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
Byte[], Byte[]!	ref (array (mut i8))
Int[], Int[]!	ref (array (mut i32))
Float[], Float[]!	ref (array (mut f64))
C[], C[]!	ref (array (mut eqref))

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⟨A⟩ ≠ C⟨B⟩ 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 (⟦T⟧)
Int	i31ref(3)
C	ref $C-vt
T[]	ref (array (i31ref(9)) ⟦T⟧))
(T, U, V)	ref (array (i31ref(8)) ⟦T⟧ ⟦U⟧ ⟦V⟧))
C⟨A, B⟩	ref (array (ref $C-vt) ⟦A⟧ ⟦B⟧))

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

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 rtt.fresh

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	Nominal	Structural	Hybrid
bools	objects	arrays	generics
numbers		tuples	
text		(functions)	

want runtime type canonicalisation

Use of Wasm casts

1. As first stage of explicit source-level casts
2. On projections from tuples or arrays, for reference types

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

3. On results of generic call sites

   ```
   func f⟨T⟩(x : T) : T {…};     let x : C = f⟨C⟩(c);
   ```

4. 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

Wob 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 Wob 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

Waml

What’s in Waml

currying + closures

polymorphism + type inference

algebraic data types + pattern matching

higher-order + first-class modules

…pretty much a complete ML dialect

What's *not* in Waml

exceptions

records

arrays

Primitive data types

Representative base types: Bool, Byte, Int, Float, Text

Unboxed as locals or module fields (except Text)

Otherwise boxed for universal representation

Waml type	Value rep	Unboxed rep	Boxed rep
Bool	i32 & 0x1	i8 & 0x1	i31ref
Byte	i32 & 0xff	i8 & 0xff	i31ref
Int	i32 & 0x7fffffff	i32 & 0x7fffffff	i31ref
Float	f64	f64	ref (struct f64)
Text	ref (array (mut i8))	ref (array (mut i8))	ref (array (mut i8))

Data Types and Pattern Matching

```
rec data Exp =
  | Lit Int
  | Add Exp Exp
  | Mul Exp Exp

rec val eval e =
  case e of
  | Lit x => x
  | Add e1 e2 => eval e1 + eval e2
  | Mul e1 e2 => eval e1 * eval e2

val exp = Add (Lit 3) (Mul (Add (Lit 1) (Lit 2)) (Lit (-3)))
val x = eval exp
```

Polymorphism

```
rec data List a = Nil | Cons a (List a)

rec val fold xs y f =       ;; foldl : List a -> b -> (a -> b -> b) -> b
  case xs of
  | Nil => y
  | Cons x xs' => fold xs' (f x y) f

val list = [1, 2, 5, 6, -8]
val sum = fold list 0 (fun i sum => sum + i)
assert sum == 6
```

all typing implicit, way more polymorphic on average than in OO,
and more expressive polymorphism ⇒ universal representation everywhere

Data Representation

Nullary constructors are unboxed

Others are boxed structs with tag field

Even monomorphic constructor arguments need to be in uniform representation, since modules allow type abstraction after the fact

Waml type	Wasm representation
Nil | Cons a (List a)	i31ref resp. ref (struct i32 eqref eqref)
(Int, Text)	ref (struct eqref eqref)
ref T	ref (struct (mut eqref))
T -> U	ref (struct i32 …)

Currying

```
val caller f = f 1 2      ;; caller : (Int -> Int -> a) -> a

;; Exact application
val add x y = x + y      ;; add : Int -> Int -> Int

assert caller add == 3

;; Under-application
val add3 x y z = x + y + z      ;; add3 : Int -> Int -> Int -> Int

assert caller add3 4 == 7

;; Over-application
val add' x = let val f y = x + y in f      ;; add' : Int -> (Int -> Int)

assert caller add' == 3
```

Implementing Currying

Non-static calls require 2-dimensional dispatch

…on call arity and callee arity

Call arity is static

Callee arity is dynamic

Port standard *eval/apply* technique [cf. GHC, OCaml]

Compiling Currying: *eval/apply*

Every closure stores *syntactic* function arity

Non-static calls compiled by calling an *apply* combinator

generated for each call arity, dispatches on caller arity

when over-applied, splits call into two,
recursing into apply with lower arity

when under-applied, allocates internal closure that binds args,
aux function which performs apply with higher arity

All apply combinators and aux functions are mutually recursive

Must top out at some max arity, falling back to arg array

3-level Closure Type Hierarchy

```
$anyclos                    = struct i32
 +- $clos1                  = struct i32(1) (ref $code1)
 |    +- $clos1/i32         = struct i32(1) (ref $code1) i32
 |    +- $clos1/f64-i32     = struct i32(1) (ref $code1) f64 i32
 |   …
 +- $clos2                  = struct i32(2) (ref $code2)
 |   ...
 +- $clos3                  = struct i32(3) (ref $code3)
 |   …
 :
 +- $closM                  = struct i32(M) (ref $codeM)
 |   …
 +- $closVar                = struct i32(N>M) (ref $codeVar)
     ...
```

```
$code1 =
  func (param (ref $clos1) eqref) (result eqref)

$code2 =
  func (param (ref $clos2) eqref eqref) (result eqref)

$code3 =
  func (param (ref $clos3) eqref eqref eqref) (result eqref)

...

$codeM =
  func (param (ref $clos3) eqref eqref ... eqref) (result eqref)

$codeVar =
  func (param (ref $clos3) (ref $argv)) (result eqref)

$argv = array eqref

...
```

ML Modules

```
signature Set = {
    type Elem
    type Set
    val empty : Set
    val add : Elem -> Set -> Set
    val mem : Elem -> Set -> Bool
}

signature Ord = {
    type T
    val lt : T -> T -> Bool
}

module MakeSet (Elem : Ord) : Set with type Elem = Elem.T = {
    …
}

module IntSet = MakeSet Int
```

Compiling ML Modules

ML structures compiled to structs

ML functors compiled to functions and closures

Signature subtyping is compiled coercively

…higher-order for functor subtyping

…Wasm-level subtyping not needed

Types can be abstracted after the fact (sealing)

…more polymorphism, forces use of eqref even in data types

But structure fields can still be concrete refs, thanks to coercions

Module Representation

All module fields boxed, but with exact concrete type

…avoids cast on every module access (which are frequent)

Module types are structural, can nest, and typically have many fields

…infinitely many possible types, even for tiny modules

Waml module type	Wasm representation
{ val x : Int val t : Text val f : Int -> Int module M : { val y : Int; … } … }	struct i31ref (ref $text) (ref $clos1) (ref (struct i31ref …)) …
{ val x : Int; … } -> { val y : Text; … }	func [(ref $self) (ref struct i31ref …)] → [(ref struct (ref $text) …)]

Type structure

Primitive	Nominal	Structural	Hybrid
bools	(datatypes)*	datatypes*	poly datatypes
numbers		tuples	
text		functions	
		structures	
		functors	

* datatype are nominal on the core level but can be structurally abstracted on the module level; for all purposes of separate compilation, they must hence be handled as structural types

Use of Wasm casts

1. On projections from tuples, references, and datatypes

   val t = (D1, D2, D1) **val** x : D = t.1

2. On results of polymorphic call sites

   val f x = … **val** x : D = f D2

3. In calls to first-class functions (for currying)

NB: *not* for projections from structs

Separate compilation

Imports & batch compilation essentially analogous to Wob

REPL loader

JS loader (identical)

Static linker (identical)

(Componentification)

NB: would need multiple external implementations of type canonicalisation (in multiple languages) if Wasm did not have structural types built-in

Wrapping up
(Wob + Waml)

Things confirmed and learnt

Both languages are expressible with MVP

Some casting involved, but we knew that

…currying in FP was the worst case, not overriding in OO

Separate compilation was fairly straightforward

…but depends on structural lowerings

…some types cannot be bounded without significant loss

Some unexpected annoyances that could be fixed

Current annoyances

No resolution for non-nullable locals yet

…mostly using nullable ones, spurious **ref.as_non_null**

Missing ability to initialise arrays from segments

…indirection through memory, needed only for that

Missing null type

…need to inject spurious conversions, scalability limited

Missing ability to use i31.new in constant expressions

…needed to avoid more nullable types in globals

Missing rtt.fresh variant

…need to implement in user space for source casts, type identity in vtables

Missing function subtyping

…requires more casts and eta expansions

Next steps

Port to iso-recursive proposal branch

…not expecting specific problems, but you never know

…these compilers are fairly simple,
good test beds for running alternative designs against

Address identified annoyances in proposal

Perhaps performance measurements?

Think more about integration with components proposal

Running in V8 / Node.js

Most Wob tests pass, except for some spurious cast failures

…probably due to incomplete implementation of **rtt.canon**?

Many Waml tests don’t pass, e.g., all that use data types

…V8 can’t handle (ref null (rtt …)), needed for constructors

…changed in proposal one or two years ago

Except these two issues, everything else seems to work

Characteristics of Type Lowering

	structural	recursive	subtyped	immutable	non-null	casted
primitive				(X)	X	
tuples	X			X		
arrays	(X)					
objects		X	X	X		X
methods		X	X	X	X	
data types	X	X		X	X	X
closures	X / (X)	X	X*	(X)	- / X	X
modules	X		X	X	X	
functor cl's	X	X	X*	(X)	X	X
generics			X		- / X	X

Wob / Waml (X) used but could be avoided * nominal subtypes would suffice

https://github.com/WebAssembly/gc/tree/wob/proposals/gc/wob (Wob)

https://github.com/WebAssembly/gc/tree/waml/proposals/gc/waml (Waml)

https://github.com/WebAssembly/gc/tree/wob/proposals/gc/wln (Linker)

outtakes

Wob

Object and Class Representation

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

class D(x : Int) <: C(2*x) {
  let y = x + 42;
  var z : Int = x + 66;
  func inc() { c := c + 2 };
  func dec() { c := c – 2 };
};
```

```
type $Cinst = struct (ref $Cvt) i32 (mut i32)
type $Cvt = struct (ref $Cget) (ref $Cset) (ref $Cinc)
type $Cget = func (param (ref $Cinst)) (result i32)
type $Cset = func (param (ref $Cinst) i32)
type $Cinc = func (param (ref $Cinst))
type $Cclass =
  struct (ref $Cvt) (rtt $Cinst) (ref $Cnew) (ref $Cpre) (ref $Cpost)

type $Dinst = struct (ref $Cvt) i32 (mut i32) i32 i32 (mut i32)
type $Dvt = struct (ref $Cget) (ref $Cset) (ref $Cinc) (ref $Ddec)
type $Ddec = func (param (ref $Dinst))
type $Dclass =
  struct (ref $Dvt) (rtt $Dinst) (ref $Dnew) (ref $Dpre) (ref $Dpost)
         (ref $Cclass)
```

Object Construction

```
class D(x : Int) <: C(2*x) {
  let y = x + 42;
  var z : Int = x + 66;
  func inc() { c := c + 2 };
  func dec() { c := c – 2 };
};
```

```
type $Dinst = struct (ref $Cvt) i32 (mut i32) i32 i32 (mut i32)

func $Dnew (param $x i32) (result (ref $Dinst))
  local $Cx, $Dx, $Dy = call $Dpre ($x)
  local $d = struct.new ($Cvt) ($Cx) (0) ($Dx) ($Dy) (0) ($Crtt)
  call $Dpost ($d)
  return $d

func $Dpre (param $x i32) (result i32 i32 i32)
  local $Cx = call $Cpre (2 * $x)
  return $Cx, $x, ($x + 42)

func $Dpost (param $d (ref $Dinst))
  call $Cpost ($d)
  struct.set $Dz ($d) ((struct.get $Dx ($d)) + 66)
```

Generics and Closures

```
func foreach<T>(a : T[], f : T ->()) {
  var i : Int = 0;
  while (i < #a) {
    f(a[i]);
    i := i + 1;
  };
  return r;
};

let a = ["Hello ", "world", "\n"];
var r : Text = "";
foreach<Text>(func(x : Text) { r := r + x });
```

Generics and Closures (2)

```
class A<T>(x : T, n : Int) {
  let a = new T[n](x);
  func map<U>(f : T -> U) : U[] {
    var i : Int = 0;
    var b : U[] = [];
    while (i < #a) {
      let u = f(a[i]);
      if i == 0 { b := new U[#a](u) } else { b[i] := u };
      i := i + 1;
    };
    return b;
  }
};

let a = new A<Text>("hello", 20);
let b = a.map<Text>(func(x : Text) : Text { x + " world" });
```

Waml

Data Type Representation

```
rec data Exp a =
  | Zero
  | Lit a
  | Var Text
  | Add (Exp a) (Exp a)
  | Mul (Exp a) (Exp a)
  | Inf

let z = zero
let a = Add z (Lit 42)
```

```
type $ExpLit = struct i32 eqref
type $ExpVar = struct i32 eqref
type $ExpAdd = struct i32 eqref eqref
type $ExpMul = struct i32 eqref eqref

local $z = i31.new 5
local $a =
  struct.new (1) ($z) (struct.new (2) (42) ($rttExpLit)) ($rttExpAdd)
```

Retroactive type abstraction

```
module IntExp = {
    type Val = Int
    rec data Exp = Lit Int | Add Exp Exp
    val example : Exp
}

interface Exp = {
    type Val
    rec data Exp = Add Exp Exp | Lit Val
    val example : Exp
}

module Eval (E : Exp) = …

module IntEval = Eval IntExp
```