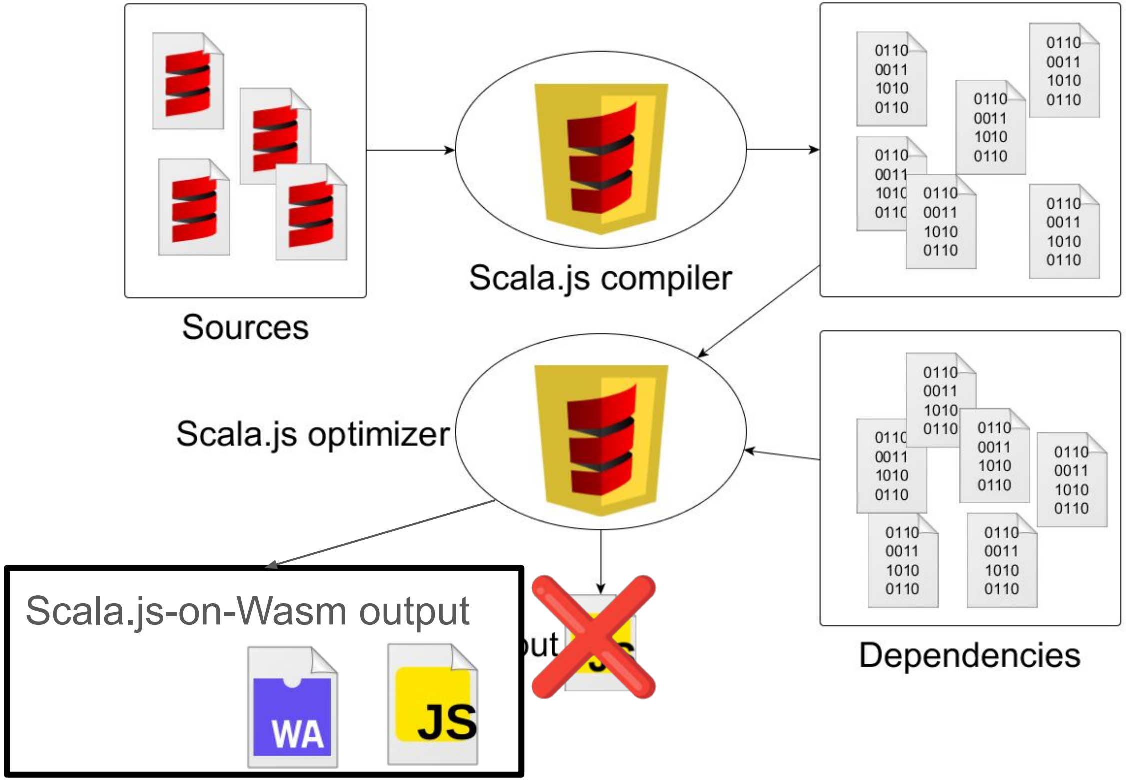
Compiling Scala.js to WebAssembly

May 9, 2025
Sébastien Doeraene

Scala.js compilation pipeline



Sources

Scala.js compiler

Scala.js optimizer

Dependencies

Scala.js-on-Wasm output

Scala.js IR

```scala
package helloworld

object Main {
 def main(args: Array[String]): Unit = {
   val result = computeResult(5, 11)
   println(result)
 }

 @noinline
 def computeResult(a: Int, b: Int): Int =
   a + 2 * b
}
```

```
module class helloworld.Main$ extends java.lang.Object {
 def main;[Ljava.lang.String;V(args: java.lang.String[]) {
   val result: int = this.computeResult;I;I;I(5, 11);
   mod:scala.Predef$.println;Ljava.lang.Object;V(result)
 }

 @hints(2) def computeResult;I;I;I(a: int, b: int): int = {
   (a +[int] (2 *[int] b))
 }

 constructor def <init>;V() {
   this.java.lang.Object::<init>;V();
   <storeModule>
 }
}
```

One Scala.js IR method in Wasm

```
module class helloworld.Main$ extends java.lang.Object {
  @hints(2) def computeResult;I;I;I(a: int, b: int): int = {
    (a +[int] (2 *[int] b))
  }
}
```

```
(func $f.helloworld.Main$.computeResult_I_I_I(type $126)
  (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)

  local.get $a
  local.get $b
  i32.const 1
  i32.shl
  i32.add)
```

(b << 1)

Object Model and Method Calls

Method calls

```
module class helloworld.Main$ extends java.lang.Object {
  def computeResult;I;I;I(a: int, b: int): int = (a +[int] this.twice;I;I(b))
  def twice;I;I(x: int): int = (2 *[int] x)
}
```

```
(func $f.helloworld.Main$.computeResult_I_I_I(type $127)
  (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
  local.get $a
  local.get $this                           }
  local.get $b                              } this.twice_I_I(b)
  call $f.helloworld.Main$.twice_I_I        }
  i32.add)

(func $f.helloworld.Main$.twice_I_I(type $128)
  (param $this (ref $c.helloworld.Main$)) (param $x i32) (result i32)
  local.get $x
  i32.const 1
  i32.shl)
```

A simple class

```
class Vec2(val x: Int, val y: Int)

class helloworld.Vec2 extends java.lang.Object {
  val helloworld.Vec2::x: int
  val helloworld.Vec2::y: int
  def x;I(): int = {
    this.helloworld.Vec2::x        getter for x
  }
  def y;I(): int = {
    this.helloworld.Vec2::y
  }
  constructor def <init>;I;I;V(x: int, y: int) {
    this.helloworld.Vec2::x = x;
    this.helloworld.Vec2::y = y;           constructor
    this.java.lang.Object::<init>;V()
  }
}
```

A simple class

```
@hints(2) def computeResult;I;I;I(a: int, b: int): int = {
  val v: helloworld.Vec2 = new helloworld.Vec2().<init>;I;I;V(a, b);
  (v.x;I() +[int] v.y;I())
}
```

```
(func $f.helloworld.Main$.computeResult_I_I_I(type $86)
  (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
  (local $1 (ref $c.helloworld.Vec2)) (local $v (ref $c.helloworld.Vec2))
  call $new.helloworld.Vec2                  } allocation
  local.tee $1
  local.get $a
  local.get $b                               } constructor call
  call $ct.helloworld.Vec2.<init>_I_I_V
  local.get $1
  local.set $v
  local.get $v
  struct.get $c.helloworld.Vec2 $f.helloworld.Vec2.x     } read v.x
  local.get $v
  struct.get $c.helloworld.Vec2 $f.helloworld.Vec2.y
  i32.add)
```

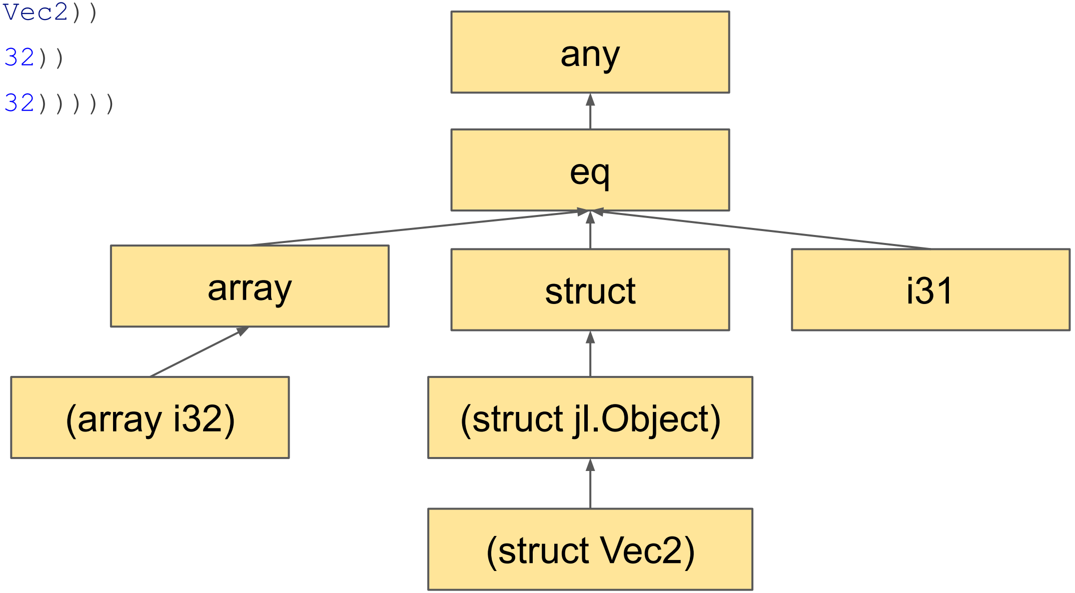
Wasm reference types

```
(type $c.helloworld.Vec2
  (sub $c.java.lang.Object
    (struct
      (field $vtable (ref $v.helloworld.Vec2))
      (field $f.helloworld.Vec2.x (mut i32))
      (field $f.helloworld.Vec2.y (mut i32)))))
```

Types

i32	f32
i64	f64
(ref null? $ht)	

Heap types (ht)



any

eq

array

struct

i31

(array i32)

(struct jl.Object)

(struct Vec2)

Why are `val` fields mutable?

```
class helloworld.Vec2 extends java.lang.Object {
  val helloworld.Vec2::x: int
  val helloworld.Vec2::y: int
  constructor def <init>;I;I;V(x: int, y: int) {
    this.helloworld.Vec2::x = x;
    this.helloworld.Vec2::y = y;
    this.java.lang.Object::<init>;V()
  }
}
```

assignments to `val` fields in the constructor

```
(func $ct.helloworld.Vec2.<init>_I_I_V (type $89)
  (param $this (ref $c.helloworld.Vec2)) (param $x i32) (param $y i32)
  local.get $this
  local.get $x
  struct.set $c.helloworld.Vec2 $f.helloworld.Vec2.x
  local.get $this
  local.get $y
  struct.set $c.helloworld.Vec2 $f.helloworld.Vec2.y)
```

assignments to `val` fields in the constructor

What about the allocation?

```
(func $f.helloworld.Main$.computeResult_I_I_I(type $86)
  (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
  (local $1 (ref $c.helloworld.Vec2)) (local $v (ref $c.helloworld.Vec2))
  call $new.helloworld.Vec2
  local.tee $1
  local.get $a
  local.get $b
  call $ct.helloworld.Vec2.<init>_I_I_V
  ...

(func $new.helloworld.Vec2 (type $90)
  (result (ref $c.helloworld.Vec2))
  global.get $d.helloworld.Vec2
  i32.const 0
  i32.const 0
  struct.new $c.helloworld.Vec2)
```

allocation with initial values for all the fields

Wasm structs

```
(type $structType
  (sub $optionalSuperType
    (struct
      (field $immutablefield tp)
      (field $mutableField (mut tp)))))
```

- `(ref $structType)` is the type of a reference to a `$structType`
- `(ref null $structType)` is a nullable variant
- `struct.new $structType` allocates a new `$structType`
  needs the initial values of the fields on the stack
- `struct.get $structType $structField`
  gets the field `$structField` of the `(ref null $structType)` on the stack
- `struct.set $structType $structField`
  sets the field `$structField` of the `(ref null $structType)` on the stack
  to the new value also on the stack

Inheritance

```
class Vec2(val x: Int, val y: Int)
class Vec3(x: Int, y: Int, val z: Int) extends Vec2(x, y)
```

```
(type $c.helloworld.Vec2
  (sub $c.java.lang.Object
    (struct
      (field $vtable (ref $v.helloworld.Vec2))
      (field $f.helloworld.Vec2.x (mut i32))
      (field $f.helloworld.Vec2.y (mut i32)))))
(type $c.helloworld.Vec3
  (sub $c.helloworld.Vec2
    (struct
      (field $vtable (ref $v.helloworld.Vec3))
      (field $f.helloworld.Vec2.x (mut i32))
      (field $f.helloworld.Vec2.y (mut i32))
      (field $f.helloworld.Vec3.z (mut i32)))))
```

repeat fields already declared in `Vec2`

Subtyping

```
class Vec2(val x: Int, val y: Int) {
  @noinline def coordsSum(): Int = x + y
}
class Vec3(x: Int, y: Int, val z: Int) extends Vec2(x, y)

@noinline def computeResult(a: Int, b: Int): Int = {
  val v = new Vec3(a, b, 3)
  v.coordsSum()
}
```

```
(func $f.helloworld.Main$.computeResult_I_I_I(type $86)
  ... (local $v (ref $c.helloworld.Vec3))
  local.get $v
  call $f.helloworld.Vec2.coordsSum_I
```

give a `Vec3` as argument to `coordsSum_I`

```
(func $f.helloworld.Vec2.coordsSum_I (type $89)
  (param $this (ref $c.helloworld.Vec2)) (result i32)
  local.get $this
  struct.get $c.helloworld.Vec2 $f.helloworld.Vec2.x
  ...
```

but coordsSum_I wants a `Vec2`

Virtual method calls

```scala
class Vec2(val x: Int, val y: Int) {
  @noinline def coordsSum(): Int = x + y
}
class Vec3(x: Int, y: Int, val z: Int) extends Vec2(x, y) {
  @noinline override def coordsSum(): Int = x + y + z
}

@noinline def computeResult(a: Int, b: Int): Int = {
  val v2: Vec2 = hide[Vec2](new Vec2(a, b))
  val v3: Vec2 = hide[Vec2](new Vec3(a, b, 3))
  v2.coordsSum() * v3.coordsSum()
}

@noinline def hide[T](x: T): T = x
```

} apparently, two calls to `Vec2.coordsSum()`

Virtual method calls

```
(local $v3 (ref null $c.helloworld.Vec2))
...
local.get $v3
ref.as_non_null
local.tee $4
local.get $4
struct.get $c.helloworld.Vec2 $vtable
struct.get $v.helloworld.Vec2 $m.helloworld.Vec2.coordsSum_I
call_ref $1
```

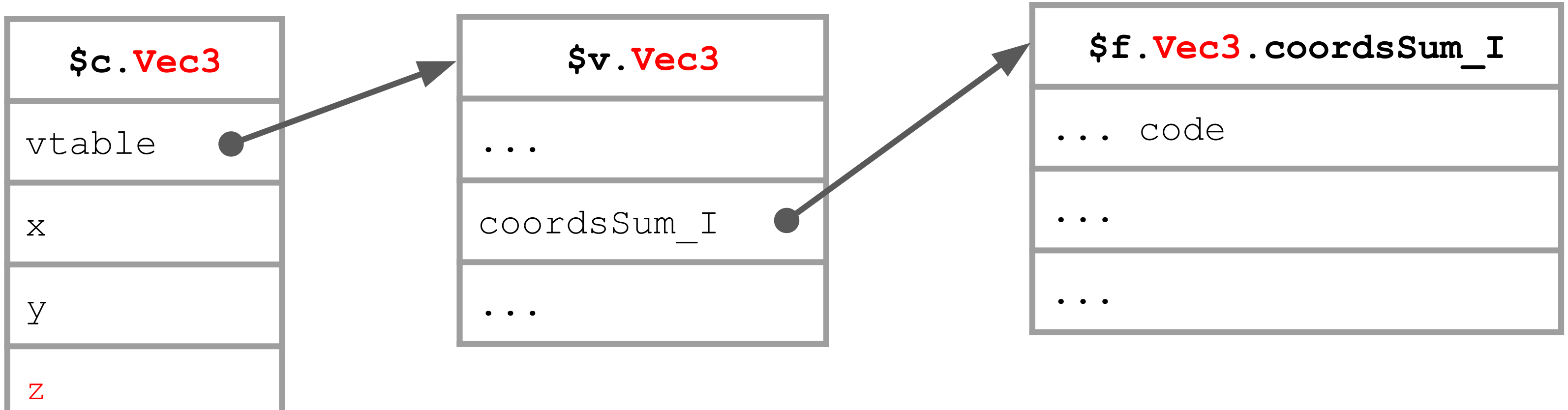
load v3 on the stack, and cast away nullability

duplicate the value on the stack (one for the this param; one for the struct.get below)

get the "vtable" of v3

get the field coordsSum in that vtable

call the function whose address is on top of the stack

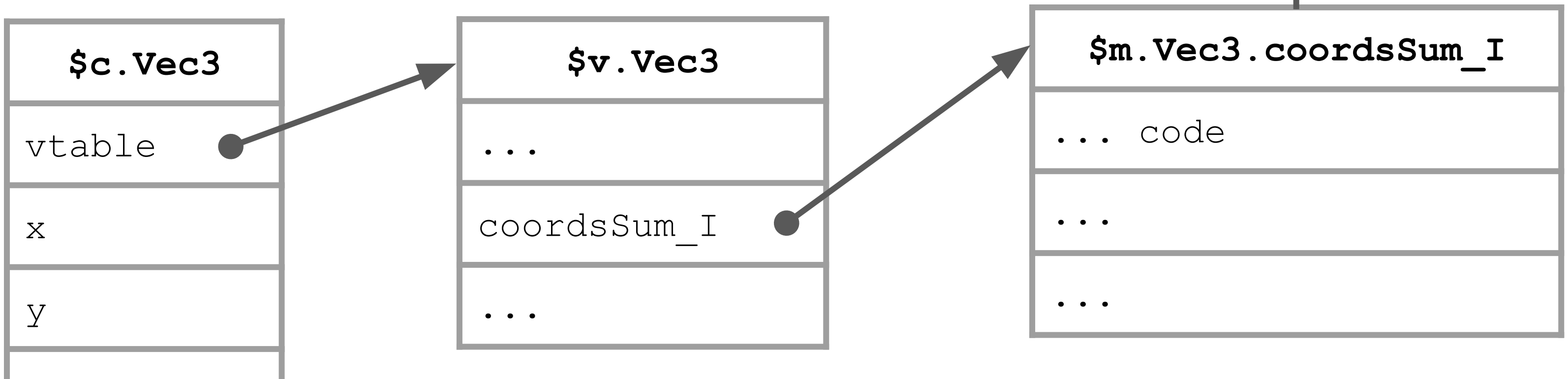


What is the type of that function pointer?

```
(func $f.helloworld.Vec2.coordsSum_I (type $90)
  (param $this (ref $c.helloworld.Vec2)) (result i32)
(func $f.helloworld.Vec3.coordsSum_I (type $112)
  (param $this (ref $c.helloworld.Vec3)) (result i32)
```

No common type!

```
(func $m.helloworld.Vec3.coordsSum_I (type $1)
  (param $this (ref any)) (result i32)
  local.get $this
  ref.cast (ref $c.helloworld.Vec3)
  return_call $f.helloworld.Vec3.coordsSum_I)
```



$c.Vec3

vtable

x

y

$v.Vec3

...

coordsSum_I

...

$m.Vec3.coordsSum_I

... code

...

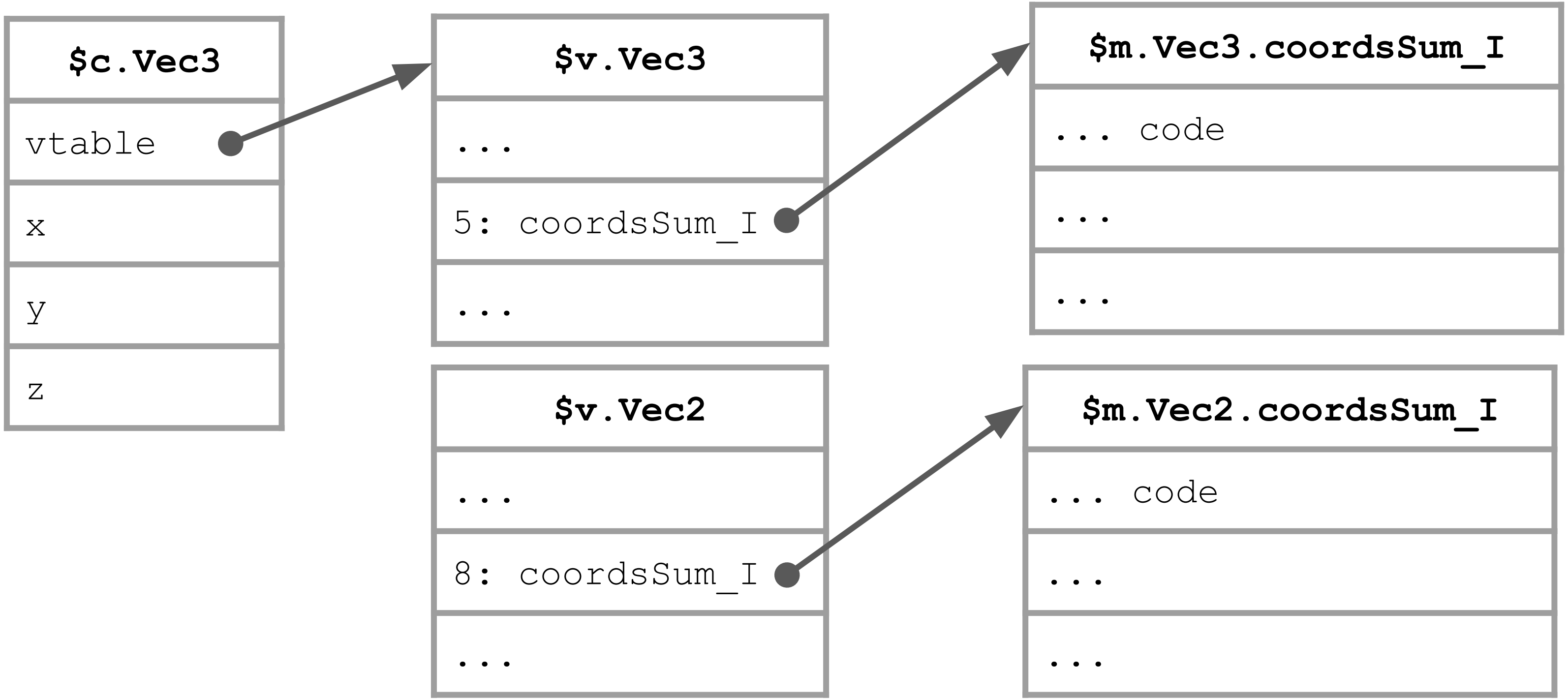
...

Interface method calls

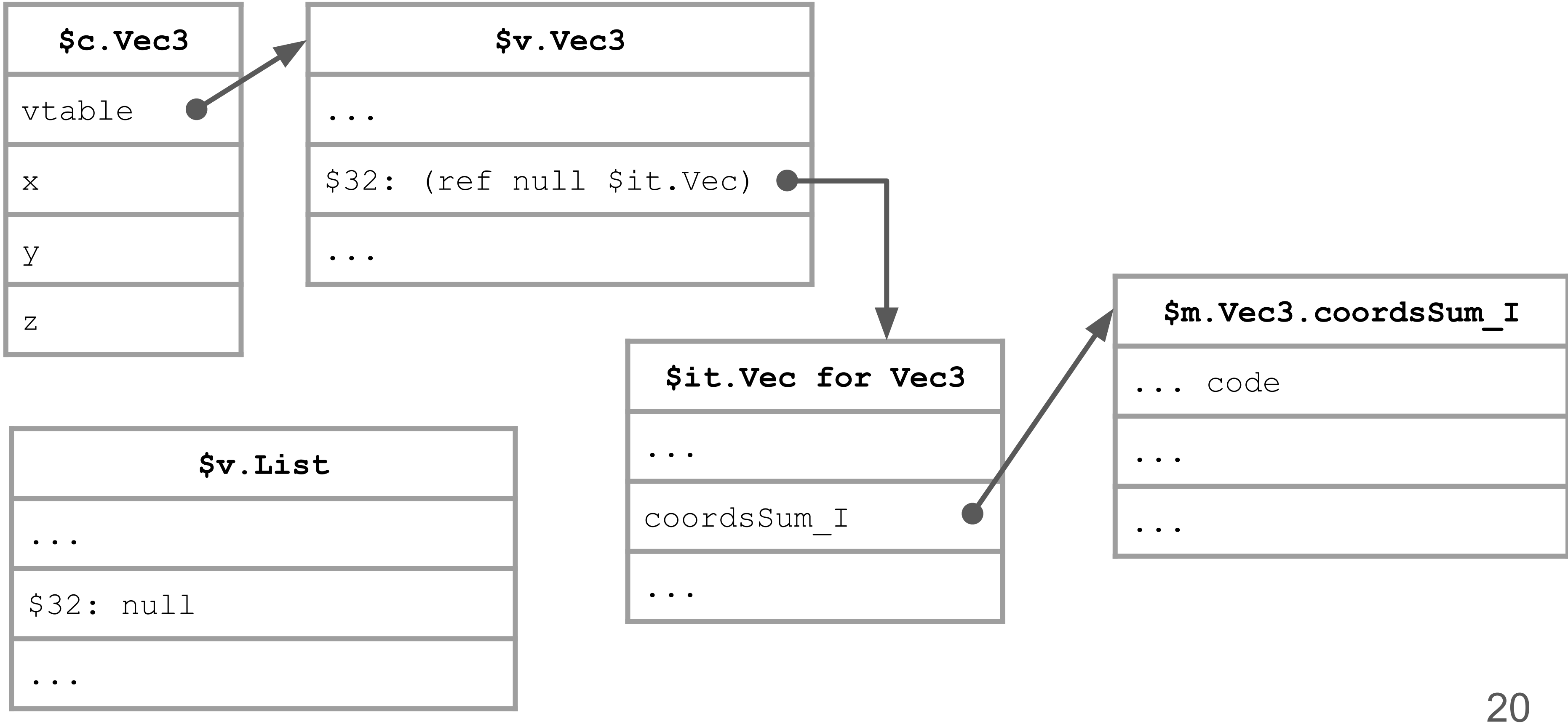
```
trait Vec {
  def coordsSum(): Int
}
class Vec2(val x: Int, val y: Int) extends Vec {
  def coordsSum(): Int = x + y
}
class Vec3(val x: Int, val y: Int, val z: Int) extends Vec {
  override def coordsSum(): Int = x + y + z
}

@noinline def computeResult(a: Int, b: Int): Int = {
  val v2: Vec = hide[Vec](new Vec2(a, b))
  val v3: Vec = hide[Vec](new Vec3(a, b, 3))
  v2.coordsSum() * v3.coordsSum()
}

@noinline def hide[T](x: T): T = x
```

} apparently, two calls to `Vec.coordsSum()`

Interface method calls: no common index



Interface method calls: one more indirection



$c.Vec3

vtable

x

y

z

$v.Vec3

...

$32: (ref null $it.Vec)

...

$it.Vec for Vec3

...

coordsSum_I

...

$m.Vec3.coordsSum_I

... code

...

...

$v.List

...

$32: null

...

Interface method calls: the code

```
local.get $v2
ref.as_non_null
local.tee $3
local.get $3
struct.get $c.java.lang.Object $vtable
struct.get $v.java.lang.Object $1
ref.cast (ref $it.helloworld.Vec)
struct.get $it.helloworld.Vec $m.helloworld.Vec.coordsSum_I
call_ref $1
```

} cast required because different types of `$it.X` in the same slot

Interface numbering

Need one index for every interface in the world!
(and a lot of entries in all the vtables too!)

Or do we? Most of the entries will be `null`.

Smarter: allocate the same index to interfaces that have no common subclass.

Efficient Type Inclusion Tests
Vitek, Jan & Nigel, R. & Krall, Horspool. (2000).
SIGPLAN Notices (ACM Special Interest Group on Programming Languages).

In practice, we need about 50 interface slots

Arbitrary method calls (for completeness)

```
type Vec = {
  def coordsSum(): Int
}
@noinline def computeResult(a: Int, b: Int): Int = {
  val v2: Vec = hide[Vec](new Vec2(a, b))
  val v3: Vec = hide[Vec](new Vec3(a, b, 3))
  v2.coordsSum() * v3.coordsSum()
}
```

```
local.get $v3
ref.as_non_null
local.tee $4
local.get $4
ref.cast (ref $c.java.lang.Object)
struct.get $c.java.lang.Object $vtable
i32.const 0
call $searchReflectiveProxy
ref.cast (ref $4)
call_ref $4
```

run-time binary search lookup Θ(log n)

Other things we put in vtables

- Lazy reference to the `java.lang.Class` object representing the class, returned by `obj.getClass()`
- Class name, returned by `obj.getClass().getName()`
- Other `jl.Class`-related metadata: `isInterface()`, `isPrimitive()`, `arrayOf()`, `componentType()`, etc.
- Function pointer to a `clone()` function for instances of that class

Boxing and `asInstanceOf`

Upcasts and downcasts of classes

```
@noinline def computeResult(a: Int, b: Int): Int = {
  // no optimizer for this example
  val v3: Vec3 = new Vec3(a, b, 3)
  val v2: Vec2 = v3
  val v3Again: Vec3 = v2.asInstanceOf[Vec3]
  v3Again.x
}
```

```
(func $f...computeResult_I_I_I (type $143)
  (param $this (ref $c...Main$)) ... (result i32)
  (local $v3 (ref null $c.helloworld.Vec3))
  (local $v2 (ref null $c.helloworld.Vec2))
  (local $v3Again (ref null $c.helloworld.Vec3))
  (local $2 (ref $c.helloworld.Vec3))
  ... ; new Vec3(a, b, 3)
  local.set $v3
```

```
local.get $v3
local.set $v2
```

no cast from Vec3 to Vec2: Wasm understands our subtyping relationship

```
local.get $v2
ref.cast (ref null $c.helloworld.Vec3)
local.set $v3Again
```

cast down from Vec2 to Vec3: requires ref.cast

```
local.get $v3Again
...)
```

Q.: What happens if v2 is not, in fact, a Vec3?

Checking `ClassCastException`s

```scala
@noinline def computeResult(a: Int, b: Int): Int = {
  // no optimizer for this example
  val v3: Vec3 = new Vec3(a, b, 3)
  val v2: Vec2 = v3
  val v3Again: Vec3 = v2.asInstanceOf[Vec3]
  v3Again.x
}
```

```
(func $as.helloworld.Vec3 (type $315)
  (param $obj anyref) (result (ref null $c.helloworld.Vec3))
  block $1 (result (ref null $c.helloworld.Vec3))
    local.get $obj
    br_on_cast $1 anyref (ref null $c.helloworld.Vec3)
    global.get $d.helloworld.Vec3
    call $classCastException
    unreachable
  end)
```

```
local.get $v3
local.set $v2
local.get $v2
call  $as.helloworld.Vec3
local.set $v3Again
local.get $v3Again
...)
```

no cast from `Vec3` to `Vec2`: Wasm understands our subtyping relationship

checked cast down from `Vec2` to `Vec3`: calls helper

Upcasts and downcasts of primitives

```
@noinline def computeResult(a: Int, b: Int): Int = {
  // no optimizer for this example
  val i: Int = a + b
  val any: Any = i
  val intAgain = any.asInstanceOf[Int]
  intAgain
}
```

```
(func $f...Main$.computeResult_I_I_I (type $143)
  (param $this (ref $c...Main$))
    (param $a i32) (param $b i32) (result i32)
  (local $i i32)
  (local $any anyref)
  (local $intAgain i32)
  ... ; a + b
  local.set $i
```

Wrong attempt:

```
  local.get $i
  local.set $any
  local.get $any
  ref.cast i32
  local.set $intAgain
  ...)
```

not possible; in Wasm, i32 </: anyref

not possible; ref.cast cannot be used with a primitive type

Upcasts and downcasts of primitives

```
@noinline def computeResult(a: Int, b: Int): Int = {
  // no optimizer for this example
  val i: Int = a + b
  val any: Any = i
  val intAgain = any.asInstanceOf[Int]
  intAgain
}
```

```
(func $f...Main$.computeResult_I_I_I (type $143)
  (param $this (ref $c...Main$))
    (param $a i32) (param $b i32) (result i32)
  (local $i i32)
  (local $any anyref)
  (local $intAgain i32)
  ... ; a + b
  local.set $i
```

Fixed:

```
  local.get $i
  call $bI                 boxing
  local.set $any
  local.get $any
  call $uI                 unboxing
  local.set $intAgain
  ...)
```

Box integers

```
@noinline def computeResult(a: Int, b: Int): Int = {
  // no optimizer for this example
  val i: Int = a + b
  val any: Any = i
  val intAgain = any.asInstanceOf[Int]
  intAgain
}
```

Fixed:

```
  local.get $i
  call $bI
  local.set $any
  local.get $any
  call $uI
  local.set $intAgain
  ...)
```

boxing

unboxing

```
(func $bI (type $97)
  (param $x i32) (result (ref any))
  local.get $x
  local.get $x
  i32.const 1
  i32.shl
  i32.xor
  i32.const -2147483648
  i32.and
  if (result (ref any))
    local.get $x
    call $bIFallback
  else
    local.get $x
    ref.i31
  end)
```

Is bit 31 important?

If yes, do the slow thing

If not, use `ref.i31`

Unbox integers

```
@noinline def computeResult(a: Int, b: Int): Int = {
  // no optimizer for this example
  val i: Int = a + b
  val any: Any = i
  val intAgain = any.asInstanceOf[Int]
  intAgain
}
```

```
(func $uI (type $92)
  (param $x anyref) (result i32)
  block $1 (result anyref)
    local.get $x
    br_on_cast_fail $1 anyref (ref i31)
    i31.get_s
    return
  end
  call $uIFallback)
```

Fixed:

```
  local.get $i          }
  call $bI              } boxing
  local.set $any        }
  local.get $any        }
  call $uI              } unboxing
  local.set $intAgain   }
  ...)
```

What about the fallbacks?

```
(import "__scalaJSHelpers" "bIFallback"
  (func $bIFallback (type (func (param i32) (result (ref any))))))
(import "__scalaJSHelpers" "uIFallback"
  (func $uIFallback (type (func (param anyref) (result i32)))))
```

JavaScript code, yeah!

```
const scalaJSHelpers = {
  bIFallback: (x) => x,
  uIFallback: (x) => x,
  ...
}
```

In practice, JavaScript allocates an object on the heap.

JavaScript Interoperability

What Scala.js interop looks like

```scala
// Create the board canvas
val boardCanvas = jQuery(
    s"<canvas width='$BoardSizePx' height='$BoardSizePx'></canvas>")
val domCanvas = boardCanvas.get(0).asInstanceOf[HTMLCanvasElement]
val context = domCanvas.getContext("2d").asInstanceOf[CanvasRenderingContext2D]

// Draw a pawn
if (square.owner != NoPlayer) {
  context.fillStyle = if (square.owner == White) "white" else "black"
  context.beginPath()
  context.arc(x+HalfSquareSizePx, y+HalfSquareSizePx, PawnRadiusPx, 0, 2*Math.PI, true)
  context.fill()
}

// Configure clicks on the board
boardCanvas.click({ (event: JQueryEvent) =>
  ...
})
```

console.log call

```
@js.native @JSGlobal
object console extends js.Object {
  def log(x: Any): Unit = js.native
}

@noinline def computeResult(a: Int, b: Int): Int = {
  console.log(a + b)
  a
}



native js module class helloworld.console$ extends scala.scalajs.js.Object loadfrom global:console {
}

@hints(2) def computeResult;I;I;I(a: int, b: int): int = {
  mod:helloworld.console$["log"]((a +[int] b));
  a
}
```

console.log call in Wasm

```
native js module class helloworld.console$ extends scala.scalajs.js.Object loadfrom global:console {
}

@hints(2) def computeResult;I;I;I(a: int, b: int): int = {
  mod:helloworld.console$["log"]((a +[int] b));
  a
}
```

```
(import "__scalaJSCustomHelpers" "2"
  (func $customJSHelper.2 (type (func (param i32) (result anyref)))))

(func $f.helloworld.Main$.computeResult_I_I_I(type $87)
   (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
   local.get $a
   local.get $b
   i32.add
   call $customJSHelper.2
   drop
   local.get $a)
```

`console.log` call in Wasm + JavaScript

```
"__scalaJSCustomHelpers": {
  ...
  "2": ((x) => console.log(x)),
  ...
}
```

```
(import "__scalaJSCustomHelpers" "2"
  (func $customJSHelper.2 (type (func (param i32) (result anyref)))))

(func $f.helloworld.Main$.computeResult_I_I_I(type $87)
   (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
   local.get $a
   local.get $b
   i32.add
   call $customJSHelper.2
   drop
   local.get $a)
```

A more complex example

```
@noinline def computeResult(a: Int, b: Int): Int = {
  val arr = js.Array(a, b)
  arr.push(3)
  arr.length
}
```

```
@hints(2) def computeResult;I;I;I(a: int, b: int): int = {
  val arr: any = [a, b];
  arr["push"](3);
  arr["length"].asInstanceOf[int]
}
```

A more complex example: Wasm + JavaScript

```
(import "__scalaJSCustomHelpers" "2"
  (func $customJSHelper.2 (type (func (param i32) (param i32) (result (ref any))))))
(import "__scalaJSCustomHelpers" "3"
  (func $customJSHelper.3 (type (func (param (ref any)) (result anyref)))))
(import "__scalaJSCustomHelpers" "4"
  (func $customJSHelper.4 (type (func (param (ref any)) (result anyref)))))
(func $f.helloworld.Main$.computeResult_I_I_I(type $88)
  (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
  (local $arr (ref any))
  local.get $a
  local.get $b
  call $customJSHelper.2
  local.set $arr
  local.get $arr
  call $customJSHelper.3
  drop
  local.get $arr
  call $customJSHelper.4
  call $uI)
```

```
@hints(2) def computeResult;I;I;I(a: int, b: int): int = {
  val arr: any = [a, b];
  arr["push"](3);
  arr["length"].asInstanceOf[int]
}
```

```
  "2": ((x, x1) => [x, x1]),
  "3": ((x) => x.push(3)),
  "4": ((x) => x.length),
```

Closures

```
@noinline def computeResult(a: Int, b: Int): Int = {
  val arr = js.Array(a, b)
  arr.sort((x, y) => -Integer.compare(x, y))
  arr(0)
}
```

```
@hints(2) def computeResult;I;I;I(a: int, b: int): int = {
  val arr: any = [a, b];
  arr["sort"]((arrow-lambda<>(arg1$2: any, arg2$2: any): any = {
    val arg1: int = arg1$2.asInstanceOf[int];
    val arg2: int = arg2$2.asInstanceOf[int];
    helloworld.Main$::$anonfun$computeResult$1;I;I;I(arg1, arg2)
  }));
  arr[0].asInstanceOf[int]
}
@hints(1) static def $anonfun$computeResult$1;I;I;I(x: int, y: int): int = {
  (y -[int] x)
}
```

Closures, inlined

```
@noinline def computeResult(a: Int, b: Int): Int = {
  val arr = js.Array(a, b)
  arr.sort((x, y) => -Integer.compare(x, y))
  arr(0)
}
```

```
@hints(2) def computeResult;I;I;I(a: int, b: int): int = {
  val arr: any = [a, b];
  arr["sort"]((arrow-lambda<>(arg1$2: any, arg2$2: any): any = {
    val arg1: int = arg1$2.asInstanceOf[int];
    val arg2: int = arg2$2.asInstanceOf[int];
    (arg2 -[int] arg1)
  }));
  arr[0].asInstanceOf[int]
}
```

Closure body in Wasm

```
(func $f.helloworld.Main$.computeResult_I_I_I__c0(type $88)
  (param $__captureData (ref $87)) (param $arg1 anyref) (param $arg2 anyref) (result anyref)
  (local $arg11 i32) (local $arg21 i32)
  local.get $arg1
  call $uI
  local.set $arg11
  local.get $arg2
  call $uI
  local.set $arg21
  local.get $arg21
  local.get $arg11
  i32.sub
  call $bI)
```

unbox the parameters

body of the lambda that was written by the developer

box the result

Closure construction in Wasm + JavaScript

```
(func $f.helloworld.Main$.computeResult_I_I_I(type $92)
  (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
  (local $arr (ref any))
  local.get $a
  local.get $b
  call $customJSHelper.2
  local.set $arr
  local.get $arr
  ref.func $f.helloworld.Main$.computeResult_I_I_I__c0
  struct.new $87
  call $customJSHelper.3
  call $customJSHelper.4
  drop
  local.get $arr
  call $customJSHelper.5
  call $uI)
```

construct the JS closure via helper

```
"2": ((x, x1) => [x, x1]),
"3": ((f, d) => ((arg1$2, arg2$2) => f(d, arg1$2, arg2$2))),
"4": ((x, x1) => x.sort(x1)),
"5": ((x) => x[0]),
```

Exception Handling

Exception handling example

```scala
@noinline def computeResult(a: Int, b: Int): Int = {
  try {
    succ(a)
  } catch {
    case th: IllegalArgumentException =>
      b
  }
}

@noinline def succ(x: Int): Int = {
  if (x < 0)
    throw new IllegalArgumentException("negative")
  x + 1
}
```

Wasm exception handling

- Declare an exception "tag" with an associated payload

  ```
  (tag $exception (param payload-type))
  ```

  In our case, as an import:

  ```
  (import "__scalaJSHelpers" "JSTag" (tag $exception (param externref)))
  ```

- Throw an exception of a given tag, with the payload on the stack:

  ```
  throw $exception
  ```

- Set up an exception handler around a block (try/catch):

  ```
  try_table (result result-type) (catch $exception $handlerBlock)
    ; code block of type result-type that might throw
  end
  ```

- `$handlerBlock` should be a surrounding block whose result type matches the payload type of the given tag

Throwing

```
(func $f.helloworld.Main$.succ_I_I (type $88)
   (param $this (ref $c.helloworld.Main$)) (param $x i32) (result i32)
   (local $1 (ref $c.java.lang.IllegalArgumentException))
   local.get $x
   i32.const 0
   i32.lt_s
   if
     call $new.java.lang.IllegalArgumentException
     local.tee $1
     global.get $'negative
     call $ct.java.lang.IllegalArgumentException.<init>_Ljava.lang.String_V
     local.get $1
     extern.convert_any
     throw $exception
   end
   i32.const 1
   local.get $x
   i32.add)
```

if (x < 0)

`new Exception(...)`, store in `$1`

load exception on the stack (as payload) and throw a `$exception`

Catching

```
(func $f.helloworld.Main$.computeResult_I_I_I (type $87)
 (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
 (local $e anyref)
 block $1 (result i32)
   block $2 (result externref)
     try_table (result externref) (catch $exception $2)
       local.get $this
       local.get $a
       call $f.helloworld.Main$.succ_I_I
       br $1
     end
   end ; block $2
   any.convert_extern
   local.set $e
   local.get $e
   ref.test (ref $c.java.lang.IllegalArgumentException)
   if (result i32)
     local.get $b
   else
     local.get $e
     extern.convert_any
     throw $exception
   end
 end) ; block $1
```

try block

store caught payload in `$e`

test if is the (Scala) type of exception we are watching out for

if yes, return `$b` (our handler code)

otherwise, rethrow the exception

try/finally

```
@noinline def computeResult(a: Int, b: Int): Int = {
  val v2: Vec2 = try {
    hide(new Vec2(succ(a), b))
  } finally {
    println("finally")
  }
  v2.x
}
```

try/finally and return

```
@noinline def computeResult(a: Int, b: Int): Int = {
  val v2: Vec2 = try {
    if (b < 0)
      return 42
    hide(new Vec2(succ(a), b))
  } finally {
    println("finally")
  }
  v2.x
}
```

More Wasm exception handling

- Set up an exception handler for *any* exception tag:

```
try_table (result result-type) (catch_all_ref $handlerBlock)
  ; code block of type result-type that might throw
end
```

- `$handlerBlock` should be a surrounding block whose result type matches the special type `exnref`
- Rethrow a caught `exnref` that is on top of the stack

```
throw_ref
```

try/finally

```
(func $f.helloworld.Main$.computeResult_I_I_I (type $87)
  (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
  (local $1 (ref null $c.helloworld.Vec2)) ... (local $v2 (ref null $c.helloworld.Vec2))
  block $1
    block $2 (result exnref)
      try_table (catch_all_ref $2)
        ... ; body of try, leaves a (ref null $c.helloworld.Vec2) on the stack
        local.set $1
      end
      ref.null exn
    end ; block $2
    local.get $this
    global.get $'finally
    any.convert_extern
    call $f.helloworld.Main$.println_Ljava.lang.Object_V
    br_on_null $1
    throw_ref
  end ; block $1
  local.get $1
  local.set $v2
  local.get $v2
  struct.get $c.helloworld.Vec2 $f.helloworld.Vec2.x)
```

store successful result in local

put a `null exnref` on the stack

`try` block

body of the `finally` block
(all run on top of the `exnref` left on the stack)

if the exnref was not null, rethrow it

if we got out without rethrowing, load back the temp local on the stack

code after the `try/finally`

`try/finally` and `return`: IR

```
@hints(2) def computeResult;I;I;I(a: int, b: int): int = {
  _return[int]: {
    val v2: helloworld.Vec2 = try {
      if ((b <[int] 0)) {
        return@_return 42
      };
      this.hide;Ljava.lang.Object;Ljava.lang.Object(
        new helloworld.Vec2().<init>;I;I;V(this.succ;I;I(a), b)
      ).asInstanceOf[helloworld.Vec2]
    } finally {
      this.println;Ljava.lang.Object;V("finally")
    };
    v2.x;I()
  }
}
```

labeled block with a result type

return from the labeled block with a result

Labeled blocks and returns of the Scala.js IR are semantically equivalent to Wasm block's and br's
(they were independently and concurrently invented, remarkably)

try/finally and return

```
(func $f.helloworld.Main$.computeResult_I_I_I (type $87)
 (param $this (ref $c.helloworld.Main$)) (param $a i32) (param $b i32) (result i32)
 (local $1 (ref null $c.helloworld.Vec2)) ... (local $v2 (ref null $c.helloworld.Vec2))
 block $_result (result i32)
  block $1
   block $2 (result exnref)
    try_table (catch_all_ref $2)
     ... ; body of try
      i32.const 42
      br $_result
     local.set $1
    end
    ref.null exn
   end ; block $2
   ... ; body of finally
   br_on_null $1
   throw_ref
  end ; block $1
  local.get $1
  ... ; code after try/finally
 end) ; block $_result
```

block for the labeled block

return from the block with a result

Not so simple! This is wrong!
br bypasses the finally block

Find out what we actually do by reading this big comment, if you dare.

Conclusion

Conclusion

- From an architecture point of view,
  a real compiler to Wasm is quite similar to what you're doing in the project
- but there are lots and lots more "stuff" to take care of

Topics we covered:

- Object Model and virtual dispatch
- Boxing
- JavaScript Interoperability
- Exception Handling