Low-level GC
Wasm proposal update

Andreas Rossberg
Dfinity

why?

efficient support for high-level languages

… fast execution, fast loading, small binaries

… instant access to industrial-strength GCs

efficient interop with embedder

… avoid inter-heap GC problem

non-goal: seamless inter-language interop

… key to success of Wasm at large

… deferred to higher layers, e.g. interface types

avoid old problems

language/paradigm-specific constructs

heavyweight object & class system

heavyweight reified generics

unsound or unmodular type system

goals

stay agnostic to languages or paradigms

… no bias: avoid specialised constructs

… diversity: OO, FP, simple dynamic languages

define low-level data layout, not objects

… only high-level enough to be safe & portable

simple and lightweight

… avoid complicated or expensive constructs

… pay as you go

… casts as escape hatch

design principles

low-level, assume producer does most work

… defining runtime data structures (e.g. vtables)

… optimising & canonicalising data layout

… limited by portability requirement

explicit, provide predictable cost model

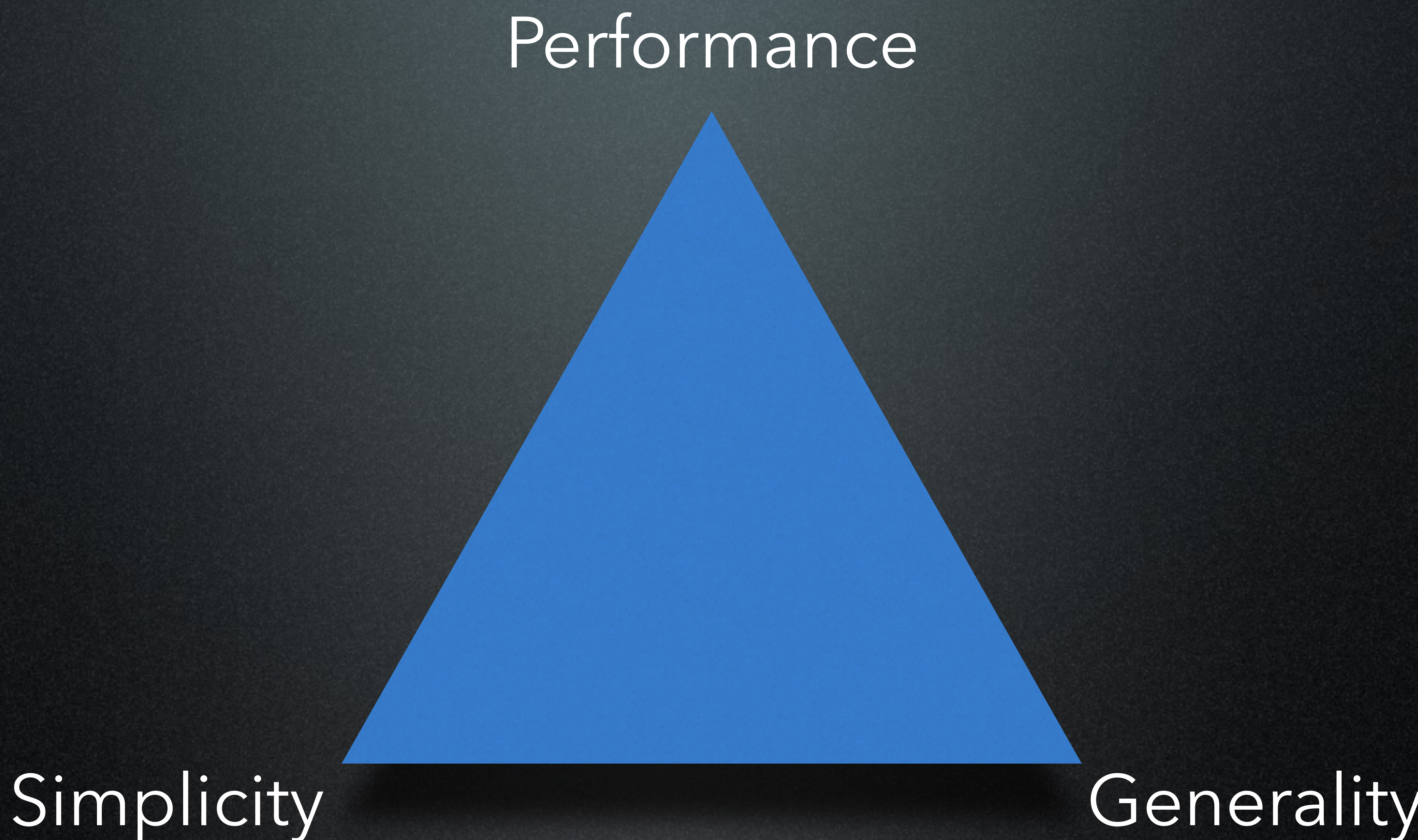
… no hidden runtime type checks

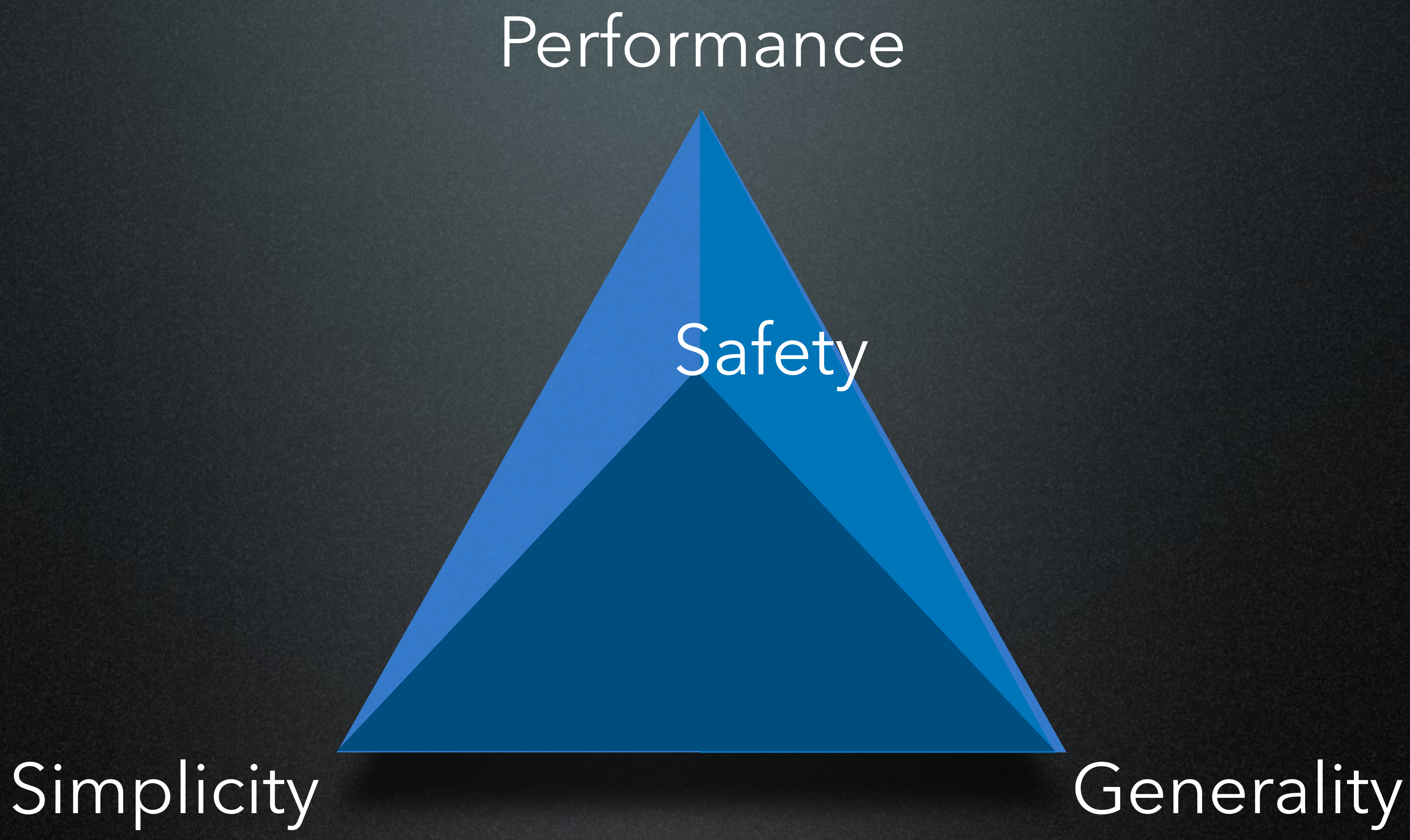
… no hidden allocations

modular, match Wasm's strong modularity

… avoid requiring specific tool chain support

… merging (& splitting) of modules


Performance
Simplicity
Generality


Performance
Safety
Simplicity
Generality

m.v.p.

focus on minimal feature set for now

… basic structs, arrays, scalars

… moderate encoding overhead is fine for now

defer any feature that isn't essential or can be worked around

… flat struct/array nesting, sophisticated representations, …

… start with what's easily supported in existing engines

ship & iterate (a.k.a. shiterate)

… keep design simple but open

… avoid premature optimisation or over-engineering

… we don't have all the answers yet (nor all requirements)

how?

tuples – heterogeneous, statically indexed

arrays – homogeneous, dynamically indexed

scalars – unboxed ints

casts – checked, escape hatch, explicit type tags datatype ::= **func** <valtype>* → <valtype>*

| **struct** <fieldtype>*

| **array** <fieldtype>

| **scalar**

fieldtype ::= **mut**$^?$ <storetype>

storetype ::= **i8** | **i16** | <valtype>

valtype ::= **i32** | **i64** | **f32** | **f64**

| **anyref** | **opt**$^?$**ref** <datatype>

In the future...

datatype ::= **func** <valtype>* → <valtype>*
| **struct** <fieldtype>*
| **array** <fieldtype> <u32>?
| **scalar**

fieldtype ::= **mut**? <storetype>
storetype ::= **i8** | **i16** | <valtype>
valtype ::= **i32** | **i64** | **f32** | **f64**
| **anyref** | **opt**?**ref** <datatype>

In the future...

datatype ::= **func** <valtype>* → <valtype>*

| **struct** <fieldtype>*

| **array** <fieldtype> <u32>?

| **scalar**

fieldtype ::= **mut**? <storetype> | <datatype>

storetype ::= **i8** | **i16** | <valtype>

valtype ::= **i32** | **i64** | **f32** | **f64**

| **anyref** | **opt**?**ref** <datatype>

In the future…

datatype ::= **func** <valtype>* → <valtype>*
| **struct** <fieldtype>*
| **array** <fieldtype> <u32>$^?$
| **scalar**
| **variant** <fieldtype>*

fieldtype ::= **mut**$^?$ <storetype> | <datatype>
storetype ::= **i8** | **i16** | <valtype>
valtype ::= **i32** | **i64** | **f32** | **f64**
| **anyref** | **opt**$^?$**ref** <datatype>

In the future...

End point is a composable C-style type algebra

Allows flattened structs of arrays of structs, etc.
… important for efficient access & data locality

More flexible than merging structs and arrays

Similar to Typed Objects on JS side

But leaving flattened nesting for post-MVP
… at the cost of an indirection

Instructions - Structs

struct.new $t : [$t_i$*] → [ref $t]

struct.get $i : [optref $t] → [$t_i$]

struct.set $i : [optref $t, t_i] → []

Instructions - Arrays

array.new $t : [t′, i32] → [ref $t]

array.get : [optref $t, i32] → [t′]

array.set : [optref $t, i32, t′] → []

array.len : [optref $t] → [i32]

Instructions - Scalars

scalar.new : [i32] → [ref scalar]

scalar.get_u : [optref scalar] → [i32]
scalar.get_s : [optref scalar] → [i32]

scalar.is : [anyref] → [i32]
scalar.as : [anyref] → [ref scalar]

(guaranteed unboxed, via pointer tagging)

Example - Classes

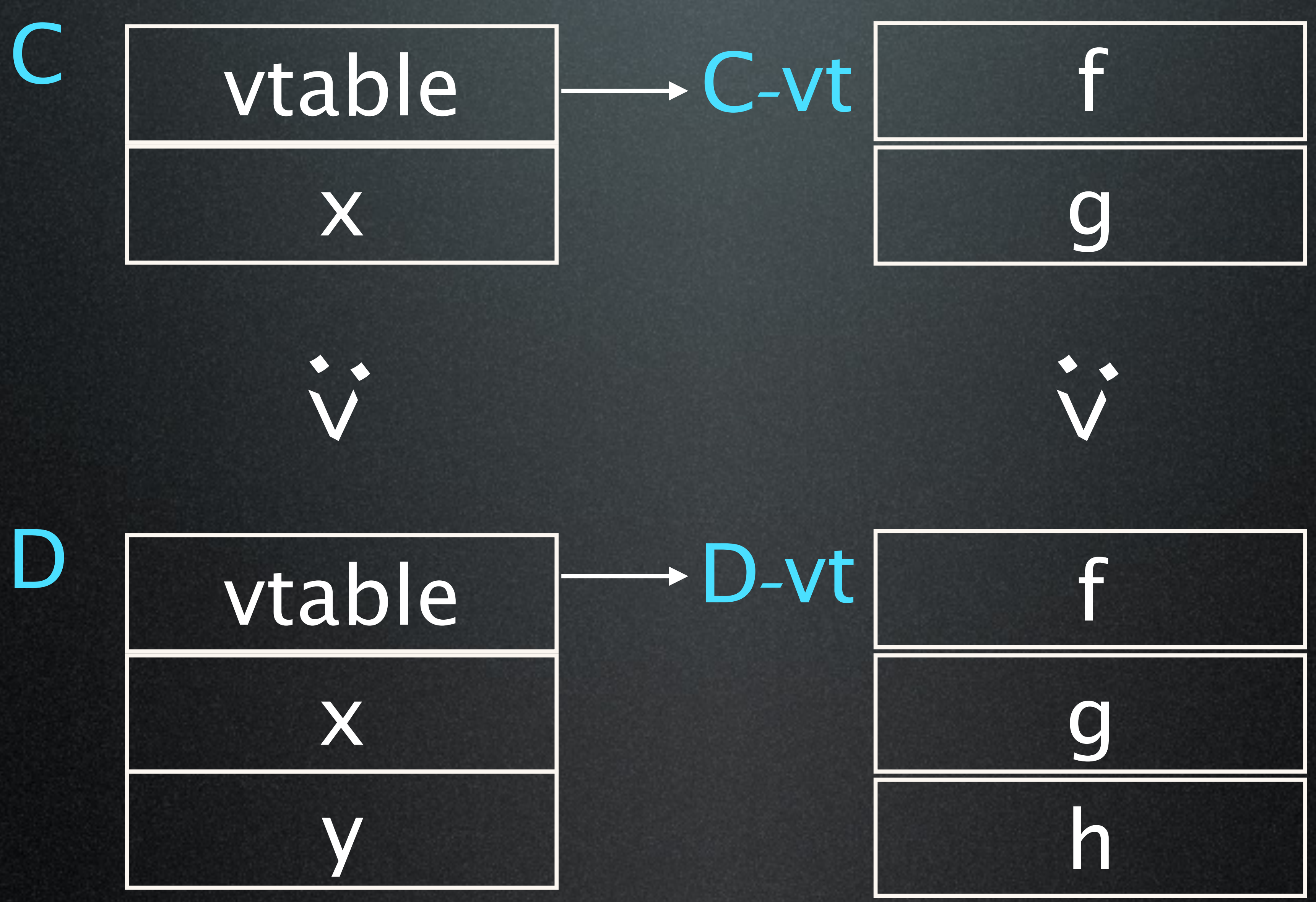
```
class C {
  int x;

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

```
class D extends C {
  double y;

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

Example - Classes



Example - Classes

type $f = **func** (ref $C) i32 → i32
type $g = **func** (ref $C) → i32
type $h = **func** (ref $D) → i32

type $C = **struct** (ref $Cvt) (mut i32)
type $D = **struct** (ref $Dvt) (mut i32) (mut f64)

type $Cvt = **struct** (ref $f) (ref $g)
type $Dvt = **struct** (ref $f) (ref $g) (ref $h)

gc subgroup?