Late Data Layout: Unifying Data Representation Transformations (OOPSLA'14)

Vlad Ureche Eugene Burmako Martin Odersky

École polytechnique fédérale de Lausanne, Switzerland

`{first.last}@epfl.ch` `scala-ldl.org`

① Data Representation Problem

Data takes various representations when interacting with different language features. For example, the value `5` in:

Source Code
```
val a: Int = 5
val b: Any = 5
```

is compiled to two different representations:

Low-level Bytecode
```
val a: int = 5                 // unboxed: int
val b: Object = new Integer(5) // boxed:   Integer
```

Scala abstracts over the boxed and unboxed representations by exposing a single `Int` type. This simplifies the language but complicates the compiler, which must:

- choose the representation of each value and
- introduce coercions such as boxing and unboxing

We will further explore how this is implemented in a compiler.

Other language features abstract over data representations as well, requiring similar transformations:

- value classes: inline C-like structures vs boxed objects
- miniboxing: long integer encoding vs boxed values
- staging: immediate `5` vs next-stage values `2 + 3`

② Syntax-Driven Transformation

`Int → int` The unboxed representation is more efficient, so values are unboxed on a case by case basis, if possible.

`Int → Integer` After unboxing occurs, all remaining values of type `Int` can be converted to `Integer`, since the semantics of `Int` correspond to the runtime behavior of `Integer`.

Source Code
```
val c: Int = ...    // e.g.: List[Int](1,2,3).head
val d: Int = c
println(d.toString)
```

When unboxing a value, such as `c` or `d`, coercions are introduced to maintain representation consistency:

Transformed Code (Step 1: unboxing c)
```
val c: int = unbox(...)     // unboxing the rhs of c
val d: Int = box(c)         // boxing all references
println(d.toString)
```

Transformed Code (Step 2: unboxing d)
```
val c: int = unbox(...)
val d: int = unbox(box(c)) // unboxing the rhs of d
println(box(d).toString)   // boxing all references
```

Syntax-driven transformations produce redundant coercions, which slow down execution, for example in the definition of `d`.

For simple cases, peephole optimizations can eliminate the redundant coercions. Yet, as shown in the paper, they do not scale to more complex cases. A better approach is necessary.

③ Type-Driven LDL Transformation

Instead of using syntax-based rules, Late Data Layout (LDL) injects representation information in the types. It then inserts coercions when types (and thus representations) don't match:

Step 1 Inject annotations that track the representation:

Inject phase
```
val c: @unboxed Int = ...
val d: @unboxed Int = c
println(d.toString)
```

Step 2 Coerce only when representations do not match:

Coerce phase
```
// expected @unboxed Int, found Int ⇒ add coercion:
val c: @unboxed Int = unbox(...)
// expected @unboxed Int, found @unboxed Int ⇒ ok:
val d: @unboxed Int = c
// expected Int, found @unboxed Int ⇒ add coercion:
print(box(d).toString)
```

The expected type propagation (part of local type inference) allows tracking the required representation of each expression, enabling the introduction of coercions only where necessary.

Step 3 Commit to the final representation, by replacing annotated types by their target representations:

Commit phase
```
val c: int = unbox(...)
val d: int = c // optimal!
println(box(d).toString)
```

The Late Data Layout transformation has three properties:

- consistency, guaranteed by the type system
- selectivity, thanks to individual value annotation
- optimality, by virtue of expected type propagation

④ Benchmarks

Performance gains For the three Scala plugins we developed:

- value class inlining transformation, up to 2x
- miniboxing generics, speedups of up to to 22x
- staging FFT calculations, speedups of up to 59x

Development time The LDL transformation took 2 months to develop for the miniboxing plugin, and was subsequently used to implement the other two plugins from scratch:

- value class plugin: 2 developer-weeks of coding
- staging plugin: 1 developer-week of coding

⑤ Resources

- Official website: `scala-ldl.org`
- OOPSLA'14 paper: `doi>10.1145/2660193.2660197`