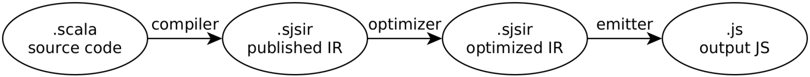
# Foundations of Software
# Fall 2023

Week 14

Sébastien Doeraene

# Elements of the Scala.js IR type system

# Scala.js compilation pipeline



# Why formally study an IR

# Why formally study an IR

- Optimizations may only be applicable if the type tystem is sound
- Prove that certain optimizations are correct
- Prove that the translation from source and to the target language are correct
- etc.

# Mixing primitives and objects

# Motivation

Featherweight Java only has objects. How do we model primitives, for example, `int` and `bool`?

# Motivation

Featherweight Java only has objects. How do we model primitives, for example, `int` and `bool`?

Moreover, in Scala, primitive types are "object-like". We can use them in arbitrary type parameters, and they should behave like objects.
On the JVM, this is implemented with *boxing*. In Scala.js, however, boxing would be detrimental to *interoperability* with JavaScript.
How do we make primitives object-like without boxing?

# Motivation

Featherweight Java only has objects. How do we model primitives, for example, `int` and `bool`?

Moreover, in Scala, primitive types are "object-like". We can use them in arbitrary type parameters, and they should behave like objects.
On the JVM, this is implemented with *boxing*. In Scala.js, however, boxing would be detrimental to *interoperability* with JavaScript.
How do we make primitives object-like without boxing?

Idea: make primitive types *subtypes* of their "representative classes".

# Types and subtyping

| | | |
|---|---|---|
| $\mathtt{T}$ ::= | | *types* |
| | $\mathtt{C}$ | *class* |
| | $\mathtt{int}$ | *primitive int* |
| | $\mathtt{bool}$ | *primitive bool* |

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}}}{\mathtt{C} <: \mathtt{D}}$$

$$\mathtt{T} <: \mathtt{T}$$

$$\frac{\mathtt{S} <: \mathtt{W} \qquad \mathtt{W} <: \mathtt{T}}{\mathtt{S} <: \mathtt{T}}$$

$\mathtt{int} <: \mathtt{Integer}$ $\qquad$ $\mathtt{bool} <: \mathtt{Boolean}$

# Representative classes

$$\mathit{tpcls}(\mathtt{C}) = \mathtt{C}$$

$$\mathit{tpcls}(\mathtt{int}) = \mathtt{Integer}$$

$$\mathit{tpcls}(\mathtt{bool}) = \mathtt{Boolean}$$

$$\mathtt{T} <: \mathit{tpcls}(\mathtt{T})$$

# Syntax (terms)

| | | |
|---|---|---|
| `t ::=` | | *terms* |
| | `x` | *variable* |
| | `t.f` | *field access* |
| | $\mathtt{t.m(}\overline{\mathtt{t}}\mathtt{)}$ | *method invocation* |
| | $\mathtt{new\ C(}\overline{\mathtt{t}}\mathtt{)}$ | *object creation* |
| | `(T) t` | *cast* |
| | `false` | |
| | `true` | |
| | `if t then t else t` | |
| | `0` | |
| | `succ t` | |
| | `pred t` | |
| | `iszero t` | |

# Syntax (values)

| | | |
|---|---|---|
| `v` ::= | | *values* |
| | `new C(`$\overline{\texttt{v}}$`)` | *object creation* |
| | `nv` | *numeric value* |
| | `bv` | *boolean value* |
| `nv` ::= | | *numeric values* |
| | `0` | *zero* |
| | `succ nv` | *non-zero* |
| `bv` ::= | | *boolean values* |
| | `false` | *false* |
| | `true` | *true* |

# Typing rules: method calls

Adapting from Featherweight Java:

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{S}} \to \mathtt{T} \qquad \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{S}_1} \qquad \overline{\mathtt{S}_1} <: \overline{\mathtt{S}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) : \mathtt{T}} \quad \text{(T-Invk)}$$

What if $\mathtt{t}_0$ is a primitive?

# Typing rules: method calls

Adapting from Featherweight Java:

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{S}} \to \mathtt{T} \qquad \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{S}_1} \qquad \overline{\mathtt{S}_1} <: \overline{\mathtt{S}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) : \mathtt{T}} \quad \text{(T-Invk)}$$

What if $\mathtt{t}_0$ is a primitive?

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathit{tpcls}(\mathtt{T}_0)) = \overline{\mathtt{S}} \to \mathtt{T} \qquad \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{S}_1} \qquad \overline{\mathtt{S}_1} <: \overline{\mathtt{S}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) : \mathtt{T}} \quad \text{(T-Invk)}$$

If $\Gamma \vdash \mathtt{x} : \mathtt{int}$, the call `x.m(...)` is typed by looking up `m` in `Integer`.

# Example

```
class Boolean extends Object { Boolean() { super(); } }
class Integer extends Object {
  Integer() { super(); }
  int plus(int that) {
    return if (iszero that) then ((int) this)
           else (succ this.plus(pred that)); }
}
class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd; }
  int sum() {
    return ((int) this.fst).plus((int) this.snd); }
}

new Pair(5, 11).sum()
```

# Typing rules: fields

Adapting from Featherweight Java:

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{f}_i : \mathtt{T}_i} \qquad \text{(T-Field)}$$

What if $\mathtt{t}_0$ is a primitive?

# Typing rules: fields

Adapting from Featherweight Java:

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{t}_0\mathtt{.f}_i : \mathtt{T}_i} \qquad \text{(T-Field)}$$

What if $\mathtt{t}_0$ is a primitive?

We can't have that!

# Typing rules: fields

Adapting from Featherweight Java:

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{f}_i : \mathtt{T}_i} \qquad \text{(T-FIELD)}$$

What if $\mathtt{t}_0$ is a primitive?

We can't have that!

Add additional well-formedness conditions for representative classes:

$$\frac{\mathit{fields}(\mathtt{Integer}) = \emptyset \qquad \mathit{fields}(\mathtt{Boolean}) = \emptyset}{\mathtt{repr\ classes\ OK}}$$

# Typing rules: casts

Straightforward generalization to all types.

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T}}{\Gamma \vdash (\mathtt{T})\mathtt{t}_0 : \mathtt{T}} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{S} \qquad \mathtt{T} <: \mathtt{S} \qquad \mathtt{T} \neq \mathtt{S}}{\Gamma \vdash (\mathtt{T})\mathtt{t}_0 : \mathtt{T}} \qquad \text{(T-DCast)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{S} \qquad \mathtt{T} \not<: \mathtt{S} \qquad \mathtt{S} \not<: \mathtt{T} \qquad \textit{stupid warning}}{\Gamma \vdash (\mathtt{T})\mathtt{t}_0 : \mathtt{T}} \qquad \text{(T-SCast)}$$

# Typing rules: casts

Since it is an Intermediate Representation, warnings are not relevant anymore. Therefore, we keep only one typing rule for casts.

$$\frac{\Gamma \vdash \mathtt{t_0} : \mathtt{S}}{\Gamma \vdash \mathtt{(T)t_0} : \mathtt{T}} \qquad \text{(T-Cast)}$$

# Typing rules: casts

Since it is an Intermediate Representation, warnings are not relevant anymore. Therefore, we keep only one typing rule for casts.

$$\frac{\Gamma \vdash \mathtt{t_0 : S}}{\Gamma \vdash \mathtt{(T)t_0 : T}} \qquad \text{(T-Cast)}$$

Question: can we remove the premise of that rule?

# Evaluation rules

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})).\mathtt{f}_i \longrightarrow \mathtt{v}_i} \qquad \text{(E-ProjNew)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathit{tpcls}(\mathit{vtpe}(\mathtt{v}))) = (\overline{\mathtt{x}}, \mathtt{t}_0)}{\mathtt{v.m}(\overline{\mathtt{u}}) \longrightarrow [\overline{\mathtt{x}} \mapsto \overline{\mathtt{u}}, \mathtt{this} \mapsto \mathtt{v}]\mathtt{t}_0} \qquad \text{(E-InvkVal)}$$

$$\frac{\mathit{vtpe}(\mathtt{v}) <: \mathtt{T}}{(\mathtt{T})\mathtt{v} \longrightarrow \mathtt{v}} \qquad \text{(E-CastVal)}$$

$$\mathit{vtpe}(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})) = \mathtt{C} \qquad \mathit{vtpe}(\mathtt{nv}) = \mathtt{int} \qquad \mathit{vtpe}(\mathtt{bv}) = \mathtt{bool}$$

plus congruence rules and rules for `if`, `pred`, `succ` and `iszero` (omitted)

# Labeled blocks

# Presentation

In JavaScript, we have labeled *statements* with `break`s:

```
label: {
  ...
  if (x)
    break label;
  ...
}
```

If execution reaches `break label`, it jumps to *after* the block.

# Presentation

We generalize the concept to *expressions*. A `return` to a label jumps out of the block, resulting in the specified value as the value of the block.

```
val y: T = label[T]: {
  ...
  if (x)
    return@label someT;
  ...
  someOtherT
}
```

# Use cases: modeling `return`

Traditional `return`:

```
def foo(x: int): int = {
  if (x < 0) {
    return -x
  }
  x
}
```

Modeled as:

```
def foo(x: int): int = {
  ret[int]: {
    if (x < 0) {
      return@ret -x
    }
    x
  }
}
```

# Use cases: modeling `break` and `continue`

Traditional `break` and `continue`:

```
def foo(x: int): unit = {
  var i: int = x
  while (i > 0) {
    if (i % 3 == 0)
      continue
    if (i % 10 == 0)
      break
    println(i)
    i = i + 1
  }
}
```

# Use cases: modeling `break` and `continue`

Modeled as two nested labels:

```
def foo(x: int): unit = {
  var i: int = x
  breakLoop[unit]: {
    while (i > 0) {
      continueLoop[unit]: {
        if (i % 3 == 0)
          return@continueLoop unit
        if (i % 10 == 0)
          return@breakLoop unit
        println(i)
        i = i + 1
      }
    }
  }
}
```

# Use cases: encoding of tail recursion

Tail recursive function in source code:

```
def fact(n: int, acc: int): int = {
  if (n == 0) acc
  else fact(n - 1, n * acc)
}
```

# Use cases: encoding of tail recursion

Encoding with a creative use of labeled blocks:

```
def fact(var n: int, var acc: int): int = {
  ret[int]: {
    while (true) {
      tailcall[unit]: {
        return@ret {
          if (n == 0) acc
          else {
            val n’ = n - 1
            val acc’ = n * acc
            n = n’
            acc = acc’
            return@tailcall unit
          }
        }
      }
    }
  }
```

# Use cases: encoding of tail recursion

Encoding with a creative use of labeled blocks and `loop`:

```
def fact(var n: int, var acc: int): int = {
  ret[int]: {
    loop {
      tailcall[unit]: {
        return@ret {
          if (n == 0) acc
          else {
            val n' = n - 1
            val acc' = n * acc
            n = n'
            acc = acc'
            return@tailcall unit
          }
        }
      }
    }
  }
```

# Formalization

On the board

# Evaluation rules

$$\texttt{loop t}_1 \mid \mu \longrightarrow \texttt{t}_1\texttt{; loop t}_1 \mid \mu \qquad \text{(E-Loop)}$$

$$\alpha\,\texttt{[T] \{v}_1\texttt{\}} \mid \mu \longrightarrow \texttt{v}_1 \mid \mu \quad \text{(E-LabeledValue)}$$

$$\alpha\,\texttt{[T] \{return@}\alpha\ \texttt{v}_1\texttt{\}} \mid \mu \longrightarrow \texttt{v}_1 \mid \mu \quad \text{(E-LabeledRetMatch)}$$

$$\frac{\beta \neq \alpha}{\alpha\,\texttt{[T] \{return@}\beta\ \texttt{v}_1\texttt{\}} \mid \mu \longrightarrow \texttt{return@}\beta\ \texttt{v}_1 \mid \mu} \quad \text{(E-LabeledRetDiff)}$$

Plus congruence rules and propagation rules for `return`, for example:

$$\texttt{(return@}\alpha\ \texttt{v}_1\texttt{) t}_2 \mid \mu \longrightarrow \texttt{return@}\alpha\ \texttt{v}_1 \mid \mu \quad \text{(E-AppRet1)}$$

$$\texttt{(return@}\alpha\ \texttt{v}_1\texttt{); t}_2 \mid \mu \longrightarrow \texttt{return@}\alpha\ \texttt{v}_1 \mid \mu \quad \text{(E-SeqRet)}$$

# Typing rules

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash \mathtt{t_1} : \mathtt{Unit}}{\Gamma \mid \Delta \mid \Sigma \vdash \mathtt{loop}\ \mathtt{t_1} : \mathtt{nothing}} \qquad \text{(T-Loop)}$$

$$\frac{\Gamma \mid \Delta, \alpha : \mathtt{T} \mid \Sigma \vdash \mathtt{t_1} : \mathtt{T}}{\Gamma \mid \Delta \mid \Sigma \vdash \alpha\mathtt{[T]}\ \mathtt{\{t_1\}} : \mathtt{T}} \qquad \text{(T-Labeled)}$$

$$\frac{\alpha : \mathtt{T_1} \in \Delta \qquad \Gamma \mid \Delta \mid \Sigma \vdash \mathtt{t_1} : \mathtt{T_1}}{\Gamma \mid \Delta \mid \Sigma \vdash \mathtt{return@}\alpha\ \mathtt{t_1} : \mathtt{nothing}} \qquad \text{(T-Return)}$$

# Proofs

On the board

# Typing rules, fixed

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash \texttt{t}_1 : \texttt{Unit}}{\Gamma \mid \Delta \mid \Sigma \vdash \texttt{loop t}_1 : \texttt{nothing}} \quad \text{(T-Loop)}$$

$$\frac{\Gamma \mid \Delta, \alpha : \texttt{T} \mid \Sigma \vdash \texttt{t}_1 : \texttt{T}}{\Gamma \mid \Delta \mid \Sigma \vdash \alpha\texttt{[T] \{t}_1\texttt{\}} : \texttt{T}} \quad \text{(T-Labeled)}$$

$$\frac{\alpha : \texttt{T}_1 \in \Delta \qquad \Gamma \mid \Delta \mid \Sigma \vdash \texttt{t}_1 : \texttt{T}_1}{\Gamma \mid \Delta \mid \Sigma \vdash \texttt{return@}\alpha\ \texttt{t}_1 : \texttt{nothing}} \quad \text{(T-Return)}$$

$$\frac{\Gamma, \texttt{x}:\texttt{T}_1 \mid \emptyset \mid \Sigma \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \mid \Delta \mid \Sigma \vdash \lambda\texttt{x}:\texttt{T}_1\texttt{.t}_2 : \texttt{T}_1 \to \texttt{T}_2} \quad \text{(T-Abs)}$$