Type-safe Off-heap Memory for Scala

Denys Shabalin, LAMP/EPFL

Off-heap memory: memory which is allocated and managed outside of garbage collected heap.

Why?

- You want to handle large data in-memory
- GC does not meet your latency requirements
- You want to share memory with native code

State of the off-heap

Direct byte buffers

```
case class Point(x: Int, y: Int)
val point = Point(10, 20)

// alocating
val bb = java.nio.ByteBufer.allocateDirect(size)
bb.putInt(0, p.x)
bb.putInt(4, p.y)

// reading
val x = bb.getInt(0)
val y = bb.getInt(4)
val point = Point(x, y)
```

Direct byte buffers: issues

- low-level api
- at most 2GB per buffer
- bound checking affects performance

sun.misc.Unsafe

```
case class Point(x: Int, y: Int)
val point = Point(x = 10, y = 20)

// alocating
val unsafe = sun.misc.Unsafe.getUnsafe()
val addr = unsafe.allocateMemory(size)
unsafe.putInt(addr, p.x)
unsafe.putInt(addr + 4L, p.y)

// reading
val x = unsafe.getInt(addr)
val y = unsafe.getInt(addr + 4L)
val point = Point(x, y)
```

sun.misc.Unsafe: issues

- even lower level api
- lack of memory safety
- memory leaks

JNI/JNA interop with C code

```
struct point { int x; int y; }

JNIEXPORT jlong JNICALL Offheap_allocate(JNIEnv *env, jobject jpoint) {
  struct *point  = (*point) malloc(sizeof(point));
  ...
  return (jlong) point;
}

JNIEXPORT jobject JNICALL Offheap_read(JNIEnv *env, jlong address) {
  ...
}
```

JNI/JNA: issues

- as low-level as it gets
- non-trivial amount of boilerplate
- JNI calls limit JIT optimizations
- lack of memory safety
- memory leaks
- code distribution is complicated

State of the off-heap

Very low-level.

What is a Memory?

Any subtype of trait:

```
trait Memory {
  def allocate(size: Size): Addr
  def copy(from: Addr, to: Addr, size: Size)
  def getChar(addr: Addr): Char
  def getByte(addr: Addr): Byte
  ....
  def putChar(addr: Addr, value: Char): Unit
  def putByte(addr: Addr, value: Byte): Unit
  ...
}
```

What is a Memory?

Single interface, many implementations:

- NativeMemory (Unsafe-based)
- ByteBufferMemory
- ...

Why not just use unsafe directly?

It might go away in the future. Some thin abstraction layer lets us swap implementations without changing client code.

Memory

Best of Unsafe and ByteBuffers:

- versions with x64 and x32 addressing
- safety is optional
- automatic resource cleanup
- easily implementable interface
- still low-level

Offheap classes

@data classes

Just like case classes only off-heap.

```
@data class Point(x: Int, y: Int)

val memory = NativeMemory()
val point = Point(10, 20)(at = memory)
```

@data classes

Just like case classes only off-heap.

```
@data class Point(x: Int, y: Int)

implicit val memory = NativeMemory()
val point = Point(10, 20)
```

@data classes

```
// field access
point.x + point.y

// pattern matching
val Point(x, y) = point

// copy on write
val point2 = point.copy(x = 42)

// nice toString
point.toString == "Point(10, 20)"
```

@enum classes

Tagged unions with straightforward syntax.

```
@enum class Figure
object Figure {
  @data class Point(x: Float, y: Float)
  @data class Circle(center: Point, radius: Float)
  @data class Segment(start: Point, end: Point)
}
```

@enum classes

```
// implicit upcasts
val fig: Figure = Figure.Circle(Figure.Point(10, 20), 30)

// type tests
fig.is[Figure.Circle]

// explicit downcasts
val circle = fig.as[Figure.Circle]

// pattern matching
fig match { case Figure.Circle(center, r) => }

// nice toString
fig.toString == "Figure.Circle(Figure.Point(10.0, 20.0), 30.0)"
```

Offheap arrays

Looks and feels just like the standard ones.

```
implicit val memory = NativeMemory()
var arr = Array(1, 2, 3)

// bound-checked indexed access
arr(0) == 1
arr(1) == 2
arr(2) == 3
arr(3) // throws OutOfBoundsException

// mapping
val arr2 = arr.map(_ * 2)

// iterating
arr2.foreach(println)
```

So we've got memory, what about management?

Regions are the answer.

Region-based memory

Delimited scopes with constant-time allocation & clean-up.

```
implicit val pool = Pool(NativeMemory())

Region { implicit r =>
  val point = Point(10, 20)
}
```

Region-based memory

Objects are accessible as long as Region is open.

```
implicit val pool = Pool(NativeMemory())

var point: Point = _
Region { implicit r =>
  point = Point(10, 20)
}
point.x // throws InaccessibleRegionException
```

Does it have to be scoped?

No, open-ended regions are also supported.

```
val region = Region.open
...
region.close
```

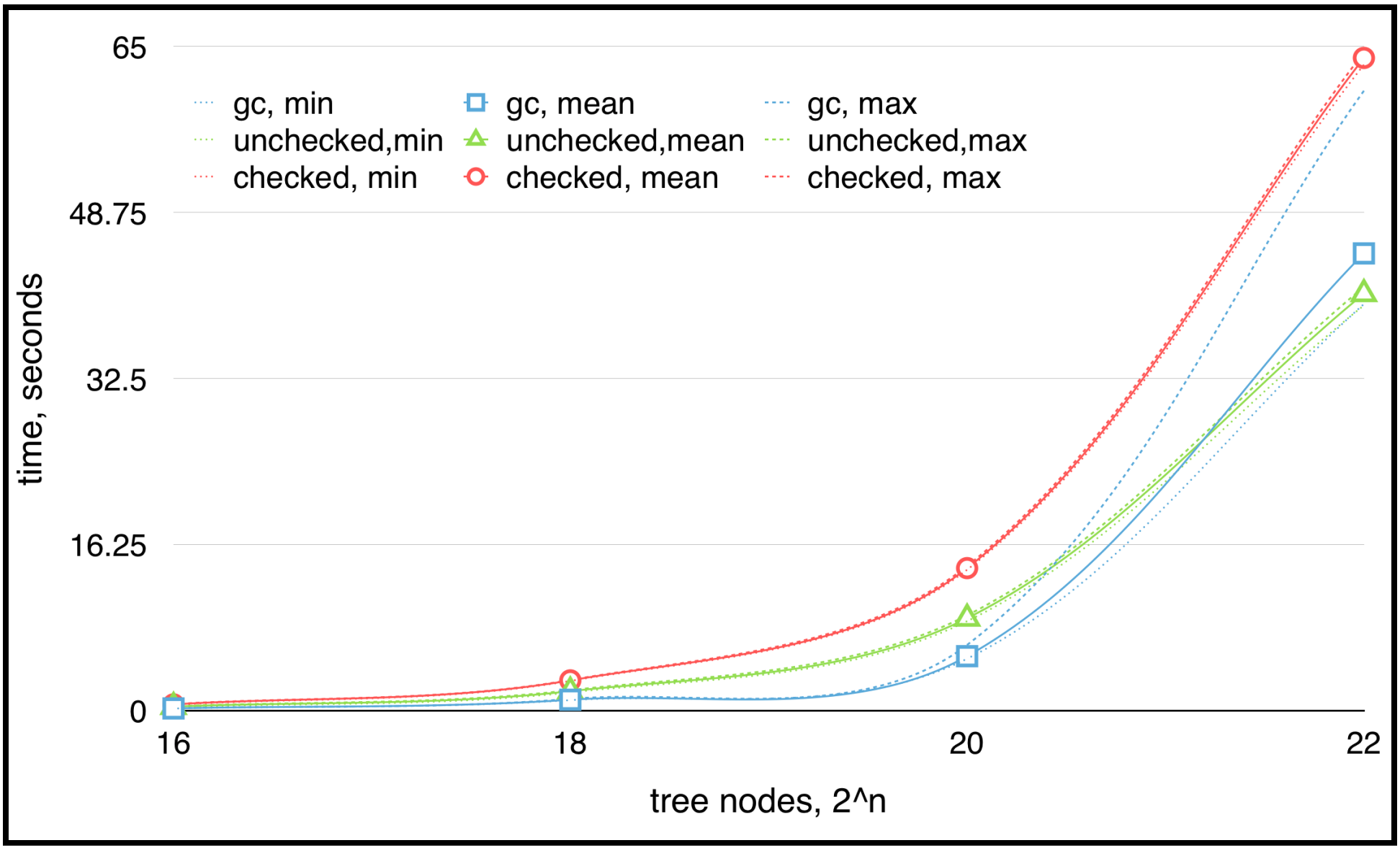
Do I have to close the region?

No, it will be automatically closed once finalized.

```
val region = Region.open
...
```

What about performance?

BinaryTree benchmark



How does it work?

Macros all the way.

@data and @enum are macro annotations.

```
// lets look at expansion of @data
@data class Point(x: Int, y: Int)
```

Macros all the way.

Checked mode desugaring.

```
class Point(ref: Ref) extends AnyVal {
  def x: Int = ref.memory.getInt(ref.addr)
  def y: Int = ref.memory.getInt(ref.addr)
  ...
}
object Point {
  def apply(x: Int, y: Int)(implicit m: Memory): Point = {
    val addr = m.allocate(8)
    m.putInt(addr,      x)
    m.putInt(addr + 4L, y)
    new Point(Ref(addr, m))
  }
}
```

Macros all the way.

Unchecked mode desugaring.

```
class Point(addr: Long) extends AnyVal {
  def x: Int = unsafe.getInt(addr)
  def y: Int = unsafe.getInt(addr)
  ...
}
object Point {
  def apply(x: Int, y: Int)(implicit m: Memory): Point = {
    val addr = m.allocate(8)
    unsafe.putInt(addr,      x)
    unsafe.putInt(addr + 4L, y)
    new Point(addr)
  }
  ...
}
```

Macros all the way

Array operations are blackbox macros.

```
arr.map(_ * 2)
```

Macros all the way

```
{
  val narr  = Array.uninit[int](arr.length)
  var p     = narr.ref.addr + Memory.sizeof[size]
  arr.foreach { v: int =>
    mme.putInt(p, v * 2)
    p += Memory.sizeof[Int]
  }
  narr
}
```

Macros all the way

```
{
  val narr = Array.uninit[int](arr.length)(mem)
  var p    = narr.ref.addr + Memory.sizeof[Size]

  {
    val len = mem.getLong(arr.ref.addr)
    var p2 = arr.ref.addr + Memory.sizeof[Size]
    val bound = p2 + len * Memory.sizeOf[Int]
    while (p2 < bound) {
      mem.putInt(p, mem.getInt(p2) * 2)
      p += Memory.sizeof[Int]
      p2 += Memory.sizeof[Int]
    }
  }

  narr
}
```

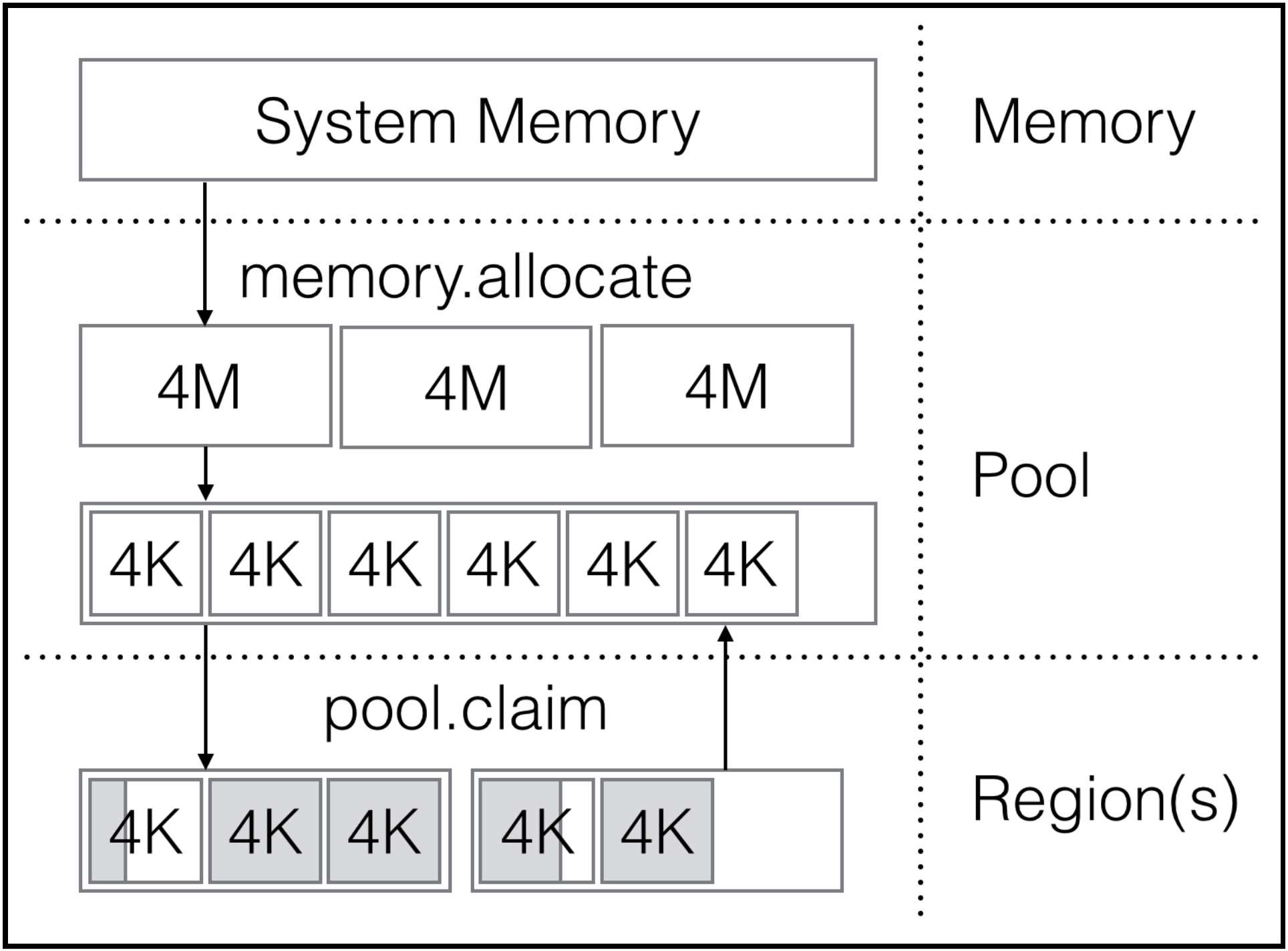
Macros all the way

```
{
  val narr = Array.uninit[int](arr.length)(mem)
  var p    = narr.ref.addr + 8

  {
    val len = mem.getLong(arr.ref.addr)
    var p2 = arr.ref.addr + 8
    val bound = p2 + len * 4
    while (p2 < bound) {
      mem.putInt(p, mem.getInt(p2) * 2)
      p += 4
      p2 += 4
    }
  }

  narr
}
```

Efficient memory pooling machinery


System Memory
Memory
memory.allocate
4M
4M
4M
Pool
4K
4K
4K
4K
4K
4K
pool.claim
4K
4K
4K
4K
4K
Region(s)

Can I start using it today?

Not yet, but experimental 0.1 release coming soon.

Source code is available today:

github.com/densh/scala-offheap

Summary

scala-offheap is:

- high-level and easy-to-use API to offheap memory
- with optional memory safety
- and deterministic performance

Questions?