Garbage Collection for Wasm

Proposal update

Andreas Rossberg
Dfinity

Motivation

Efficient support for high-level languages

… fast execution, small executables

… instant access to industrial-strength GCs

Efficient interop with embedder

… avoid inter-heap GC problem

Non-goal: seamless interlanguage interop

Design Principles

Simple and lightweight

As low-level as possible

Agnostic to language or paradigm

Pay as you go; no dependencies

Simple types, checked casts as escape hatch

MVP Proposal

Plain struct and array definitions

Reference types for those

Instructions for allocation and access

Explicit runtime types and checked down casts

Scalar and function references

Possibly Post-MVP

Nested structs & arrays, inner references

Dynamically-sized structs

Type Parameters

Header fields or meta objects

Means for eliminating more casts

Abstract and nominal types

Closures

Thread-shared references

Weak refs & finalisation

Type Definitions

datatype ::= <structtype> | <arraytype>
structtype ::= **struct** <fieldtype>*
arraytype ::= **array** <fieldtype>

fieldtype ::= <mutability> <storagetype>
storagetype ::= <valtype> | **i8** | **i16**
mutability ::= **const** | **var**

In the future...

datatype ::= <structtype> | <arraytype>

structtype ::= **struct** <fieldtype>*

arraytype ::= **array** <fieldtype> <u32>$^{?}$

fieldtype ::= <mutability> <storagetype>

storagetype ::= <valtype> | **i8** | **i16** | <datatype>

mutability ::= **const** | **var**

Reference Types

reftype ::= **anyref** | **funcref** | **eqref**

| **ref** $t | **optref** $t

| **rtt** $t | **i31ref**

Instructions - Functions

ref.func $f : [] → [ref $t-of-f]

call_ref : [optref $t, t*] → [t’*]

return_call_ref : [optref $t, t*] → [t’*]

func.bind…?

Instructions - Structs

struct.new $t : [t*] → [ref $t]

struct.get $t $x : [optref $t] → [t]

struct.set $t $x : [optref $t, t] → []

Instructions - Arrays

array.new $t : [t, i32] → [ref $t]

array.get $t : [optref $t, i32] → [t]

array.set $t : [optref $t, i32, t] → []

array.len $t : [optref $t] → [i32]

Example - Classes

```
class C {
  int x;

  void f(int i);
  int g();
}
```

```
class D extends C {
  double y;

  override int g();
  int h();
}
```

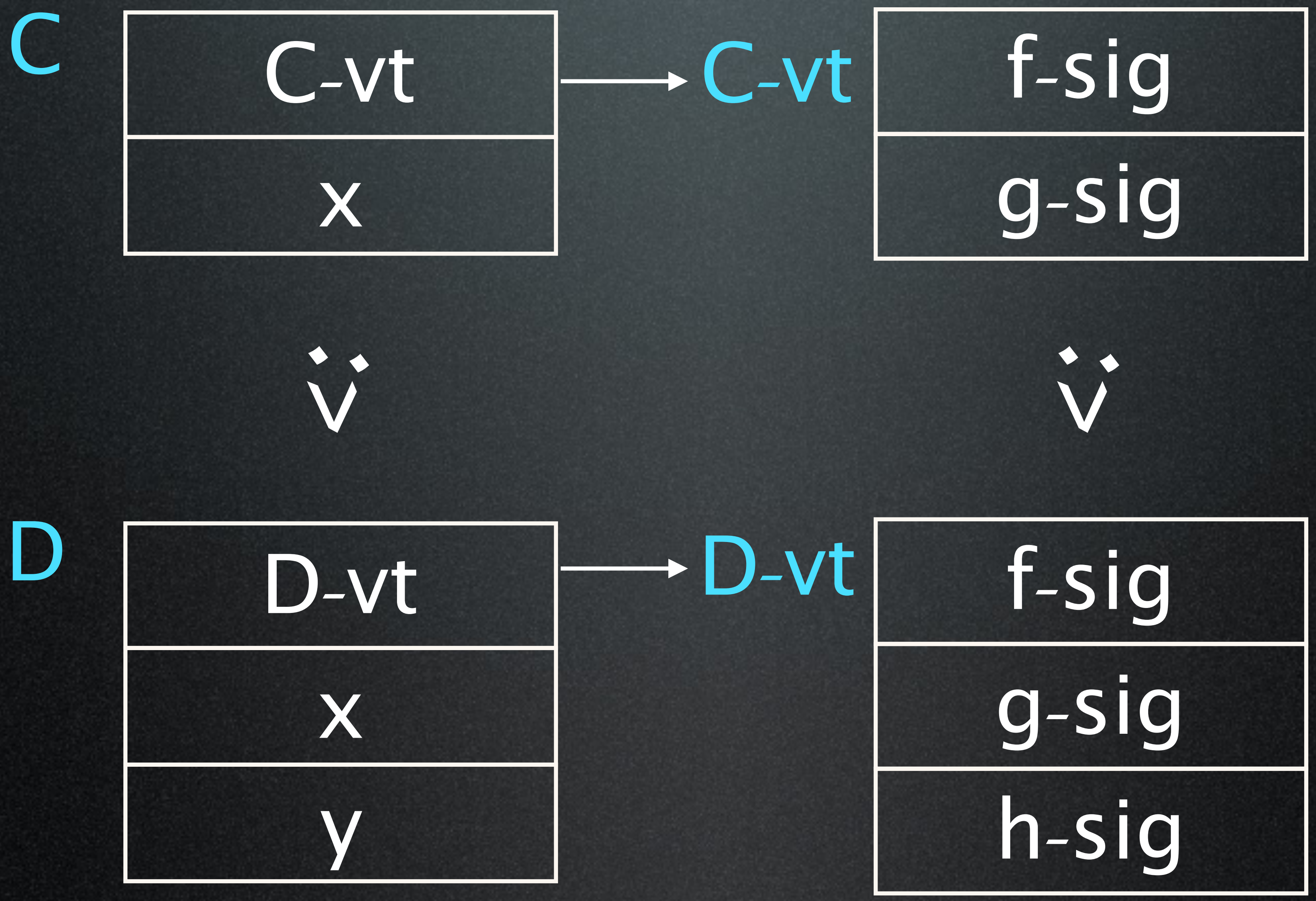
Example - Classes

```
(type $f-sig (func (param (ref $C)) (param i32)))
(type $g-sig (func (param (ref $C)) (result i32)))
(type $h-sig (func (param (ref $D)) (result i32)))

(type $C (struct (ref $C-vt) (mut i32))
(type $C-vt (struct (ref $f-sig) (ref $g-sig)))

(type $D (struct (ref $D-vt) (mut i32) (mut f64)))
(type $D-vt (struct (ref $f-sig) (ref $g-sig) (ref $h-sig)))
```

Example - Classes



C

C-vt

x

C-vt

f-sig

g-sig

D

D-vt

x

y

D-vt

f-sig

g-sig

h-sig

Example - Classes

```
(type $f-sig (func (param (ref $C)) (param i32)))
(type $g-sig (func (param (ref $C)) (result i32)))
(type $h-sig (func (param (ref $D)) (result i32)))

(type $C (struct (ref $C-vt) (mut i32))
(type $C-vt (struct (ref $f-sig) (ref $g-sig)))

(type $D (struct (ref $D-vt) (mut i32) (mut f64)))
(type $D-vt (struct (ref $f-sig) (ref $g-sig) (ref $h-sig)))
```

Example - Classes

```
(func $D.g (param $Cthis (ref $C))
   (local $this (ref $D))
   (local.get $Cthis)
   (ref.cast $C $D (global.get $D-rtt))
   (local.set $this)
   ...
)
```

Instructions - Casts

ref.test $t $t′ : [optref $t, rtt $t′] → [i32]

ref.cast $t $t′ : [optref $t, rtt $t′] → [ref $t′]

br_on_cast $l $t $t′ : [optref $t, rtt $t′] → [optref $t]
(where $l : [ref $t′])

Instructions - RTTs

rtt.new $t $t′ : [rtt $t] → [rtt $t′]

rtt.anyref : [] → [rtt anyref]

struct.new_with_rtt $t $t′ : [t*, rtt $t′] → [ref $t]
array.new_with_rtt $t $t′ : [t, rtt $t′] → [ref $t]

Instructions - Optref

ref.as_nonnull : [optref $t] → [ref $t]

br_on_null : [optref $t] → [ref $t]

Instructions - Equality

ref.eq : [eqref, eqref] → [i32]

Unboxed Scalars

Many languages rely on a uniform representation

… every value is word-sized

… 1st-class polymorphism, dynamic typing, etc.

Still want to avoid boxing for small scalars

Usual trick: pointer tagging

Need equivalent for Wasm references

Unboxed Scalars: Goals

Type of scalars that is subtype of anyref

Guaranteed to be unboxed on all platforms
… no hidden branches, no hidden allocations

Engines can implement it with pointer tagging

Unboxed Scalars: Solution

Add a type i31ref

Conceptually, a reference to an integer

Practically, a tagged unboxed integer

Largest integer range that can be unboxed on all Wasm platforms

While staying representation-compatible with anyref

Instructions - Scalars

ref.i31 : [i32] → [i31ref]

ref.get_i31_u : [i32ref] → [i32]

ref.get_i31_s : [i32ref] → [i32]

Type Imports/Exports

exportdesc ::= … | **type** $t

importdesc ::= … | **type** <typedesc>
typedesc ::= **sub** $t | **eq** $t

Open Questions

Function bind

Details of RTT introduction

Casts over function references

“Syntactic” woes: <typeidx> vs <reftype>

JS API