# GC-savvy Closure conversion

Miguel Garcia
http://lamp.epfl.ch/~magarcia
LAMP, EPFL

2012-07-03

# Outline

## Status-quo of Closure conversion in `scalac`

- Optimizations already in place

## Lightweight optimizations

- 1 of 2: Special-case inlining for forwarders to specialized methods
- 2 of 2: “Proceduralization” before UnCurry

## Heavyweight optimizations

## How can `MethodHandle`s help with Closure conversion?

What we get now, say for "`(i => i < args.length)`" where "`args`" is a formal param of the enclosing method.

```
@SerialVersionUID(0)
final <synthetic>
class $anonfun$main$1
extends scala.runtime.AbstractFunction1$mcZI$sp
with Serializable {

  final
  def apply(i: Int): Boolean = apply$mcZI$sp(i);

  <specialized>
  def apply$mcZI$sp(v1: Int): Boolean = v1.<(args$1.length());

  final <bridge>
  def apply(v1: Object): Object = scala.Boolean.box(apply(scala.Int.unbox(v1)));

  <synthetic> <paramaccessor>
  private[this]
  val args$1: Array[String] = _;

  def <init>($outer: Test, args$1: Array[String]): anonymous class $anonfun$main$1 = {
    $anonfun$main$1.this.args$1 = args$1;
    $anonfun$main$1.super.<init>();
    ()
  }

}
```

Optimizations already in place:

- GC tracks liveness of formal-params via DFA, so that closures received as argument may be GC'ed before method exit. No need to null-out right after last use (doable at bytecode level).
- In some cases, the `$outer` field and its accessor are eliminated due to the following in `Constructors`:

```
// Could symbol's definition be omitted, provided it is not accessed?
// This is the case if the symbol is defined in the current class, and
// ( the symbol is an object private parameter accessor field, or
//   the symbol is an outer accessor of a final class which does not o
def maybeOmittable(sym: Symbol) = sym.owner == clazz && (
  sym.isParamAccessor && sym.isPrivateLocal ||
  sym.isOuterAccessor && sym.owner.isEffectivelyFinal && !sym.isOverri
  !(clazz isSubClass DelayedInitClass)
)
```

In a “forwarder-to-specialized,” caller and callee share the same signature. Example:

```
final def apply(i: Int): Boolean = apply$mcZI$sp(i);

<specialized> def apply$mcZI$sp(v1: Int): Boolean = v1.<(5);
```

Their inlining increases method size, adding superfluous locals.

Alternatives:

1. inline without adding superfluous locals, or
2. run both `closelim` and `dce` in-between inlining iterations (required to remove boxing and superfluous locals). Slow.

Instead of converting a `Function` node to

```
Block( List(local-ClassDef), <| instantiation-of-localclass |> )
```

convert to

```
Block( List ( <| val rcv = ... |> ,
              DefDef-inlinedA     ,
              DefDef-inlinedB      ) ,
       <| invocation-of-inlinedA |> )
```

where `DefDef-A` is an (early inlined) higher-order method, with two changes in its body:

- replace `This` with the original receiver (“`rcv`” above).
- applications of the function-arg rephrased as invocations of `DefDef-B` (which in turn is derived from the closure body)

## Example: "`(1 to 10) foreach println`" becomes:

```
{ val rcv = Predef.intWrapper(1).to(10)

  def inlinedA() {
    /*- OK the condition below should either
          (1) veto early-inlining as a whole; or
          (2) be inlined to invoke 'inlinedB(i)';
    Workaround: make Range.foreach not pass the fun-arg around.*/
    if (rcv.validateRangeBoundaries(f)) {
      var i = rcv.start
      val terminal = rcv.terminalElement
      val step = rcv.step
      while (i != terminal) {
        inlinedB(i) /*- this used to be 'f(i)' */
        i += step
      }
    }
  }

  def inlinedB(x: Int): Unit = Predef.println(x);

  inlinedA()
}
```

1. Pros:
   - no object instantiation
2. Cons:
   - number of method calls isn't reduced
   - code duplication
3. Both "`DefDef-inlinedA`" and "`DefDef-inlinedB`" may contain `return`s (in the latter, we can detect whether a `return` is non-local to avoid mistakes).
4. WARNING what if the closure-instantiation has constructor-args with side-effects (initialization of lazy-vals).
5. Most useful in connection with another rewriting, right before lambda-lift: "inlining-of-local-methods-invoked-just-once". `https://groups.google.com/d/msg/scala-internals/KMp-5DNn5NI/Ac_-lTPBxDsJ`.

## Heavyweight optimizations (analysis of higher-order control-flow)

1. Let's assume a mutable local L, captured by a closure C, ie C has a constructor arg of type ...Ref for L. In case L isn't modified from the time C is instantiated till the last (read) access in C, then L can be passed by value. Caveat: the conversion of L to ...Ref might still be imposed by other closures or local classes.
2. `http://blog.cdleary.com/2010/05/notes-from-the-js-pit-closure-optimization`

   *"As long as there's no possibility of escape between a declaration and its use in a nested function, the nested function knows exactly how far to reach up the stack to retrieve or manipulate the variable — the activation record stack is totally determined at compile time. Because there's no escaping, there's not even any need to import the upvar into the Algol-like function."*

## `MethodHandle`: enabler for `invokedynamic` (JSR 292)

1. seen as a type-safe function pointer:
   1.1 can be invoked via `invokeExact()` (no auto-boxing)
   1.2 method access checks are based on a method handle's creator, not its caller (invoke private and super methods from anywhere)
2. seen as an object:
   2.1 `ldc` bytecode refers to a constant MH
   2.2 can bind arguments (partial application)
   2.3 closed under composition with other MHs. Use cases:
      - function composition, argument adaptation via casting or boxing
      - runtime metaprogramming `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2012Q2/RuntimeMP.pdf`

More info:

John Rose. Bytecodes meet combinators: invokedynamic on the JVM

`http://cr.openjdk.java.net/~jrose/pres/200910-VMIL.pdf` `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`

The Java 8 design (JSR 335)

- favors compatibility with pre-MH libraries, which use eg Runnable.
- at the cost of instantiation overhead (as compared to raw MHs)
- minimized by VM-specific *meta factories* offered by JDK API

Highlights:

1. *“Instead of generating bytecode to create the object that implements the lambda expression . . . we delegate the actual construction to the language runtime . . . so that . . . JRE implementations can choose their preferred implementation strategy.”*
2. *“Performance impact: Serializability imposes some additional costs on lambdas . . . Therefore it is preferable to treat serializable lambdas separately rather than making all lambdas serializable, and imposing these costs on all lambdas.”*