Optimizing Higher-Order Functions in Scala

Iulian Dragos
EPFL

Outline

- A quick tour of Scala
  - Generalities
  - Extension through libraries
- Closure translation in Scala
  - Cost
- Optimizations in the Scala compiler

Introduction

- Scala is a statically-typed language
- Brings together object-oriented and functional programming
- Seamless interoperability with Java
- Extensible: grow the language through libraries
  - *for*-loops are provided through a library class, **BigInt** indistinguishable from the primitive type **Int**
- Problem: familiar looking code may hide unexpected costs

Scala

- A statically-typed language
  - object oriented
    - has *classes, traits (interfaces), objects*
  - functional
    - has *higher order functions, pattern matching, parameterized types (generics)* and *virtual types*
  - compiles to Java bytecode
    - can run on unmodified JVMs, can use java libraries

Extensibility

- Syntactic sugar
  - infix methods:
    - `xs map println ==> xs.map(println)`
  - for comprehensions:
    - ```
      for (i <- xs) println(x)
        ==> xs.foreach(x => println(x))
      ```
- Implicit conversions
  - adapt types through user-defined conversions
    - ```
      implicit intWrapper(x: Int) =
          new RichInt(x)
      42.toHex ==> intWrapper(42).toHex
      ```

For loops

```
for (i <- 1 until 10) print(i)


(new RichInt(1)).until(10).foreach({ i: Int => print(i) })


class Range(val start: Int, val end: Int) extends Seq[Int] {
  override def foreach(f: Int => Unit) = //..
  //..
}
```

- *for*-loops are library code
  - users can write their own looping constructs
  - there is a (hidden) runtime cost

Closure conversion

- Functions are values (therefore objects)
  - Translated to anonymous classes
    - Implement a *FunctionN* trait, where N is the arity
    - The captured environment is saved as fields, initialized on construction

```
trait Function1[R, A] {
  def apply(x: A): R
}
```

```
final class anonfun3 extends
                        Function1[Unit, Int] {

  def apply(i: Int) = print(Int.box(i))
  def apply(x1: Object): Object = {
    apply(scala.Int.unbox(x1))
    scala.runtime.BoxedUnit.UNIT
  }
}
```

Closure conversion

```
def sum(xs: List[Int], bound: Int): Int = {
  var sum = 0
  for (i <- 1 until xs.length)
    if (xs(i) > bound) sum += xs(i)
  sum
}
```

```
def sum1(xs$1: List, bound$1: Int): Int = {
  var sum$1: IntRef = new IntRef(0);
  Predef.intWrapper(1).until(xs$1.length).foreach({
    (new anonfun$1(this, xs$1, bound$1, sum$1): Function1)
  });
  sum$1.elem
};
```

- Captured variables are turned into fields of closure classes
  - Mutable fields are wrapped by reference cells

Closure conversion

```
final class anonfun$1 extends Object with Function1 with ScalaObject {
  def this(outer: test.Main, xs: List, bound: Int, sum: IntRef) = {
    this.outer = outer; this.xs = xs
    this.bound1 = bound; this.sum = sum
  };

  final def apply(i: Int): Unit =
    if (Int.unbox(xs.apply(i)) > bound)
      sum.elem = elem + Int.unbox(xs.apply(i));

  final <bridge> def apply(x$1: Object): Object = {
    apply(Int.unbox(x$1));
    BoxedUnit.UNIT
  };

  private val outer: test.Main = _;
  private val xs: List = _;
  private val bound: Int = _;
  private val sum: scala.runtime.IntRef = _
}
```

Closure conversion

- Cost that should be eliminated in known contexts:
  - indirection (bridge methods, boxing/unboxing)
  - object allocation (closure objects)
  - class generation for anonymous functions

Problem

- Closures should be optimized:
  - Class explosion leading to long load times
  - Primitive values are boxed
  - Many short-lived objects
- The JVM optimizer is not enough
- Can the Scala compiler do better?
  - Not having the whole program at hand

Optimizations in the Scala compiler

- Uses a stack-based, CFG based intermediate representation (ICode)
  - Java bytecode can be parsed back to ICode
- Phases
  - Inlining
  - Closure elimination
  - Dead-code elimination
  - Peephole optimizer

ICode reader

- Need to analyze/inline library code
  - But shouldn't rule out separate compilation
- Java bytecode is parsed to ICode
  - Has to resolve symbols
  - Has to type locals (sometimes needs splitting)

Inlining

- Uses TFA for deriving the most precise types at local variables and stack positions
  - Propagates types from allocation sites
  - Method calls are resolved when the type of the receiver is determined to be final
  - It is more precise than CHA and RTA
    - Method calls to *FunctionN* methods can't be resolved by RTA pruning
- Methods are inlined repeatedly
  - Higher-order functions and closure applications are preferred

Closure elimination

- Determines what values on the stack, or in object fields are copies of a local variable
  - Also tracks special values like **this**, or primitive constants
  - Simple heap model: objects are records, populated by known constructors
  - Replaces field accesses by local variables whenever possible
  - Often closures' environments become dead
    - And optimizes unnecessary boxing as well!

DCE

- Dead-code elimination cleans after the previous phases
  - Removes closure object allocation
  - Suppresses code generation for dead closure classes
  - Uses a simple mark & sweep algorithm, starting with 'useful' instructions

Results

Test Case	Running time (ms)	Optimized (ms)	Speed up
assert	104.8	67.4	36%
assert(dis)	79.6	44.4	44%
matrix	75.4	40.4	46%

- Each test was run once to warm up the VM
- Each measurement is an average over 5 runs

Future work

- Improve compilation times
  - Inlining repeatedly requires solving the data-flow problem for very similar flow-graphs
    - Idea: reuse and combine solutions for the caller and callee.
- Improve precision
  - pureness analysis