Nominal vs Structural Types

Andreas Rossberg

Dfinity

types in wasm

define data layout, not source-language types

… e.g., structure of vtable vs declaration of class

exist to aid engine, not programmer or compiler writer

… needed to validate and generate efficient code

and for encapsulation at module boundaries

… pass own data to untrusted modules safely

need to be sound

… otherwise they would not be useful for the above

lowering

all source types need to be lowered to wasm types

all wasm types wasm need to be defined in a module

structural vs nominal

structural: types match when they have the same form

… e.g., arrays, tuples, pointers, functions

… also, generic types

… a type definition is an alias for a type

nominal: types match when they have the same name

… e.g., classes, algebraic data types, abstract types

… a type definition creates a new type

almost all languages have some of both

few are fully structural (e.g., TypeScript)

what about wasm?

structural?

nominal?

both?

note: function types are structural

lowering structural into nominal

compiler needs to dedupe type definitions

but can only do that per module

what about types shared across modules?

Example

```
;; func hash(blob : byte[]) : int64
```

```
(module
  (type $blob (array i8))
  (func (export “hash”) (param (ref $blob)) (result i64) ...)
)
```

Example

```
;; func zip(blob : byte[]) : byte[]
```

```
(module
  (type $blob (array i8))
  (func (export “zip”) (param (ref $blob)) (result (ref $blob)) …)
)
```

Example

```
(module
  (type $blob (array i8))
  (func $hash (import "H" "hash") (param (ref $blob)) (result i64))
  (func $zip (import "Z" "zip") (param (ref $blob)) (result (ref $blob)))

  (func (export "foo")
    ...
    (local.get $data)
    (call $zip)
    (call $hash)
    ...
  )
)
```

if array is a structural type,
this works as expected

Example (nominal)

```
;; func hash(blob : byte[]) : int64
```

```
(module
  (type $blob (array i8))
  (func (export “hash”) (param (ref $blob)) (result i64) …)
)
```

Example (nominal)

```
;; func hash(blob : byte[]) : int64
```

```
(module
  (type $blob (export “blob”) (array i8))
  (func (export “hash”) (param (ref $blob)) (result i64) …)
)
```

Example (nominal)

```
;; func zip(blob : byte[]) : byte[]
```

```
(module
  (type $blob (export “blob”) (array i8))
  (func (export “zip”) (param (ref $blob)) (result (ref $blob)) …)
)
```

Example (nominal)

```
(module
  (type $hblob (import "H" "blob") (array i8))
  (func $hash (import "H" "hash") (param (ref $hblob)) (result i64))
  (type $zblob (import "Z" "blob") (array i8))
  (func $zip (import "Z" "zip") (param (ref $zblob)) (result (ref $zblob)))

  (func (export "foo")
    ...
    (local.get $data)
    (call $zip)
    (call $hash)
    ...
  )
)
```

import inversion

type exports need to turn into type imports

Example (nominal)

```
;; func hash(blob : byte[]) : int64
```

```
(module
  (type $blob (export “blob”) (array i8))
  (func (export “hash”) (param (ref $blob)) (result i64) …)
)
```

Example (nominal)

```
;; func hash(blob : byte[]) : int64
```

```
(module
  (type $blob (import "?" "?") (array i8))
  (func (export "hash") (param (ref $blob)) (result i64) ...)
)
```

aside: linking nominal type imports

Q: how are nominal type imports checked?

```
(module $A
  (type (import "B" "t") (struct i32 i64 i32))
)

(module $B
  (type (export "t") (struct i32 i64 i32))
)
```

A: structurally

aside: linking nominal type imports

Q: how are nominal type imports checked?

```
(module $A
  (type (import “B” “t”))
)
```

Cannot compile or validate before instantiation!

```
(module $B
  (type (export “t”) (struct i32 i64 i32))
)
```

A: structurally

aside aside: why this is not a problem in Java

because Java effectively is unsound

every object access has a runtime type check

optimised with dynamic VM machinery

contrary to Wasm's goals

Example (nominal)

```
;; func hash(blob : byte[]) : int64
```

```
(module
  (type $blob (import "?" "?") (array i8))
  (func (export "hash") (param (ref $blob)) (result i64) ...)
)
```

who owns these types?

type sharing

1. either by centralisation
   ... shared types imported from a shared module

2. or by parameterisation
   ... shared types imported from each client module

type sharing by centralisation

there is a central module

from which *any* type can be imported

and that dedupes structurally equivalent types

Example (nominal, centralised)

```
;; func hash(blob : byte[]) : int64
```

```
(module
  (type $blob (import “types” “array_i8”) (array i8))
  (func (export “hash”) (param (ref $blob)) (result i64) ...)
)
```

Example (nominal, centralised)

```
;; func zip(blob : byte[]) : byte[]
```

```
(module
  (type $blob (import “types” “array_i8”) (array i8))
  (func (export “zip”) (param (ref $blob)) (result (ref $blob)) ...)
)
```

Example (nominal, centralised)

```
(module
  (type $blob (import “types” “array_i8”) (array i8))
  (func $hash (import “H” “hash”) (param (ref $blob)) (result i64))
  (func $zip (import “Z” “zip”) (param (ref $blob)) (result (ref $blob)))

  (func (export “foo”)
    …
    (local.get $data)
    (call $zip)
    (call $hash)
    …
  )
)
```

type sharing by centralisation

there exists a central module
from which *any* type can be imported
and that dedupes structurally equivalent types

…so it is magic in multiple ways

…needs to be accessible across packages & langs

…hence, needs to be built-in into the platform

type sharing by centralisation

we just re-implemented structural types!

but with the most clumsy interface

certain tools would need to reimplement the same (e.g., static linkers, in order to dedupe types)

overall, more complexity and less coherence

type sharing by parameterisation

each importing module provides the types

which it may itself have imported

some higher-up module defines the types

problem: not currently expressible in Wasm, requires module parameters, or a way for binding imports of an imported module

Example (nominal, parameterised)

```
;; func hash(blob : byte[]) : int64
```

```
(module
  (type $blob (import "" "blob") (array i8))
  (func (export "hash") (param (ref $blob)) (result i64) ...)
)
```

Example (nominal, parameterised)

```
;; func zip(blob : byte[]) : byte[]
```

```
(module
  (type $blob (import "" "blob") (array i8))
  (func (export "zip") (param (ref $blob)) (result (ref $blob)) ...)
)
```

Example (nominal, parameterised)

```
(module
  (type $blob (import "" "blob") (array i8))
  (export "H" "blob" (type $blob))
  (func $hash (import "H" "hash") (param (ref $blob)) (result i64))
  (export "Z" "blob" (type $blob))
  (func $zip (import "Z" "zip") (param (ref $blob)) (result (ref $blob)))

  (func (export "foo")
    …
   (local.get $data)
   (call $zip)
   (call $hash)
    …
  )
)
```

type sharing by parameterisation

each importing module provides the types
which it may itself have imported
some higher-up module defines the types

…explicit type sharing is incredibly hard to deal with in practice!
(reoccurring theme in the module literature)

…every module accumulates type imports from lower layers

…can never partially abstract
(unless a group of types becomes completely private)

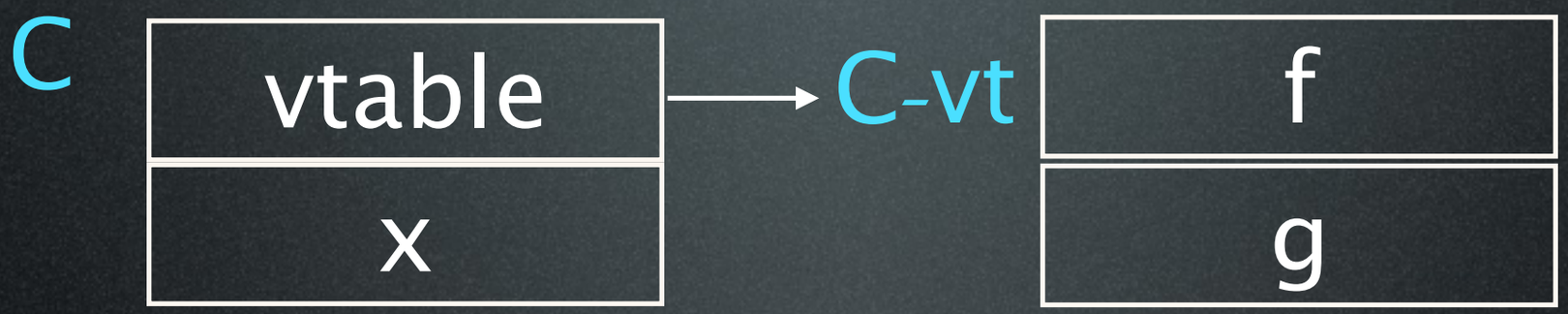
…type imports grow exponential in depth of dependency graph
[cf. Pierce & Harper, ATTAPL Ch. 8]

…every import creates a fresh instance of the module (no sharing of state!)
(unless we want to ML-style “applicative functor” semantics, meta-structural)

Example - Classes

```
class C {
  int x;
  void f(int i);
  int g();
}
```

Example - Classes



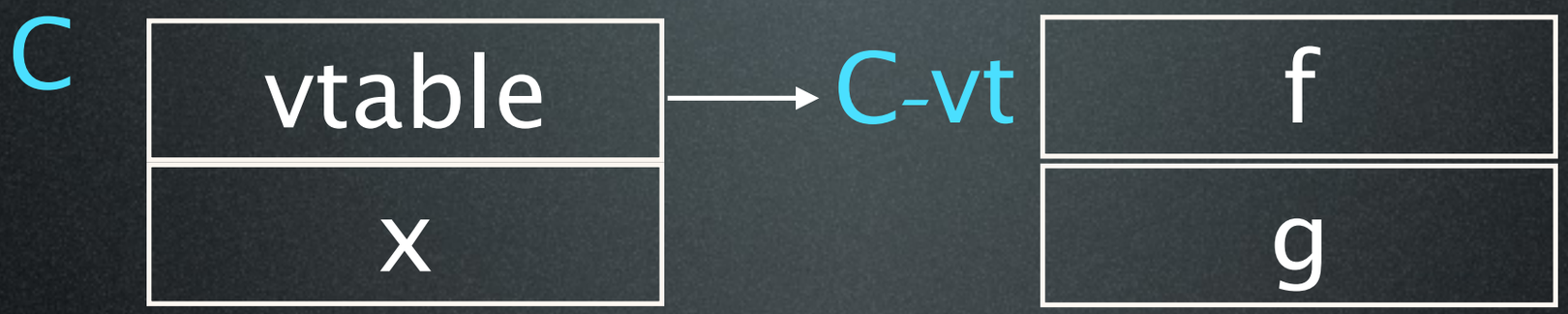
Example - Classes

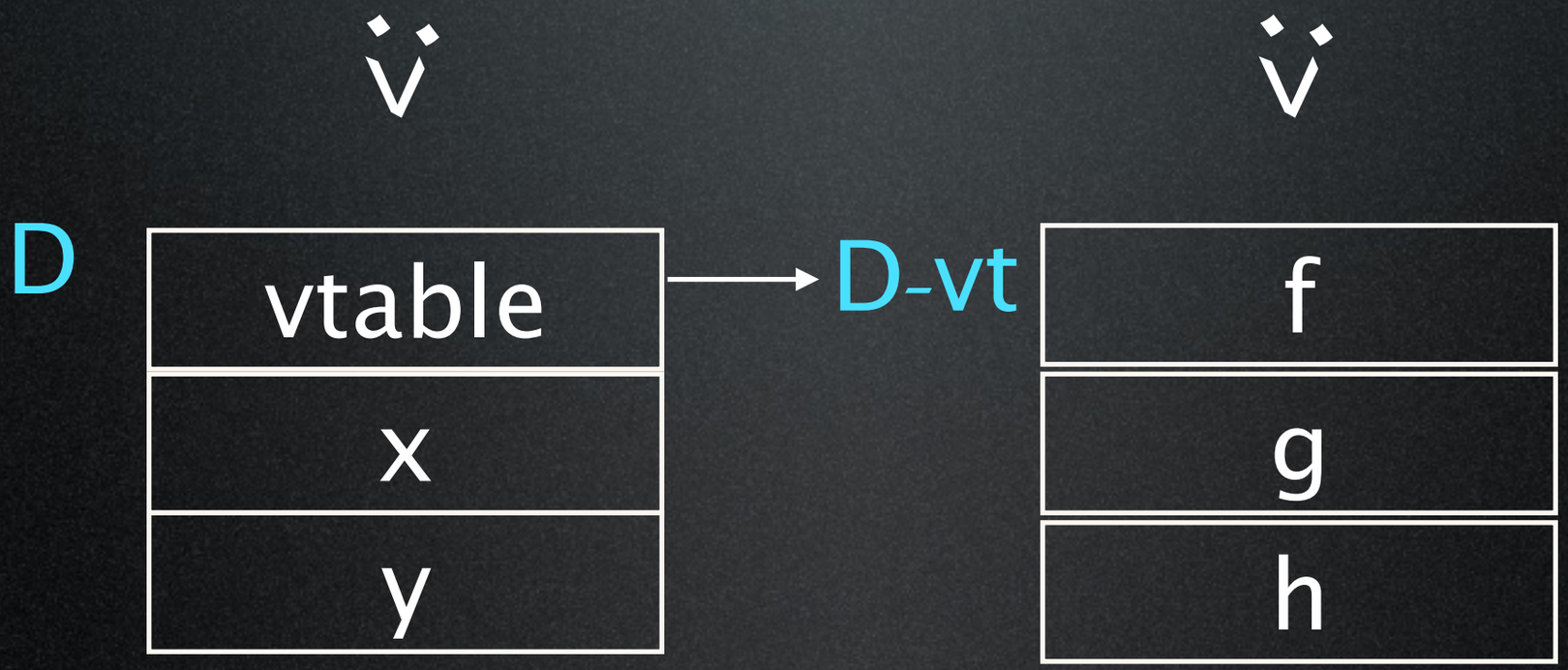
```
(module $C
  (type $f (import "" "f") (func (ref $C) i32 → i32))
  (type $g (import "" "g") (func (ref $C) → i32))
  (type $C (import "" "C") (struct (ref $Cvt) (mut i32)))
  (type $Cvt (import "" "Cvt") (struct (ref $f) (ref $g)))
  …
  (global (export "Cvt") (ref $Cvt) …)
)
```

Example - Classes

```
class D extends C {
  double y;
  override int g();
  int h();
}
```

Example - Classes





Example - Classes

```
(module $D
  (type $f (import "" "f") (func (ref $C) i32 → i32))
  (type $g (import "" "g") (func (ref $C) → i32))
  (type $C (import "" "C") (struct (ref $Cvt) (mut i32)))
  (type $Cvt (import "" "Cvt") (struct (ref $f) (ref $g)))

  (type $f (export "C" "f"))
  (type $g (export "C" "g"))
  (type $C (export "C" "C"))
  (type $Cvt (export "C" "Cvt"))
  (global (import "C" "Cvt") (ref $Cvt) …)

  (type $h (import "" "h") (func (ref $D) → i32))
  (type $D (import "" "D") (struct (ref $Dvt) (mut i32) (mut f64)))
  (type $Dvt (import "" "Dvt") (struct (ref $f) (ref $g) (ref $h)))
  …
  (global (export "Dvt") (ref $Dvt) …)
)
```

Example - Classes

```
class E extends D {
  int j();
  …
}
```

Example - Classes

```
(module $E
  (type $f (import "" "f") (func (ref $C) i32 → i32))
  (type $g (import "" "g") (func (ref $C) → i32))
  (type $C (import "" "C") (struct (ref $Cvt) (mut i32)))
  (type $Cvt (import "" "Cvt") (struct (ref $f) (ref $g)))
  (type $h (import "" "h") (func (ref $D) → i32))
  (type $D (import "" "D") (struct (ref $Dvt) (mut i32) (mut f64)))
  (type $Dvt (import "" "Dvt") (struct (ref $f) (ref $g) (ref $h)))

  (type $f (export "D" "f"))
  (type $g (export "D" "g"))
  (type $C (export "D" "C"))
  (type $Cvt (export "D" "Cvt"))
  (global (import "D" "Dvt") (ref $Dvt) …)

  (type $j (import "" "j") (func (ref $E) → i32))
 …
  (type $E (import "" "E") (struct (ref $Evt) …))
  (type $Evt (import "" "Evt") (struct …))
  …
  (global (export "Evt") (ref $Evt) …)
)
```

Example - Classes

import class X; // extends Y extends Z

class E **extends** D {
  X j();
  ...
}

Example - Classes

```
(module $E
  (type $f (import "" "f") (func (ref $C) i32 → i32))
  (type $g (import "" "g") (func (ref $C) → i32))
  (type $C (import "" "C") (struct (ref $Cvt) (mut i32)))
  (type $Cvt (import "" "Cvt") (struct (ref $f) (ref $g)))
  (type $h (import "" "h") (func (ref $D) → i32))
  (type $D (import "" "D") (struct (ref $Dvt) (mut i32) (mut f64)))
  (type $Dvt (import "" "Dvt") (struct (ref $f) (ref $g) (ref $h)))

  (type $f (export "D" "f"))
  (type $g (export "D" "g"))
  (type $C (export "D" "C"))
  (type $Cvt (export "D" "Cvt"))
  (global (import "D" "Dvt") (ref $Dvt) …)

  (type $f (import "" "f") (func (ref $C) i32 → i32))
  (type $g (import "" "g") (func (ref $C) → i32))
  … //
  (type $Z (import "" "C") (struct (ref $Cvt) (mut i32)))
  (type $Zvt (import "" "Cvt") (struct (ref $f) (ref $g)))
  (type $h (import "" "h") (func (ref $D) → i32))
  (type $Y (import "" "D") (struct (ref $Dvt) (mut i32) (mut f64)))
  (type $Yvt (import "" "Dvt") (struct (ref $f) (ref $g) (ref $h)))
  (type $X (import "" "D") (struct (ref $Dvt) (mut i32) (mut f64)))
  (type $Xvt (import "" "Dvt") (struct (ref $f) (ref $g) (ref $h)))

  (type $j (import "" "j") (func (ref $E) → i32))
 …
  (type $E (import "" "E") (struct (ref $Evt) …))
  (type $Evt (import "" "Evt") (struct …))
  …
  (global (export "Evt") (ref $Evt) …)
)
```

summary

every language has structural types, APIs assume them

no scalable, modular way to lower structural into nominal types

they need to be built-in

nominal types are useful to (cross module safety)

Wasm needs both