MINIBOXING: Almost-free Bytecode Specialization (OOPSLA'13)

Vlad Ureche and **Cristian Talau** and **Martin Odersky**
École polytechnique fédérale de Lausanne, Switzerland
{first.last}@epfl.ch scala-miniboxing.org

① Problem

Generics on the Java Virtual Machine are erased, and this leads to inefficient code:

Source Code

```
def identity[T](t: T): T = t
identity(3)
```

Erased Code

```
def identity(t: Object): Object = t
unbox(identity(box(3)))
```

Boxing (storing values as heap objects) affects performance:

- indirect access to values
- large heap footprint
- more garbage collection cycles

Load-time bytecode specialization (like .NET) is not possible for two reasons:

- the bytecode does not contain generic type information
- the Java Virtual Machine does not carry reified generics

Naive specialization is not a solution, as it produces 10^n variants of the code, where n is the number of type parameters:

Specialized Code

```
def identity(t: Object): Object = t
def identity(t: Unit): Unit = t
def identity(t: Boolean): Boolean = t
def identity(t: Char): Char = t
...
```

② Tagged Union

Tagged union can help reduce the specialized variant count:

Equivalent Code with Tagged Union

```
def identity(t: Tagged): Tagged = t
identity(Tagged(INT, 3))
```

But this has two problems:

- tags need to be carried with values (including in arrays)
- dipatching on tags takes time and prevents optimizations:

Source Code

```
(t: Tagged).toString
```

Equivalent Code

```
t.tag match {
    case INT => t.value.toInt.toString
}
```

③ Miniboxing

Miniboxing resolves the two problems in tagged union:

Carrying tags: Since Scala and Java have static type systems, **miniboxing can attach tags to code** instead of values:

Equivalent Code with Miniboxing

```
def identity[T](T_Tag: Tag, t: Long): Long = t
identity(INT, 3)
```

Dispatching cost: Miniboxing specializes the code using a **three-phase load-time transform**:

① creating copies of the class with tags set statically

② constant propagation on the bytecode:

Equivalent Code

```
INT match {
   case INT => t.value.toInt.toString
}
```

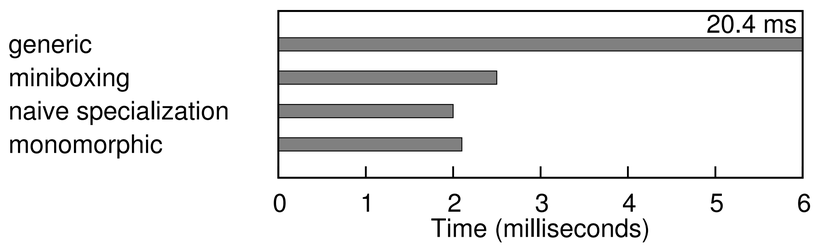
③ dead code elimination on the bytecode:

Equivalent Code

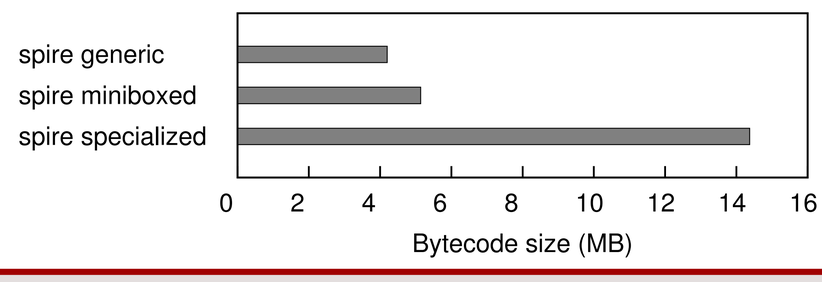
```
INT match {
   case INT => t.value.toInt.toString
}
```

④ Benchmarks

Performance is on-par with naive specialization (array buffer reverse microbenchmark):



Code size increases by 20% and is 5x less than naive specialization (on a 12KLOC numeric abstractions library):



⑤ Resources

- Official website: scala-miniboxing.org
- OOPSLA'13 paper: doi>10.1145/2509136.2509537