Faster closures in Scala via Stack-allocation

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

2012-11-04

Outline

Background

- How `uncurry` goes about closure conversion
- Some inescapable performance laws
- Terminology

Candidate approaches

- Early Inlining
- Closures as method-handles, `scala.FunctionX` compatible

Which approach results in faster code?

What we get 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>();
    ()
  }

}
```

A few steadfast performance characteristics:

1. Nothing beats stack-allocation
2. Stack-traffic always cheaper than pointer-chasing over heap-allocated data
3. Passing `IntRef` and friends on the stack not really a problem: the VM's Escape Analysis (usually) stack-allocates them.
4. Corollary: lifting local methods is OK, lifting closures is bad

Open question: Relative performance of

- calling a partially-applied method handle
  vs.
- virtual invocation providing all arguments

Terminology

- **Hi-O method**: the "higher-order" method taking one or more closure instances as argument. Example, `Range.foreach()`
- **closure-state**: the values of fields of an anonymous-closure-class, all of them final. Usually including an `$outer` field to access external locations other than local-captures.
- **closure-methods**: besides `apply()` and its specialized siblings, `lambdalift` turns `apply()`'s local methods into closure-methods.
- **closure-constructor**: initializes the closure-state with
  - THIS of the invoker, ie the closure's `$outer` reference
  - captured-locals
  - the rest if accessed via `$outer` (one or more hops)

Candidate approaches:

1. Inlining at Hi-O callsite (assumes known method to dispatch):
    1.1 before `uncurry` (aka "early inlining")
    1.2 in the experimental optimizer.
2. Closure `apply()` delegating to partially-applied method-handle

(1.1) and (1.2) result in stack-allocated closure-state, guaranteed.
The next few slides showcase (1.1) and (2.)
Extensive details about (1.2) at:

`https://groups.google.com/d/topic/scala-internals/Hnftko0MzDM/discussion`

```
rvc.hiO( <M-args> , closure , <N-args> ).etc
```

can be converted (right before uncurry) into:

```
{
  val rcv = ...
  val m_1 = ... // and so on for the M-args and the N-args
  val n_1 = ...

      def closureApply( <as-in-original> ) : ....

      def inlinedHiO( rcv, <M-fmls> , <N-fmls> ) : ... {
        // body of Hi-O adapted to use corresponding formal
        //
        // LOAD closure-arg followed by INVOKEVIRTUAL apply()
        // becomes
        //   closureApply() callsite, initially with original args
        //   (lambdalift will stack all needed closure state)
      }

   inlinedHiO( rcv, m_1, ... /* closure skipped */ , n_1, ... )

}.etc
```

A transformation in two phases. First step, on entry to `uncurry`:

```
rcv.hiO( ... , Function(vparams, body) , ... ).etc
```

can be transformed into:

```
{
    def closureApply( <Function.vparams> ) = <Function.body>

  closureApply( <zero-at-each-arg> )
  // fake call to be removed after lambdalift stacks closure-state

  val mhApply: MethodHandle = null
  // rhs will become (in the 2nd step) a partially-applied MH

  rcv.hiO( ... , InvokeExactFunction , ...).etc
}
```

where `InvokeExactFunction` is a `Function` node with body:

- `mhApply.invokeExact( <Function-vparams> )`
- encoding `Function.body`'s type as return type of that callsite.

Second step, on exit from `lambdalift`:

```
{
    def closureApply( <Function.vparams> ,
                      outer,
                      captured-locals ) =
      <Function.body>

  // lambdalift appended arguments (outer and captured-locals)
  // to the fake call that used to be here.
  // Note down those arguments and remove the call.

  val mhApply: MethodHandle =
    <constant MH targeting `closureApply`,
     partial-binding the extra args
     borrowed from the removed callsite>

  rcv.hiO( ... , InvokeExactFunction , ...).etc
}
```

To recap, candidate approaches:

1. Inlining at Hi-O callsite (assumes known method to dispatch):
   1.1 before `uncurry` (aka "early inlining")
   1.2 in the experimental optimizer.
2. Closure `apply()` delegating to partially-applied method-handle

Some factors influencing resulting speed:

- stack-allocation (of closure-state) always faster than alternatives
- code duplication (of the Hi-O method) taxes the JIT compiler
- relative performance of MH partial-binding wrt direct call