScript; not const

BYO iterators. To be fair, `(array i8)` is same

*Hoot is not JS-specific in any way*. See e.g. Hoot VM

# On stringref (8)

Hoot: Stringref is a toolchain concept

Can leave as stringref; default is to lower

JS Builtin Strings may ameliorate boundary costs, but not as string repr: unlikely to be present on non-JS hosts

# On stringref (9)

**The system would be better with abstract host strings**

DOM calls will never (?) be fast without stringref AFAICS

Smaller modules, fewer copies, less host code, better portability

Better for users than `(array i8)`

</>

# On globals

Compound literals best implemented as globals

Globals have to have const initializers

Some literals not const: `array.new_data`, hash-consed literals from aux compilation units

Solution: Just emit a non-const init; fix up in post-pass, synthesize `start`

Note! Non-constness propagates mutability, nullability to other literals, code uses; `ref.as_non_null`

# On wat

About 3.5 kSLOC of wat in low-level stdlib; good experience

Folded form is a blessing to humans

Hoot: A mechanical optimization of low-level stdlib would be fruitful

Very nice for debugging, communication

# On wat (2)

Unexpected benefit: `%inline-wasm`

```
(define (bitvector-length bv)
  (unless (bitvector? bv)
    (error "expected bitvector" bv))
  (%inline-wasm
   '(func (param $bv (ref eq)) (result (ref eq))
      (ref.i31
       (i32.shl
        (struct.get $bitvector $len
                    (ref.cast $bitvector (local.get $bv)))
        (i32.const 1))))
   bv))
```

Inline wasm, rewrite params to refer to provided locals

Only for functions ATM. Coming: FFI imports, global decls

# On integers

Native: Check fixnum by checking low bits

Wasm: Check fixnum by `ref.test i31` then `ref.cast i31` (unless you can emit `br_on_cast_fail`), then `i31.get_s` and check low bit is 0

Would be nice to be able to subtype and partition `i31` space

# On integers (2)

Native: Add const to fixnum, branch on overflow

Wasm: Uhhhhhhh, call a function

Can has `i31.br_on_add_fail`? Or anything, really.

# On integers (3)

Native: If adding fixnums fails, make a bignum

Wasm: Uhhhhhhh, call a function

Thinking the thinkable: Built-in `bignum` with `int` supertype, and `int.add_generic`? Would have to take `(ref i31)` subtyping into account.

feedforward, *n.*: just another word for puke

# On debugging

State of the art is still printf debugging, though with `string.const`

Explicit stack means we can print a backtrace, but all in a glob: what data corresponds to what frame?

Return continuations not `eq`; cannot be associated with side tables

Mitigation: Use the host; NYI

Can has funcref side tables, somehow? Associated with code, not closure

# On the component model

Haven’t had the budget yet to think about it

Impression: Component model is for linear memory modules. Cue “prove me wrong” guy meme

I would like to learn; please correct my ignorance!

# On the future

Next milestone: Spring 2024

- Full Guile language, including modules
- FFI imports
- Optimization
- Fibers
- Stack switching?
- OCapN / Goblins

# Thanks for listening!

Summary: GC stands for Great Communitygroup

`https://spritely.institute/news/scheme-wireworld-in-browser.html`

Please grab me afterwards to chat!

```
(visit-links
 "gitlab.com/spritely/guile-hoot"
 "wingolog.org"
 "wingo@igalia.com"
 "igalia.com"
 "spritely.institute"
 "mastodon.social/@wingo")
```