GC — What next?

Andreas Rossberg

2023/10/12

Many small improvements possible,
focussing on the big ones in this presentation

No-GC Profile

keep Wasm viable for all our clients, such as embedded

Shared GC Objects

allow sharing references to objects and functions between threads

motivation: make GC features interoperate with threads

Shared GC Objects

essentially, requires **shared** attribute on everything:

```
(type $f (shared func (param i32) (result i32))
(type $s (shared struct (field i64 i64))
(global $g (shared mut i32))
(table $t (shared 20 sharedfuncref))    ;; can only contain shared refs
(func $f (type $f) …)                   ;; can only access shared shared entities
```

basic blueprint can be found in Appendix of our OOSPLA'19 memory model paper

gives rise to concurrent GC in engines

Type Imports

allow type definitions to be im/exported from modules

motivation: efficient system types (like string), components, libraries

Type Imports

Risk of feature creep

Limit to existing type definitions and fixed representations for now

```
(type $t (struct …))
(export "t" (type $t))

(import "M" "u" (type $t (sub any)))   ;; bound determines representation
```

Must also allow type definitions like (type $t i31)

Value type definitions are a completely different beast and require deeper changes

Type Parameters

allow type definitions to be im/exported from modules

motivation: efficient system types (like string), components, libraries

Type Parameters

(**type** $pair (typeparam $x) (struct (field (ref $x) (ref $x)))

(**func** $fst (typeparam $x) (param (ref $pair $x)) (result (ref $x)) …)

Prevent hidden cost!

…avoid lock-in into path that would require type-passing for all generic functions

…instead, need RTTs to be explicit to construct or cast generic types

Need to be designed together with type imports

…both are forms of parameterisation and should have a coherent semantics

Self Types

introduce special form of recursive self type that is covariant

motivation: avoiding casts for closures and method overrides

Self Types

```
class C {
    var a : Int
    method f() : Int { return a }
}
```

```
class D <: C {
    var b : Int
    override f() : Int { return a + b }
}
```

```
type $C-inst     = struct (ref $C-vtable) i64
type $C-vtable = struct (ref $C-f)
type $C-f        = func (ref $C-inst) → i64

type $D-inst     = struct (ref $D-vtable) i64 i64
type $D-vtable = struct (ref $D-f)
type $D-f        = func (ref $C-inst) → i64
```

```
(func $D-f (type $D-f)
    (local $this (ref $D-inst))
    (ref.cast $D-inst (local.get 0))
    (local.set $this)
    …
)
```

Self Types

```
class C {
    var a : Int
    method f() : Int { return a }
}
```

```
class D <: C {
    var b : Int
    override f() : Int { return a + b }
}
```

```
type $C-inst     = struct (ref $C-vtable) i64
type $C-vtable = struct (ref $C-f)
type $C-f        = func (ref (self $C-inst)) → i64

type $D-inst     = struct (ref $D-vtable) i64 i64
type $D-vtable = struct (ref $D-f)
type $D-f        = func (ref (self $D-inst)) → i64
```

```
(func $D-f (type $D-f)
    (local $this (ref $D-inst))
    (ref.unpack $D-inst (local.get 0))
    (local.set $this)
    …
)
```

Nested Aggregates

allow flat nesting of structures and arrays (as in C)

motivation: avoid indirections (e.g., array objects), improve locality

Nested Aggregates

(**type** $point (struct (field f64 f64 f64)))

(**type** $pointlist (array (type $point)))

(**type** $JArray-inst (struct (field (ref $JArray-vtable) (type $JArray-contents))))

(**type** $JArray-contents (array eqref))

need interior references to access nested aggregate

(struct.inref $JArray-contents 1) : (ref $JArray-inst) → (inref $JArray-contents)

different type from regular references, may have fat pointer representation

Descriptors, Static Fields

enrich the notion of RTT with custom fields (a.k.a. static fields)

motivation: avoid additional fields

Weak References & Finalisation???

everybody wants it

but nobody has a good idea how to do it… :(

problem: too much diversity and mutually incompatible semantic choices

Status

Reduction and validation

Decoding and parsing in the works

Can handle Wasm 2.0 minus SIMD

Most of Wasm 3.0 prototyped as well

Numeric primitives still manually implemented

Prose interpreter passes 100% of test suite!