

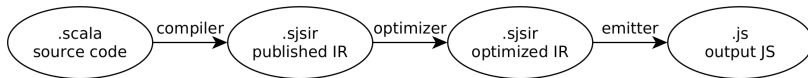
Foundations of Software Fall 2022

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 system 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

$T ::=$

C

int

bool

types

class

primitive int

primitive bool

$$\frac{CT(C) = \text{class } C \text{ extends } D \{ \dots \}}{C <: D}$$

$T <: T$

$$\frac{S <: W \quad W <: T}{S <: T}$$

$\text{int} <: \text{Integer}$

$\text{bool} <: \text{Boolean}$

Representative classes

$$tpcls(C) = C$$

$$tpcls(int) = Integer$$

$$tpcls(bool) = Boolean$$

$$T <: tpcls(T)$$

Syntax (terms)

$t ::=$

x

$t.f$

$t.m(\bar{t})$

$\text{new } C(\bar{t})$

$(T) \ t$

false

true

$\text{if } t \text{ then } t \text{ else } t$

0

$\text{succ } t$

$\text{pred } t$

$\text{iszero } t$

terms

variable

field access

method invocation

object creation

cast

Syntax (values)

`v ::=`

`new C(\bar{v})`

`nv`

`bv`

values

object creation

numeric value

boolean value

`nv ::=`

`0`

`succ nv`

numeric values

zero

non-zero

`bv ::=`

`false`

`true`

boolean values

false

true

Typing rules: method calls

Adapting from Featherweight Java:

$$\frac{\begin{array}{l} \Gamma \vdash t_0 : C_0 \\ mtype(m, C_0) = \bar{S} \rightarrow T \\ \Gamma \vdash \bar{t} : \bar{S}_1 \quad \bar{S}_1 <: \bar{S} \end{array}}{\Gamma \vdash t_0.m(\bar{t}) : T}$$

(T-INVK)

What if t_0 is a primitive?

Typing rules: method calls

Adapting from Featherweight Java:

$$\frac{\begin{array}{l} \Gamma \vdash t_0 : C_0 \\ \text{mtype}(m, C_0) = \bar{S} \rightarrow T \\ \Gamma \vdash \bar{t} : \bar{S}_1 \quad \bar{S}_1 <: \bar{S} \end{array}}{\Gamma \vdash t_0.m(\bar{t}) : T} \quad (\text{T-INVK})$$

What if t_0 is a primitive?

$$\frac{\begin{array}{l} \Gamma \vdash t_0 : T_0 \\ \text{mtype}(m, \text{tpcls}(T_0)) = \bar{S} \rightarrow T \\ \Gamma \vdash \bar{t} : \bar{S}_1 \quad \bar{S}_1 <: \bar{S} \end{array}}{\Gamma \vdash t_0.m(\bar{t}) : T} \quad (\text{T-INVK})$$

If $\Gamma \vdash x : \text{int}$, the call $x.m(\dots)$ 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:

$$\frac{\Gamma \vdash t_0 : C_0 \quad \text{fields}(C_0) = \bar{T} \ \bar{f}}{\Gamma \vdash t_0.f_i : T_i} \quad (\text{T-FIELD})$$

What if t_0 is a primitive?

Typing rules: fields

Adapting from Featherweight Java:

$$\frac{\Gamma \vdash t_0 : C_0 \quad \text{fields}(C_0) = \bar{T} \ \bar{f}}{\Gamma \vdash t_0.f_i : T_i} \quad (\text{T-FIELD})$$

What if t_0 is a primitive?

We can't have that!

Typing rules: fields

Adapting from Featherweight Java:

$$\frac{\Gamma \vdash t_0 : C_0 \quad \text{fields}(C_0) = \bar{T} \ \bar{f}}{\Gamma \vdash t_0.f_i : T_i} \quad (\text{T-FIELD})$$

What if t_0 is a primitive?

We can't have that!

Add additional well-formedness conditions for representative classes:

$$\frac{\text{fields}(\text{Integer}) = \emptyset \quad \text{fields}(\text{Boolean}) = \emptyset}{\text{repr classes OK}}$$

Typing rules: casts

Straightforward generalization to all types.

$$\frac{\Gamma \vdash t_0 : S \quad S <: T}{\Gamma \vdash (T)t_0 : T} \quad (\text{T-UCAST})$$

$$\frac{\Gamma \vdash t_0 : S \quad T <: S \quad T \neq S}{\Gamma \vdash (T)t_0 : T} \quad (\text{T-DCAST})$$

$$\frac{\Gamma \vdash t_0 : S \quad T \not<: S \quad S \not<: T \quad \textit{stupid warning}}{\Gamma \vdash (T)t_0 : T} \quad (\text{T-SCAST})$$

Typing rules: casts

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

$$\frac{\Gamma \vdash t_0 : S}{\Gamma \vdash (T)t_0 : T} \quad (\text{T-CAST})$$

Typing rules: casts

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

$$\frac{\Gamma \vdash t_0 : S}{\Gamma \vdash (T)t_0 : T} \quad (\text{T-CAST})$$

Question: can we remove the premise of that rule?

Evaluation rules

$$\frac{fields(C) = \bar{T} \ \bar{f}}{(new \ C(\bar{v})) . f_i \longrightarrow v_i} \quad (E-PROJNEW)$$

$$\frac{mbody(m, tpcls(vtpe(v))) = (\bar{x}, t_0)}{v.m(\bar{u}) \longrightarrow [\bar{x} \mapsto \bar{u}, this \mapsto v]t_0} \quad (E-INVKVAL)$$

$$\frac{vtpe(v) <: T}{(T)v \longrightarrow v} \quad (E-CASTVAL)$$

$$vtpe(new \ C(\bar{v})) = C \quad vtpen(v) = int \quad vtpen(bv) = bool$$

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

Labeled blocks

Presentation

In JavaScript, we have labeled *statements* with `breaks`:

```
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

Typing rules

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Delta \mid \Sigma \vdash \text{loop } t_1 : \text{nothing}} \quad (\text{T-LOOP})$$

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

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

Evaluation rules

$$\text{loop } t \mid \mu \longrightarrow t; \text{ loop } t \mid \mu \quad (\text{E-LOOP})$$

$$\alpha[T] \{v_1\} \mid \mu \longrightarrow v_1 \mid \mu \quad (\text{E-LABELEDVALUE})$$

$$\alpha[T] \{\text{return}@_\alpha v_1\} \mid \mu \longrightarrow v_1 \mid \mu \quad (\text{E-LABELEDRETMATCH})$$

$$\frac{\beta \neq \alpha}{\alpha[T] \{\text{return}@_\beta v_1\} \mid \mu \longrightarrow \text{return}@_\beta v_1 \mid \mu} \quad (\text{E-LABELEDRETDIFF})$$

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

$$(\text{return}@_\alpha v_1) \ t_2 \mid \mu \longrightarrow \text{return}@_\alpha v_1 \mid \mu \quad (\text{E-APPRET1})$$

$$(\text{return}@_\alpha v_1); t_2 \mid \mu \longrightarrow \text{return}@_\alpha v_1 \mid \mu \quad (\text{E-SEQRET})$$

Proofs

On the board

Typing rules, fixed

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Delta \mid \Sigma \vdash \text{loop } t_1 : \text{nothing}} \quad (\text{T-LOOP})$$

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

$$\frac{\alpha : T_1 \in \Delta \quad \Gamma \mid \Delta \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Delta \mid \Sigma \vdash \text{return}@_\alpha t_1 : \text{nothing}} \quad (\text{T-RETURN})$$

$$\frac{\Gamma, x : T_1 \mid \emptyset \mid \Sigma \vdash t_2 : T_2}{\Gamma \mid \Delta \mid \Sigma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$