UNBOXING WITH IFOS

TOMAS TAUBER

January 22, 2015

OUTLINE

- **Motivation**: JVM, boxed values of primitives
- **Specialization**
- **Mini-boxing**
- **New approach with IFOs**
- **Other related work + Conclusion/Q&A**

GENERICS IN JAVA

```
class MyLovely<Horse> {
  Horse genericEurovisionSong;
}
```

Type erasure

```
class MyLovely {
  Object genericEurovisionSong;
}
```

BOXING

- **Primitive types** (values pushed on JVM stack): int, long, short, boolean, byte, char, float, double

```
int x = 1;
```

- **Reference types** (Objects; references pushed on JVM stack): Integer, Long, Short, Boolean, Byte, Character, Float, Double

```
Integer x = 1;   →   Integer x = new Integer(1);
```

PERFORMANCE PENALTY

- More heap memory consumed
- Indirect access (chase reference, call method, …)
- Need to be GCed (doesn’t go away immediately when out of scope)
- May break locality

IDEA 1: SPECIALIZATION

- We can generate special class/method for every primitive type, for example:

$$id \equiv \Lambda\alpha\lambda(x : \alpha).x$$

→

```
Object id(Object x) { return x;}
int id(int x) { return x;}
long id(long x) { return x;}
double id(double x) { return x;}
boolean id(boolean x) { return x;}
      ...
      ...
```

SPECIALIZATION: THE GOOD

- Fairly simple transformation (duplicate and adapt original code for every primitive type)
- No performance penalty for boxing
- In many compilers, e.g. main-line stable Scala
- Partially implemented in the current compiler version (-m Unbox): ClosureIntInt, ClosureBoxInt, …

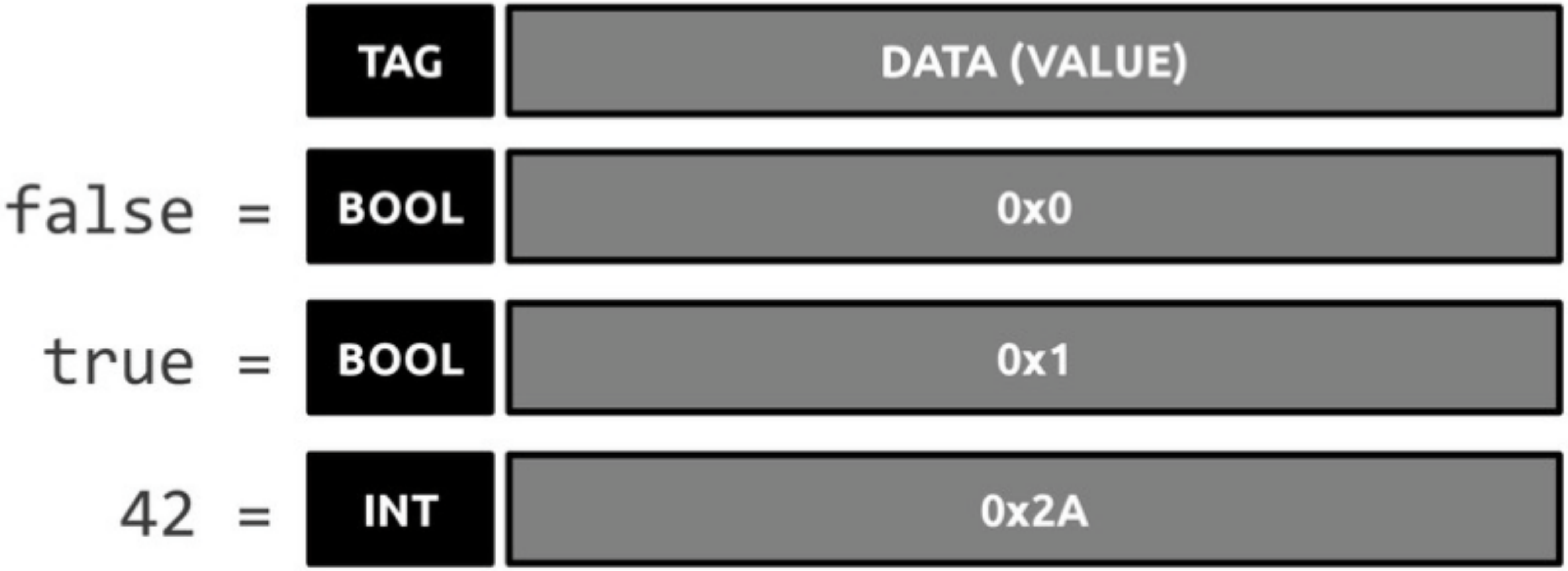
SPECIALIZATION:THE BAD

- Consider f :: A -> B -> C
- We have:
  - 1 reference type (Object)
  - 8 primitive types
  - 1 unit type (Void)

1000 different Function classes

IDEA 2: MINI-BOXING

- Encode every primitive type as long and store its original type as tag:



	TAG	DATA (VALUE)
false =	BOOL	0x0
true =	BOOL	0x1
42 =	INT	0x2A

MINI-BOXING

- We generate only two versions:

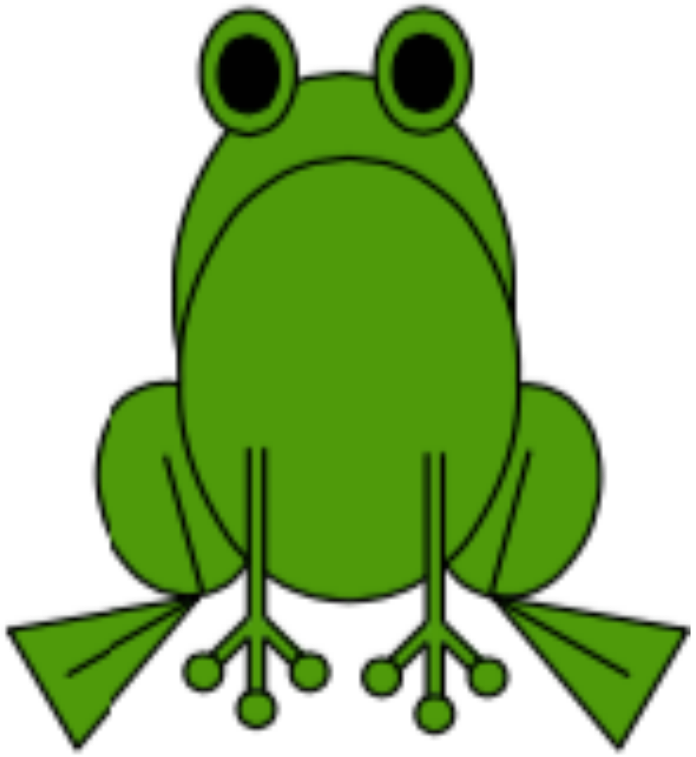
$$id \equiv \Lambda\alpha\lambda(x : \alpha).x$$

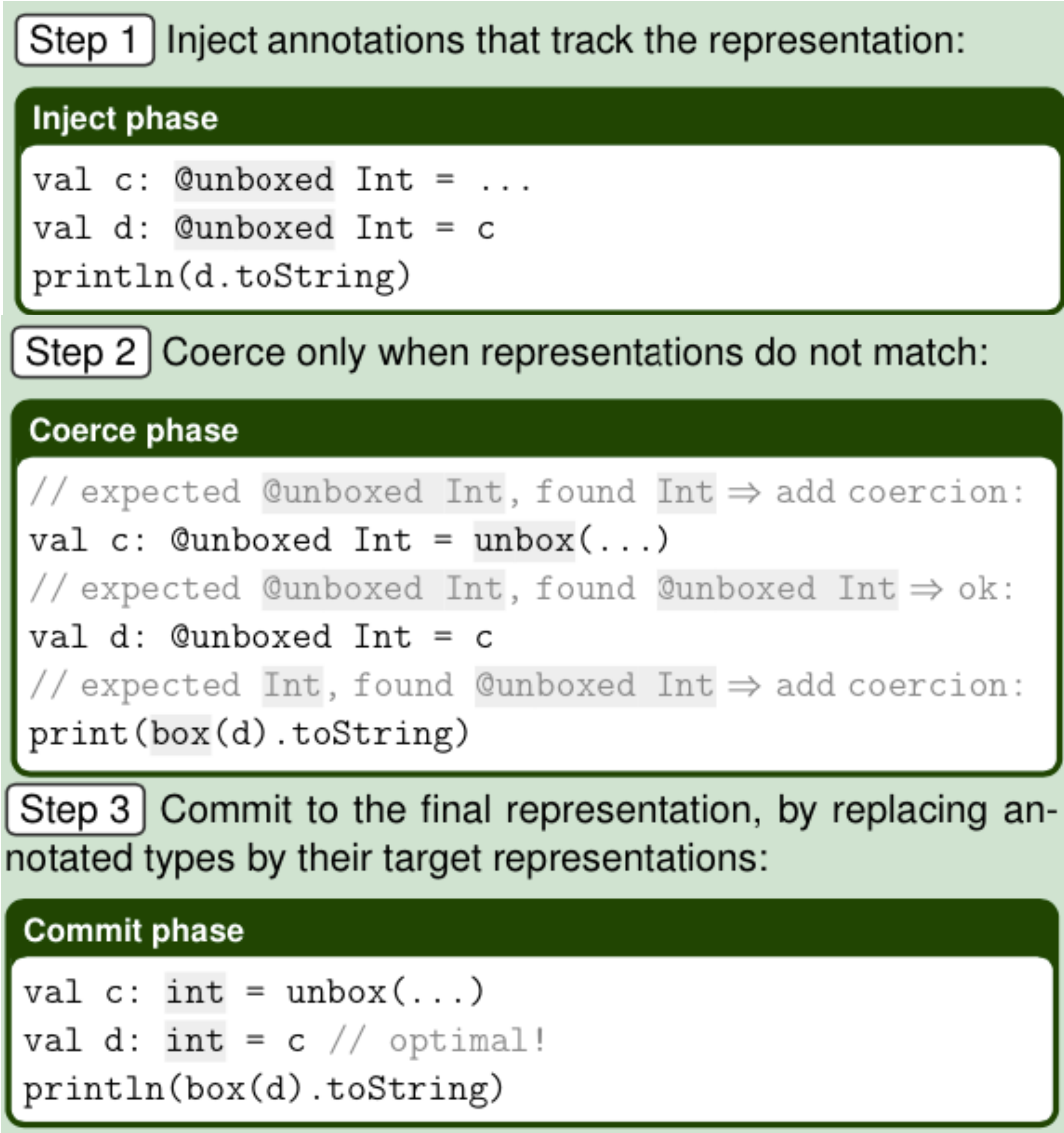
```
Object id(Object x) { return x;}
long id(byte TAG, long x) { return x;}
```

- In the 3 argument case, we generate 8 versions (instead of 1000)

- Transformation is a little bit more complex, but you already know it

TRANSFORMATION: LDL





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)
```

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)
```

MINI-BOXING: FINAL NOTES

- Probably the best option if you care about strong Java interoperability, sub-typing etc.

- We don't, plus:

    - It still has the exponential growth of classes (2^n instead of 10^n)

    - It requires type tagging + sophisticated local type inference for their propagation

NEW APPROACH WITH IFOS

```
public abstract class UFunction {

  Object oarg;
  long parg;
  Object ores;
  long pres;
  abstract void apply();

}
```

MONOMORPHIC FUNCTION

$$inc \equiv \lambda(x : Long).x + 1$$

```
UFunction uinc = new UFunction() {
   void apply() { pres = parg+1; }
}
```

POLYMORPHIC FUNCTION

$$apply \equiv \Lambda\alpha\Lambda\beta(f : \alpha \rightarrow \beta)\lambda(x : \alpha).f x$$

```
UFunction uapply = new UFunction() { void apply() {
  ores = new UFunction() { void apply() {
  UFunction f = (UFunction) uapply.oarg;
  f.oarg = oarg; // set boxed argument
  f.parg = parg; // set integer argument
  f.apply();
  ores = f.ores; // set boxed result
  pres = f.pres; // set integer result
}};}};
```

FUNCTION APPLICATION

$$apply\ Long\ Long\ inc\ 3$$

```
uapply.oarg = uinc;
uapply.apply();
UFunction c2 = (UFunction) uapply.ores;
c2.larg = 3;
c2.apply();
```

CURRENT STATUS

- Sketch of formalization (elaborating to dual Java expressions / variables)
- Skeleton of implementation
- ‘new-unboxing’ branch

RELATED WORK

- Iulian Dragos. *Compiling Scala for Performance.* PhD thesis, IC, Lausanne, 2010.

- Vlad Ureche, Cristian Talau, and Martin Odersky. *Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations*. OOPSLA '13.

- Specialization in dynamically typed languages: Lua, Lisp…

- JVM optimizations (e.g. Graal VM JITC specializes boxed types)

CONCLUSION + Q&A

- *Boxing* brings time overhead
- *Specialization* brings code size explosion
- *Mini-boxing* seems to be the optimal tradeoff
- It is, however, a bit difficult to implement from scratch and we can do a bit better with our representation