

Wob and Waml Compiling for Wasm GC

Andreas Rossberg

Exploring and Evaluating

Wrote two compilers for two different kinds of languages:

Wob – object-oriented language (in the style of Java/C#/Scala)

Waml – functional language (in the style of SML/Ocaml)

Each represent central features of their paradigm

OO features in Wob

classes

inheritance

generics

safe downcasts

FP features in Waml

currying + closures

polymorphism + type inference

algebraic data types + pattern matching

higher-order modules

Tools

interpreter

compiler

REPL
(interpreted or compiled)

JavaScript runner

static linker (tbf)

Separate compilation

cannot generally know all use sites

implies need for **compositional** compilation scheme

Wob

What's in Wob

nominal classes with single inheritance

generic classes and methods

type-safe downcasts with reified generics

primitive data types and boxing

unboxed tuples and arrays

What's not in Wob

“everything is an object”

interfaces, generics with constraints or variance

covariant mutable arrays (but covariant readonly arrays)

autoboxing (explicit operator)

access control on classes

Primitive data types

Some representative types: Bool, Byte, Int, Float, Text

Always unboxed as object, tuple, or array field (except Text)

Boxed versions are different type: T\$, boxing operator e\$

...boxing is lightweight: T\$ is not an object

Wob type	Value rep	Field rep	Boxed rep (T\$)
Bool	i32	i8	i31ref
Byte	i32	i8	i31ref
Int	i32	i32	ref (struct i32)
Float	f64	f64	ref (struct f64)
Text	ref (array (mut i8))	ref (array (mut i8))	ref (array (mut i8))

Arrays

Mutable array types $T[]$, invariant

Readonly array types $T[]!$, covariant, $T[] <: T[]!$

Unboxed primitive element types

Reference elements in uniform representation

Wob type	Wasm rep
$\text{Byte}[], \text{Byte}[]!$	<code>ref (array (mut i8))</code>
$\text{Int}[], \text{Int}[]!$	<code>ref (array (mut i32))</code>
$\text{Float}[], \text{Float}[]!$	<code>ref (array (mut f64))</code>
$C[], C[]!$	<code>ref (array (mut eqref))</code>

Tuples

Immutable, covariant

Lightweight: reference types but not objects

Unboxed primitive field types

Reference fields in uniform representation

NB: currently 8^N possible tuple representations of arity N

Wob type	Wasm rep
(Byte, Byte, Int, C, Float)	ref (struct i8 i8 i32 eqref i64)

Classes and Inheritance

```
class Counter(x : Int) {  
    var c : Int = x;  
    func get() : Int { return c };  
    func reset() { c := x };  
    func inc() { c := c + 1 };  
};  
  
class DCounter(x : Int) <: Counter(x) {  
    func dec() { c := c - 1 };  
};  
  
let c = new DCounter(10);  
c.dec(); assert c.get() == 9;
```

Classes and Inheritance

```
class C(x : Int) {  
    var c : Int = x;  
    func get() : Int { return c };  
    func reset() { c := x };  
    func inc() { c := c + 1 };  
}
```

```
class D(y : Int) <: C(y + 1) {  
    func dec() { c := c - 1 };  
    func reset() { c := y + 1 };  
}
```

Type	Wasm rep
C	struct (ref \$C-vt) (\$x i32) (\$c mut i32)
D	struct (ref \$D-vt) (\$x i32) (\$c mut i32) (\$y i32)
\$C-vt	struct (ref [ref \$C]→[]) (ref [ref \$C]→[]) (ref [ref \$C]→[])
\$D-vt	struct (ref [ref \$C]→[]) (ref [ref \$C]→[]) (ref [ref \$C]→[]) (ref [ref \$D]→[])

only half the story, will come back to this later...

Generics

```
class Exp() {
    func accept<A, R>(v : Visitor<A, R>, a : A) : R { unreachable };
}

class Var(x : Text) <: Exp() {
    func name() : Text { x };
    func accept<A, R>(v : Visitor<A, R>, a : A) : R { return v.visitVar(this, a) };
}

class Add(e1 : Exp, e2 : Exp) <: Exp() {
    func left() : Exp { e1 };
    func right() : Exp { e2 };
    func accept<A, R>(v : Visitor<A, R>, a : A) : R { return v.visitAdd(this, a) };
}

class Visitor<A, R>() {
    func visit(exp : Exp, a : A) : R { return exp.accept<A, R>(this, a) };
    func visitVar(exp : Var, a : A) : R { unreachable };
    func visitAdd(exp : Add, a : A) : R { unreachable };
}
```

Generics

```
class Env<T>() {  
    func lookup(x : Text) : T { ... };  
  
    ...  
};  
  
class Evaluator() <: Visitor<Env<Int$>, Int$>() {  
    func visitVar(exp : Var, env : Env<Int$>) : Int$ {  
        return env.lookup(exp.name())  
    };  
    func visitAdd(exp : Add, env : Env<Int$>) : Int$ {  
        let i1 = exp.left().accept<Env<Int$>, Int$>(this, env).$;  
        let i2 = exp.right().accept<Env<Int$>, Int$>(this, env).$;  
        return (i1 + i2)$;  
    };  
};
```

Generics

Generic types are lowered to `eqref`

...need Wasm casts to recover instantiation type

Generic methods imply first-class polymorphism

...static type specialisation is not possible

Requires uniform representation for any type that can be abstracted after the fact

...i.e., all reference fields become `eqref`

```
let a : Text[] = ["foo", "bar"];
```

```
func swap<T>(a : T[]) : T[] { return [a[1], a[0]] }; // can only return (array eqref)
```

```
f<Text>(a); // cannot cast (array eqref) to (array (ref $Text)) in Wasm, would be unsound
```

Explicit Casts

`exp :> T`

returns `exp` if its type is subtype of `T`

null otherwise (since we don't have exceptions)

only available on objects

type-safe, i.e., $C\langle A \rangle \neq C\langle B \rangle$ at runtime (like C#, Dart, unlike Java)

Explicit Casts

Simple type tags (or `rtt.fresh`) do not suffice

Need custom Wob runtime type representation

...[type expression tree](#), e.g., `D<C<A>, E<B, B>>`

...ultimately represented as `eqref`

...generic type parameters are [reified](#) as rep parameters

Wob type (T)	Runtime value ($\llbracket T \rrbracket$)
<code>Int</code>	<code>i31ref(3)</code>
<code>C</code>	<code>ref \$C-vt</code>
<code>T[]</code>	<code>ref (array (i31ref(9)) $\llbracket T \rrbracket$)</code>
<code>(T, U, V)</code>	<code>ref (array (i31ref(8)) $\llbracket T \rrbracket \llbracket U \rrbracket \llbracket V \rrbracket$)</code>
<code>C<A, B></code>	<code>ref (array (ref \$C-vt) $\llbracket A \rrbracket \llbracket B \rrbracket$)</code>

Generic Classes and Types

```
class St<T>(sz : Int) {  
    var s : T[] = new T[sz](null);  
    var n : Int = 0;  
    func push(x : T) { s[n] := x; n += 1 };  
    func pop() : T { n -= 1; return s[n] };  
}
```

```
class ESt<T>(sz : Int) <: St<T>(sz) {  
    func clear() { n := 0 };  
}
```

Type	Wasm rep
St	struct (ref \$St-vt) (\$T eqref) (\$sz i32) (\$s mut ref (array eqref)) (\$n i32)
ESt	struct (ref \$ESt-vt) (\$T eqref) (\$sz i32) (\$s mut ref (array eqref)) (\$n i32) (\$T eqref)
\$St-vt	struct (ref [ref \$St]→[]) (ref [ref \$St]→[]) (\$St-id eqref)
\$ESt-vt	struct (ref [ref \$St]→[]) (ref [ref \$St]→[]) (\$St-id eqref) (ref [ref \$D]→[]) (\$ESt-id eqref)

(not yet deduping inherited parameters)

Implementation of explicit casts

1. Check for null
 - ...succeed if true
2. Wasm `cast` to instance type

could be avoided with `rtt.fresh`

3. Ref-compare class id at known offset in vtable
 - ...fail if not equal
4. For each type parameter, compare `type rep` at known offset in instance
 - ...currently, call to recursive runtime function;
could be `ref.eq` by implementing runtime type rep canonicalisation
 - ...fail if not equal

could be avoided with
wasm generics + `rtt.canon`

Polymorphic recursion

```
class C<X>(x : X) { let x : X = x };
```

```
func polyr<X>(i : Int, k : Int, x : X, p : Object[]) {
    if i == k { return };
    let cx = new C<X>(x);
    p[i] := cx;
    polyr<C<X>>(i + 1, cx);
};
```

```
var p : Object[] = new Object[20](null);
polyr<Int$>(0, 20, 66$, p);
```

```
assert p[0] :> C<C<C<Int$>>> == null;
assert p[2] :> C<C<C<Int$>>> != null;
```

Type structure

Primitive

bools

numbers

text

Nominal

objects

Structural

arrays

tuples

(functions)

Hybrid

generics

want runtime type canonicalisation

Use of Wasm casts

As first stage of explicit source-level casts

On projections from tuples or arrays, for reference types

```
let a = [c, d, c]; let x : C = a[1];
```

On results of generic call sites

```
func f<T>(x : T) : T {...}; let x : C = f<C>(c);
```

In overriding methods (**this** as well as specialised generic parameters need to lower to root type)

```
class Visitor<A, R>() {  
  func visitVar(exp : Var, a : A) : R {...} // lowers to [(ref $Visitor) (ref $Var) eqref] → [eqref]  
};
```

```
class Evaluator() <: Visitor<Env<Int$>, Int$>() {  
  func visitVar(exp : Var, env : Env<Int$>) : Int$ { env.lookup(x.name()) }  
};
```

Null

```
let n = null;
```

```
let t : Text = n;  
let c : C = n;
```

```
let n3 = (n, n, n);  
let t : (Text, C, C[]) = n3;
```

Wasm lacks null type, hence need to insert spurious coercions

Latter case works only because we use uniform representation eqref

...would break if we wanted to use future Wasm generics

Imports and Modules

```
import {T, a, b} from "url"  
import M_{T, a, b} from "url"
```

Accesses exports named T, a, b from designated module
(directly or as M_T, M_a, M_b, respectively)

Importing the same URL multiple times or in multiple
places yields the same module instance

Batch Compilation

Each source file **compiles** to a Wasm module

Module body becomes **start function**

Web signature of a module is stored in **custom section**

Imported modules must be available at compile time
(incremental compilation, not independent compilation)

Language im/exports map to Wasm im/exports

Runtime system

Simple Wasm module containing a handful of intrinsics:

- “mem”, “mem_alloc”: scratch memory for text literals

- “text_new”, “text_cat”, “text_eq”: text primitives

- “rtt_eq”: runtime type comparison function

Compiled Web modules import them when needed

Optionally, compiler can generate headless code that includes relevant functions

Loading

Loading is straightforward

...recursive instantiation

...based only on import/export name matching

Agnostic to language semantics and type system

Any tool chains could load, link, or merge

JS runner (node)

```
// wob.js, usage: node wob.js name

let fs = require('fs');

let registry = {__proto__: null};

async function link(name) {
    if (! (name in registry)) {
        let binary = await fs.readFileSync(name + ".wasm");
        let module = await WebAssembly.compile(new Uint8Array(binary));

        for (let im of WebAssembly.Module.imports(module)) {
            link(im.module);
        }

        let instance = await WebAssembly.instantiate(module, registry);
        registry[name] = instance.exports;
    }
    return registry[name];
}

async function run(mainname) {
    let exports = await link(mainname);
    return exports.return;
}

run(process.argv[2]);
```

REPL

Also supports import declarations

...links Wasm module into REPL

...compiles first if only source is available

Every single user input is also compiled into Wasm

...previous environment injected as inputs

This implies incremental dynamic linking

Static linking

Simple command line tool `wln`

...merges Wasm modules provided on command line

...synthesises new start function if necessary

...again based only on import/export name matching
(can check Wasm im/export types, but optional)

Also `agnostic` to language and its semantics

Can link repeatedly into incrementally larger modules

Components

Once the module linking and components proposal is ready, want to support that as well

Natural extension of current implementation

component = compiled, optionally linked module(s)

Compiler could spit out Wasm module types

Could also generate module linking scripts (linker, too)

...no further infrastructure required

to be continued...