Observations about Types in Wasm

Andreas Rossberg

Dfinity

Generality

Performance

Simplicity

Generality

Performance

Simplicity

gc mvp priorities

1. safety

2. generality

3. simplicity

4. performance

gc mvp priorities

1. safety

2. generality

3. simplicity

4. performance

gc long-term priorities

1. safety

2. generality

3. performance

4. simplicity

types in wasm

define data layout, not language types

… e.g., structure of vtable, not definition of class

exist to aid engine, not compiler writer

… e.g., to validate and generate efficient code

observation 1

Wasm's types will never be expressive enough to handle every static type system

(not even close, a universal type system does not exist)

casts in wasm

are primarily an escape hatch for generated code,
not the reflection of language-level cast constructs
… e.g., to work around limitations of Wasm types

but can sometimes piggy-back language casting
… though neither necessary nor sufficient

observation 2

Wasm's runtime types will never be expressive enough to handle every dynamic casting system

(especially not if casts ought to be efficient)

compiling generics

generics, monomorphised

```
// C#
class A {}
class B {}

void foo(A a) {}
void fob(B b) {}

T bar<T>(T x) {
  return x;
}

foo(bar<A>(new A()));
```

```
(type $A (struct))
(type $B (struct))

(func $foo (param $x (ref $A)))
(func $fob (param $x (ref $B)))

(func $bar<A> (param (ref $A)) (result (ref $A))
  (local.get 0)
)
(func $bar<B> (param (ref $B)) (result (ref $B))
  (local.get 0)
)

(call $foo<A>
  (call $bar (struct.new $A))
)
```

generics, erased

```
// C#
class A {}
class B {}

void foo(A a) {}
void fob(B b) {}

T bar<T>(T x) {
  return x;
}

foo(bar<A>(new A()));
```

```
(type $A (struct))
(type $B (struct))

(func $foo (param $x (ref $A)))
(func $fob (param $x (ref $B)))

(func $bar<?> (param anyref) (result anyref)
  (local.get 0)
)

(call $foo
 (cast (rtt.canon $A)
  (call $bar<?> (struct.new $A))
 )
)
```

monomorphisation?

```
void bang<T>(int i, T x) {
  if (i < 0) return;
  bang<T[]>(i - 1, new T[3] {x, x, x});
}

bang<String>(30, “boo!”);
```

observation 3

The number of generic type instantiations is statically unbounded in polymorphic languages

(unless they are specifically designed with the limitations of monomorphisation in mind, e.g., C++)

```
void bang<T>(int i, T x) {
  if (i < 0) return;
  bang<T[]>(i - 1, new T[3] {x, x, x});
}

bang<String>(30, “boo!”);
```

```
Object[] array = new Object[11];

void bang<T>(int i, T x) {
  if (i < 0) return;
  bang<T[]>(i - 1, new T[3] {x, x, x});
}

bang<String>(30, “boo!”);
```

```
Object[] array = new Object[11];

void bang<T>(int i, T x) {
  if (i < 0) return;    array[i] = x;
    bang<T[]>(i - 1, new T[3] {x, x, x});
}

bang<String>(30, “boo!”);
```

```
Object[] array = new Object[11];

void bang<T>(int i, T x) {
    if (i < 0) return;    array[i] = x;
    bang<T[]>(i - 1, new T[3] {x, x, x});
}

bang<String>(30, “boo!”);

var saaa = (String[][][]) array[7];    // works
```

```
Object[] array = new Object[11];

void bang<T>(int i, T x) {
   if (i < 0) return;    array[i] = x;
   bang<T[]>(i - 1, new T[3] {x, x, x});
}

bang<String>(30, “boo!”);

var saaa = (String[][][]) array[7];    // works
var saaa2 = (String[][][]) array[6];  // throws!
```

observation 4

The number of runtime types is statically unbounded in polymorphic languages

(unless they are specifically designed with the limitations of monomorphisation in mind, e.g., C++,
or already enforce erasure in their semantics, e.g., Java)

conclusion 1

monomorphisation does not generally work

…neither for static types nor for runtime types

(NB: polymorphic recursion is just one counter-example)

corollary 1

creating RTTs from static definitions is insufficient

(when they are supposed to piggyback source types)

generics

```
class A {}

class B<T> {}

B<A> bar(){
  return new B<A>();
}

void foo(B<A> x){}

foo(bar());
```

generics & modularity

```
// A.cs
class A {}

// B.cs
class B<T> {}

// C.cs
B<A> bar(){
  return new B<A>();
}

// D.cs
void foo(B<A> x){}

// E.cs
foo(bar());
```

modularity

independent modules

separate validation

separate compilation

dynamic linking

wasm is based on open world principle!

modular compilation

imagine implementing not C# but .NET
(think Blazor with GC types)

reducing modularity in .NET-to-Wasm translation
would break the emulation

(analogous problems with other modular languages)

observation 5

translating modularity requiresmapping module boundaries

(whole program translation ought to be an option, not a requirement)

modular compilation

imagine implementing not C# but .NET
(think Blazor with GC types)

increasing dependencies in .NET-to-Wasm
translation would break the emulation

observation 6

translating modularity also requires mapping
module topologies

(source-to-Wasm translation must not introduce
extra dependencies between separately compiled user modules)

end-to-end modularity

source module ↦ wasm module(s)

⋮ ⋮

source compilation ↦ wasm compilation

⋮ ⋮

source linking ↦ wasm linking

⋮ ⋮

source dependency ↦ wasm dependency

types & modularity

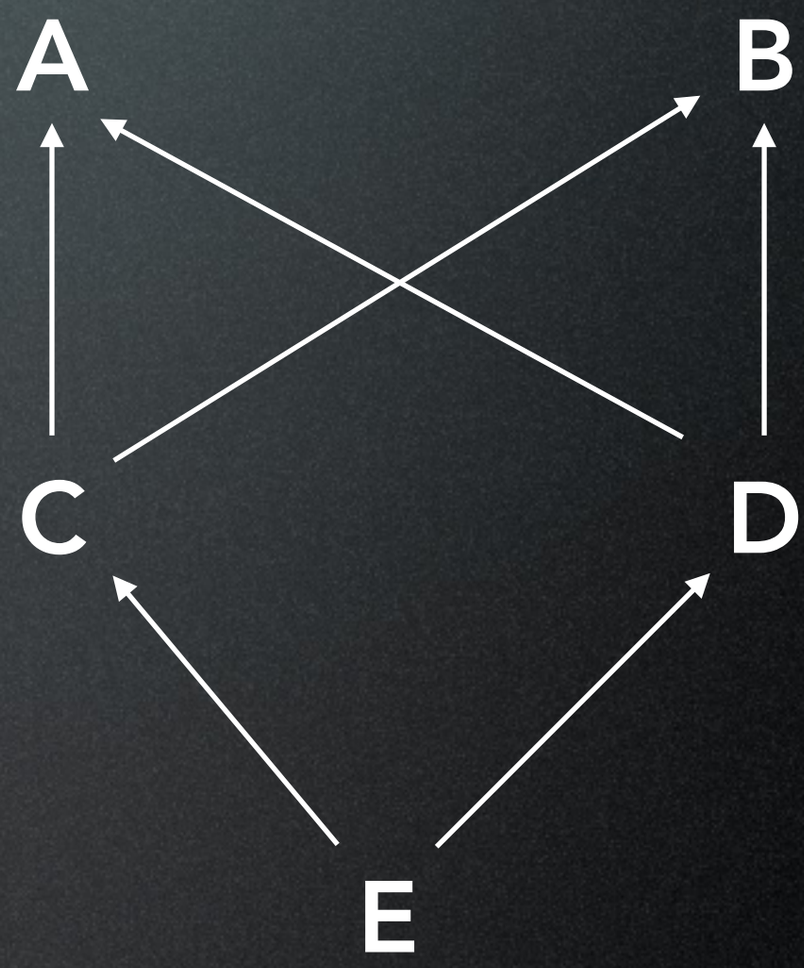
```
// A.cs
class A {}

// B.cs
class B<T> {}

// C.cs
B<A> bar(){
  return new B<A>();
}

// D.cs
void foo(B<A> x){}

// E.cs
foo(bar());
```



observation 7

generic types are inherently structural in nature

(infinite number of them; arrays are a form of built-in generic)

generic implies structural

```
class T {
    A a;
    B b;
    C c;
    D d;
}
```

```
class T<X, Y, Z, U> {
    X a;
    Y b;
    Z c;
    U d;
}
```

$T<A_1, B_1, C_1, D_1> = T<A_2, B_2, C_2, D_2>$
iff $A_1 = A_2$
$B_1 = B_2$
$C_1 = C_2$
$D_1 = D_2$

observation 8

dynamic RTT creation

$\wedge$

structural nature of generics

$\Longrightarrow$

dynamic RTT canonicalisation

```
class C<T> { public T x; public C(T t) { x = t; } }
```

```
Object[] array = new Object[11];
```

```
void bang<T>(int i, T x) {
  if (i < 0) return;    array[i] = x;
  bang<C<T>>(i - 1, new C<T>(x));
}
```

```
bang<String>(30, "boo!");

var cccs = (C<C<C<String>>>) array[7];   // works
var cccs2 = (C<C<C<String>>>) array[6];  // throws!
```

conclusion 2

dynamic RTT canonicalisation is necessary

(it is what .NET does as well under the hood)

observation 9

a central per-application module does not help
(for this reason and others)

(inherently static, and “application” is not a modular concept)

observation 10

structural types are a red herring wrt canonicalisation, relevant question is the semantics of recursive types

(which can be inductive even when structural)

flavours of recursive types

equi-recursive

recursion is implicit, allows arbitrary cycles
(co-inductive, i.e., cyclic graph algorithms)

iso-recursive

recursion groups are explicit, cycles only through backrefs
(inductive, i.e., tree algorithms)

both can be structural; nominal always has iso flavour

drawbacks of iso-recursion

cannot encode equi-recursive types

order/name dependency that is difficult to pave over

abstraction only over groups of mutually recursive types

...at odds with modular facilities of some languages

but worth investigating as a fallback

observation 11

We already require function types to be structural, to not get in the way of modularity

(similar arguments applied when CG decided this)

observation 12

Link-time checking has to be structural by nature

(and incoherence between inter- vs intra-module type semantics is neither desirable nor a simplification)

summary

structural types are inherent in all languages

not reducible to nominal in a sufficiently modular manner

end-to-end modularity is a central use case

dynamic runtime type creation is needed

dynamic runtime type canonicalisation is needed

the real question is recursive types

did not talk much about...

how concretely these observations affected proposal

...and future directions

many other constraints

many other dimensions of the design