Type-safe Off-heap Memory for Scala

Denys Shabalin, LAMP/EPFL

twitter.com/den_sh

Off-heap memory

Memory which is allocated and managed outside of garbage collected heap.

Why?

- You want to handle large datasets 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
- no safety guarantees
- memory leaks

JNI interop with C code

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

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

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

JNI issues

- as low-level as it gets
- non-trivial amount of boilerplate
- no safety guarantees
- memory leaks
- code distribution is complicated

State of the off-heap

- no data layout abstractions
- manual memory management
- very low-level

scala-offheap

- built-in data layout primitives
- semi-automatic memory management
- reasonably high-level
- doesn't stand in your way

Data layout primitives

Off-heap classes

Lightweight objects allocated in off-heap memory.

@data classes

Just like case classes only off-heap.

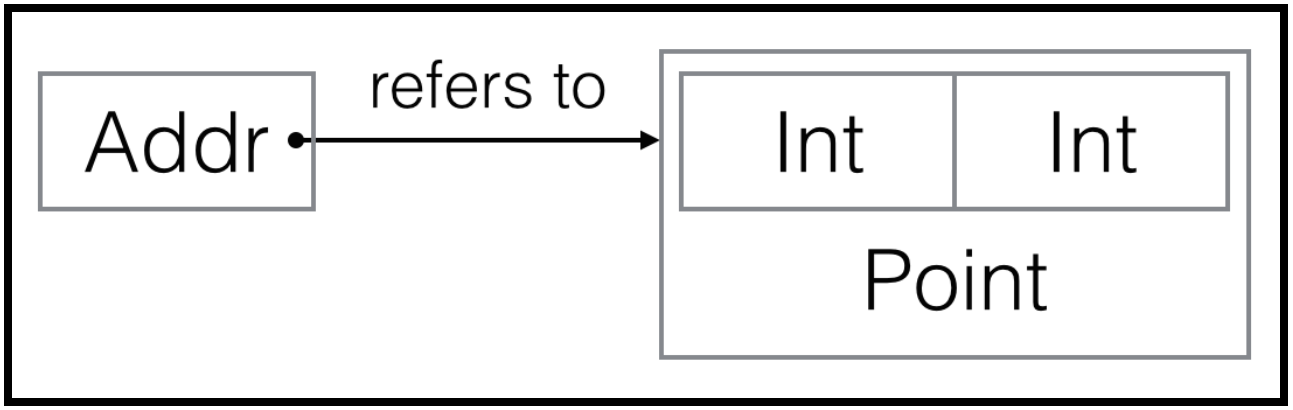
```
@data class Point(x: Int, y: Int)

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)"
```

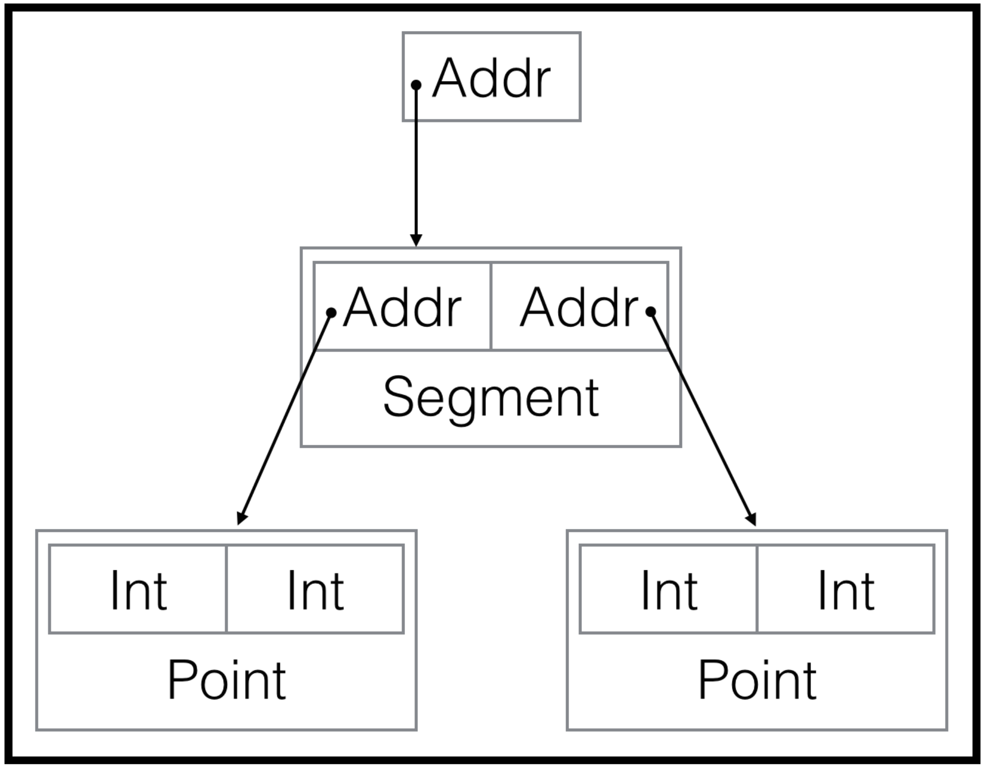
@data class representation

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

val p = Point(10, 20)
```



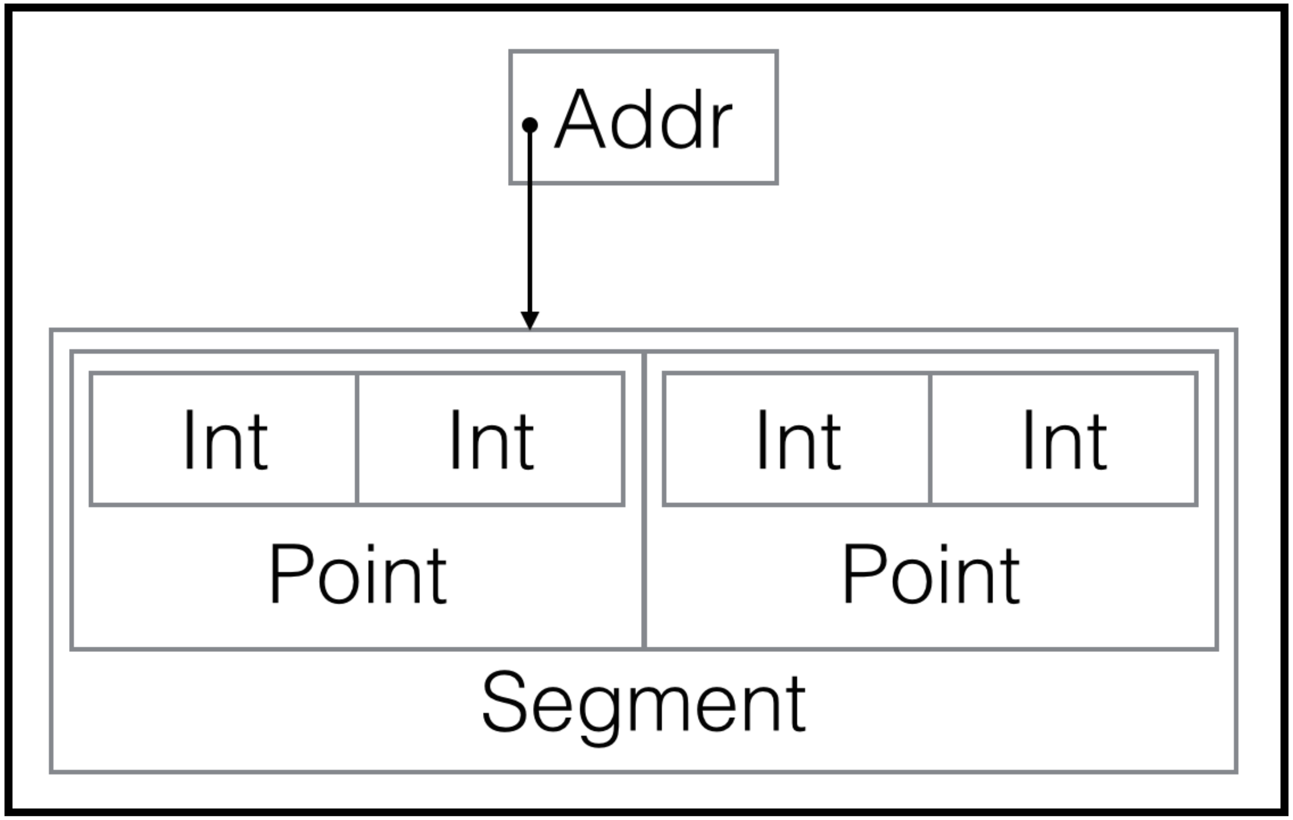
@data class composition

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

val s = Segment(Point(10, 20), Point(30, 40))
```



@embed fields

```
@data class Point(x: Int, y: Int)
@data class Segment(@embed start: Point, @embed end: Point)

val s = Segment(Point(10, 20), Point(30, 40))
```



@enum classes

Tagged unions with straightforward syntax.

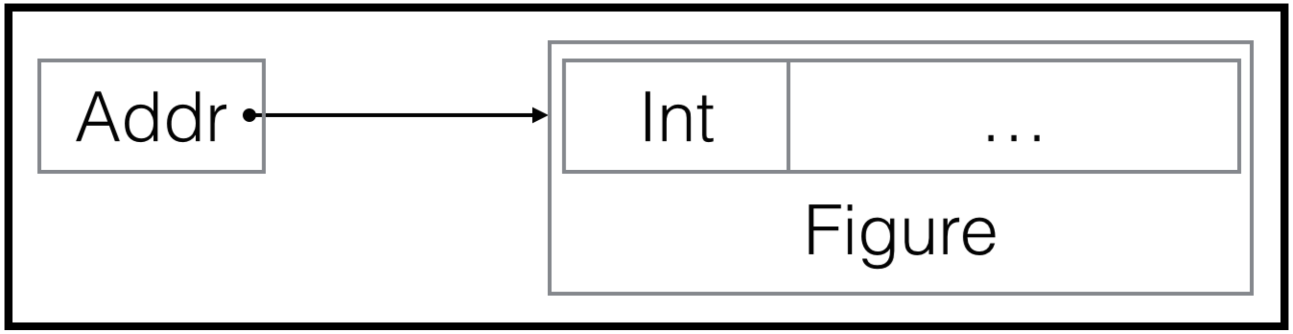
```
@enum class Figure
object Figure {
  @data class Point(x: Float, y: Float)
  @data class Circle(center: Point, radius: Float)
  @data class Segment(start: Point, end: Point)
}
```

@enum classes

```scala
// 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)"
```

@enum class representation

```
@enum class Figure
object Figure { ... }
```



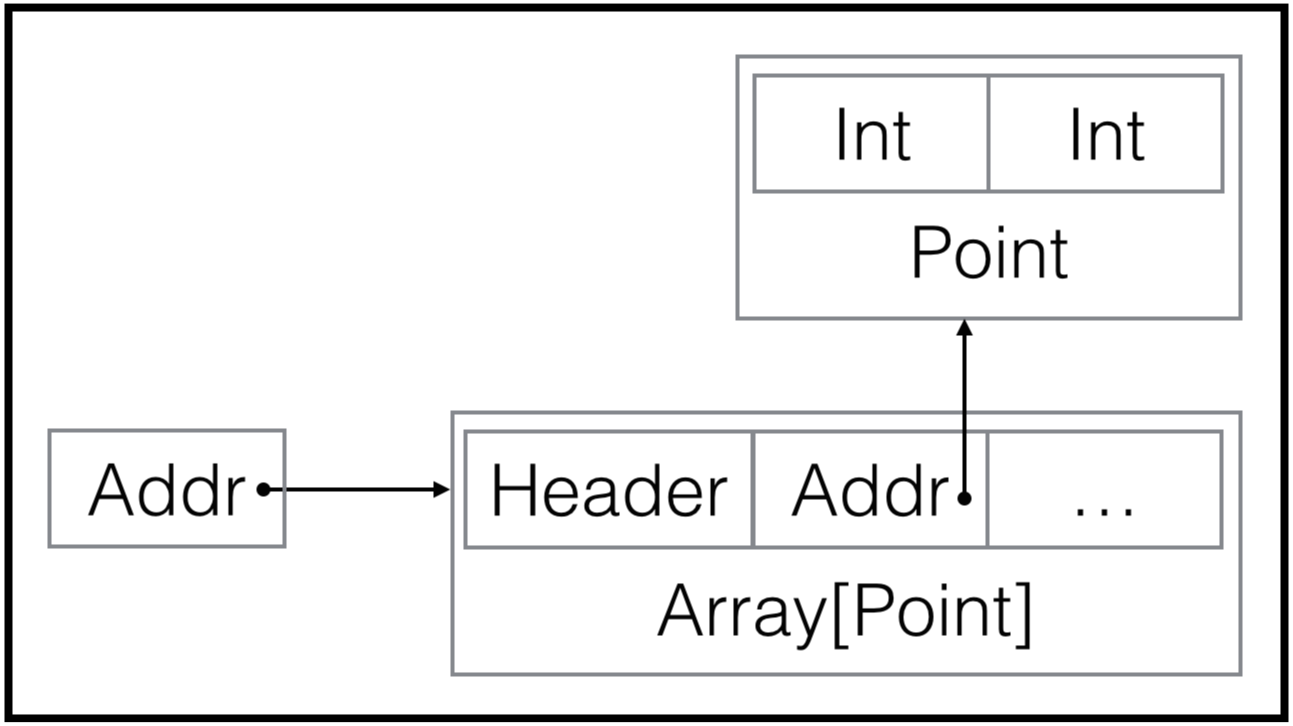
Off-heap arrays

Looks and feels just like the standard ones.

```scala
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)
```

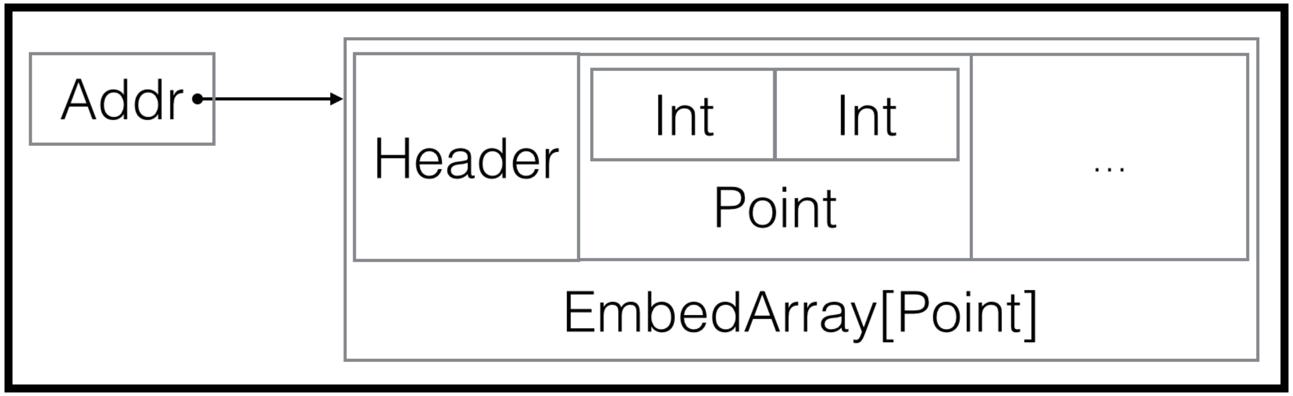
Off-heap array layout

```scala
val arr = Array(Point(10, 20), Point(30, 40))
```



Off-heap embed array layout

```
val arr = EmbedArray(Point(10, 20), Point(30, 40))
```



Memory management

offheap.Memory

Abstraction over memory access.

Thin layer over subset of `sun.misc.Unsafe` with added support for extra (optional) safety checks.

offheap.Allocator

Abstraction over memory management.

```
trait Allocator {
  def allocate(size: Size): Addr
  def reallocate(addr: Addr, size: Size): Addr
  def free(addr: Addr): Unit
}
```

An implicit instance required for all allocations.

malloc

Direct access to underlying system allocator.

```
scala> val p = Point(10, 20)(malloc)
p: Point = Point(10, 20)

scala> malloc.free(p.addr)
```

Regions

Annotated scoped memory management with (ammortized) constant-time allocation & clean-up.

```
implicit val props = Region.Props(Pool())

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

Regions

Objects are accessible as long as Region is open.

```
implicit val props = Region.Props(Pool())

var point: Point = _
Region { implicit r =>
  point = Point(10, 20)
}
point.x // throws InaccessibleMemoryException
        // under -Doffheap.checked.memory
```

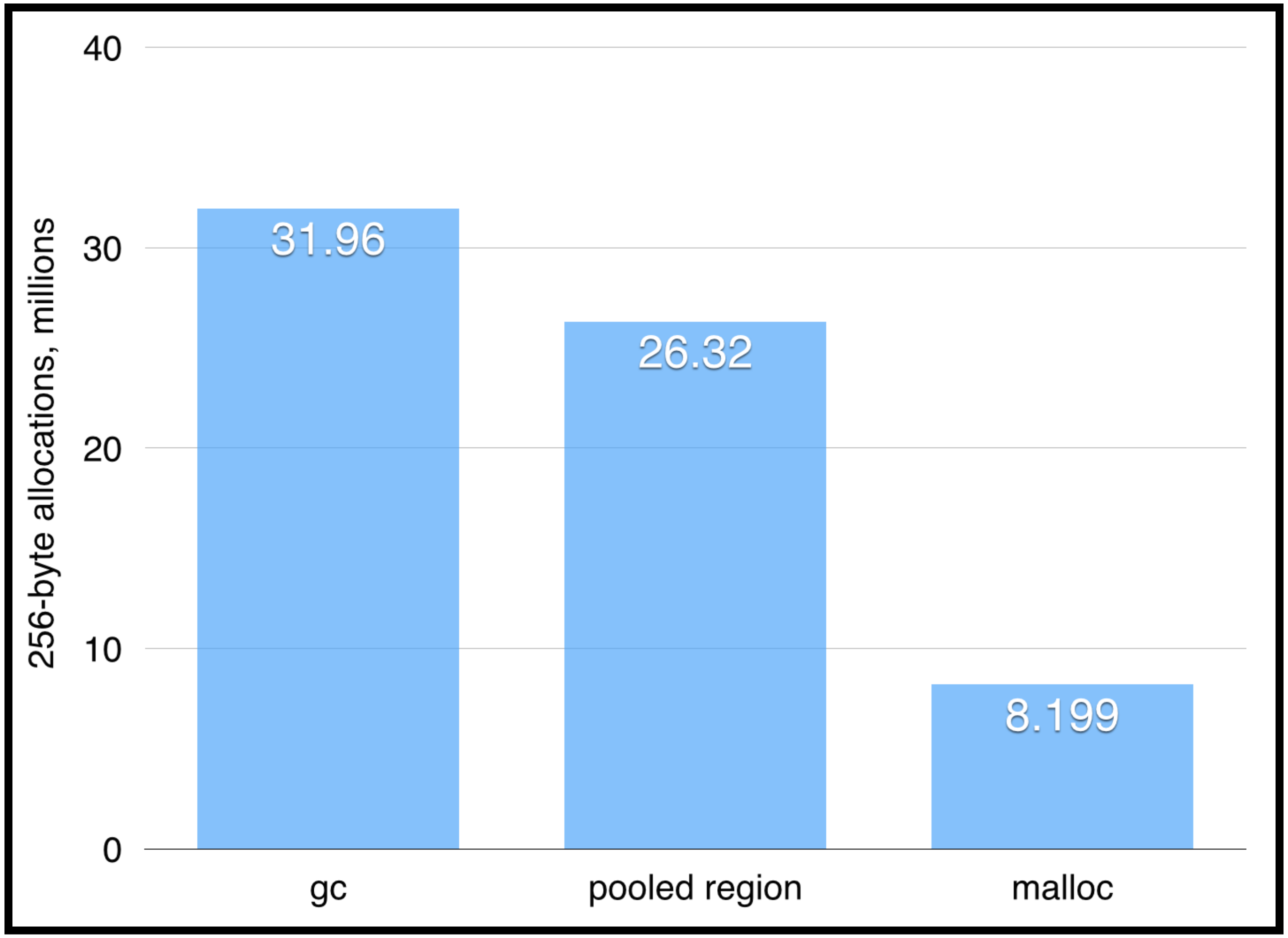
Regions

Open-ended regions are also supported.

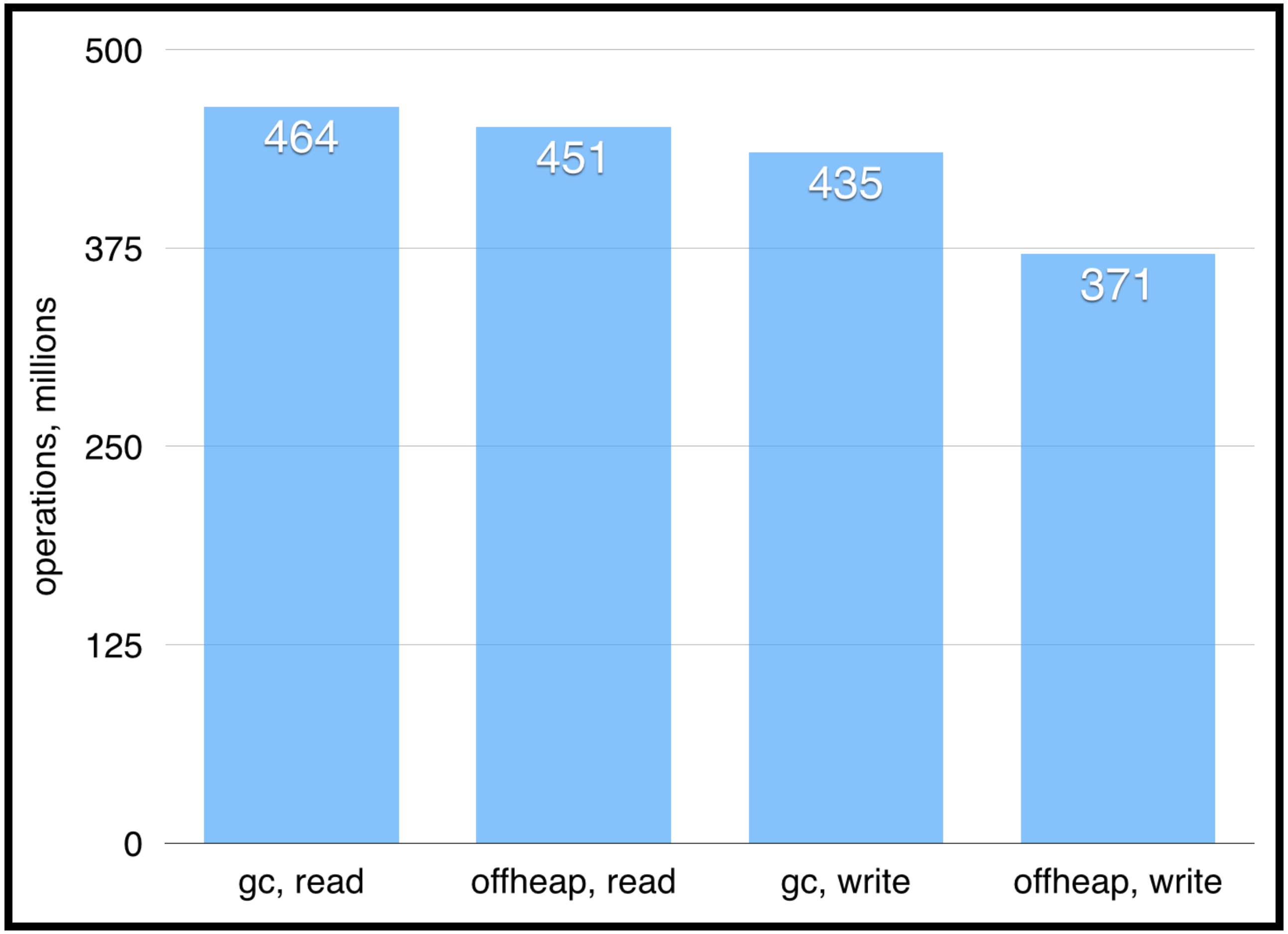
```
implicit val props = Region.Props(Pool())

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

Performance

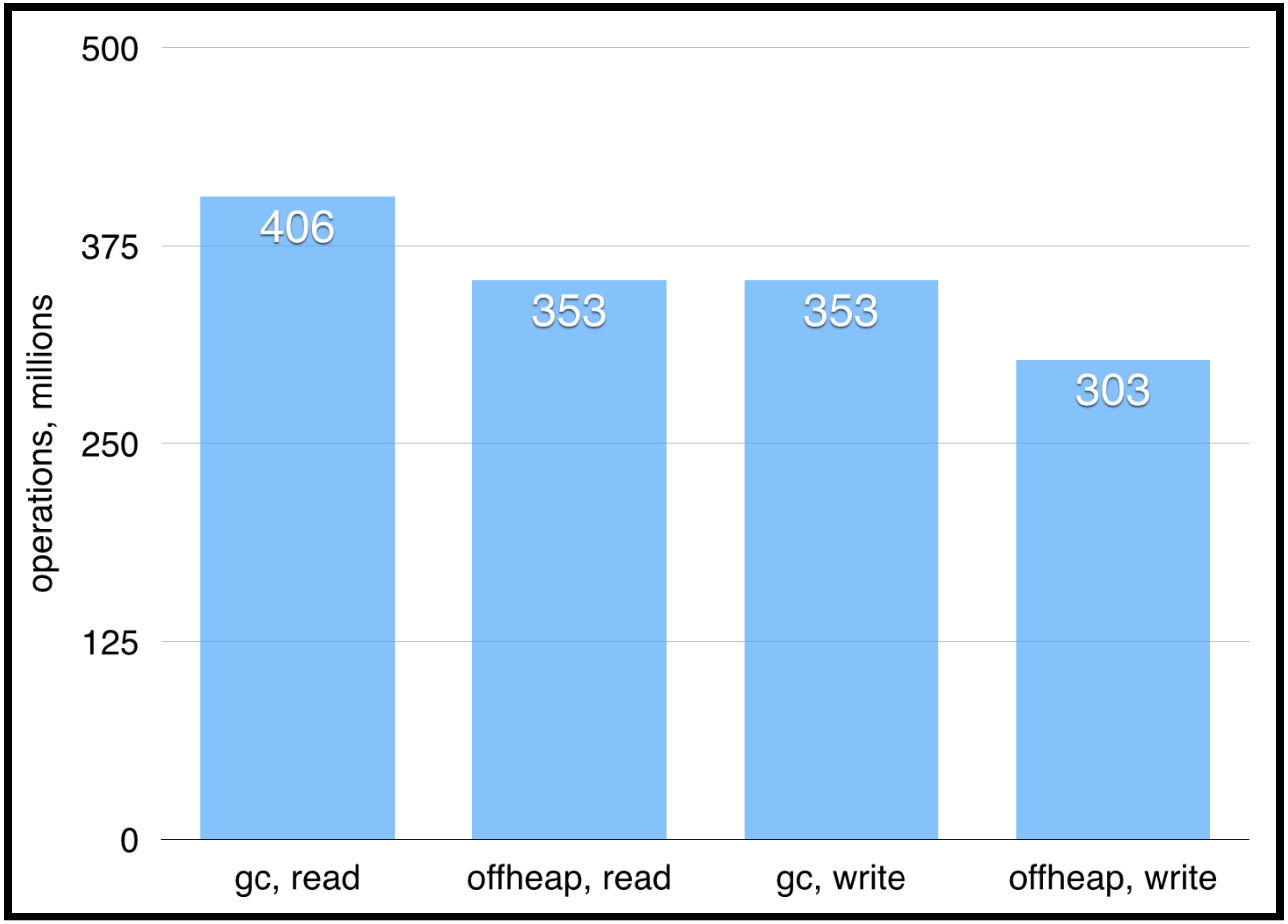
Performance: allocation



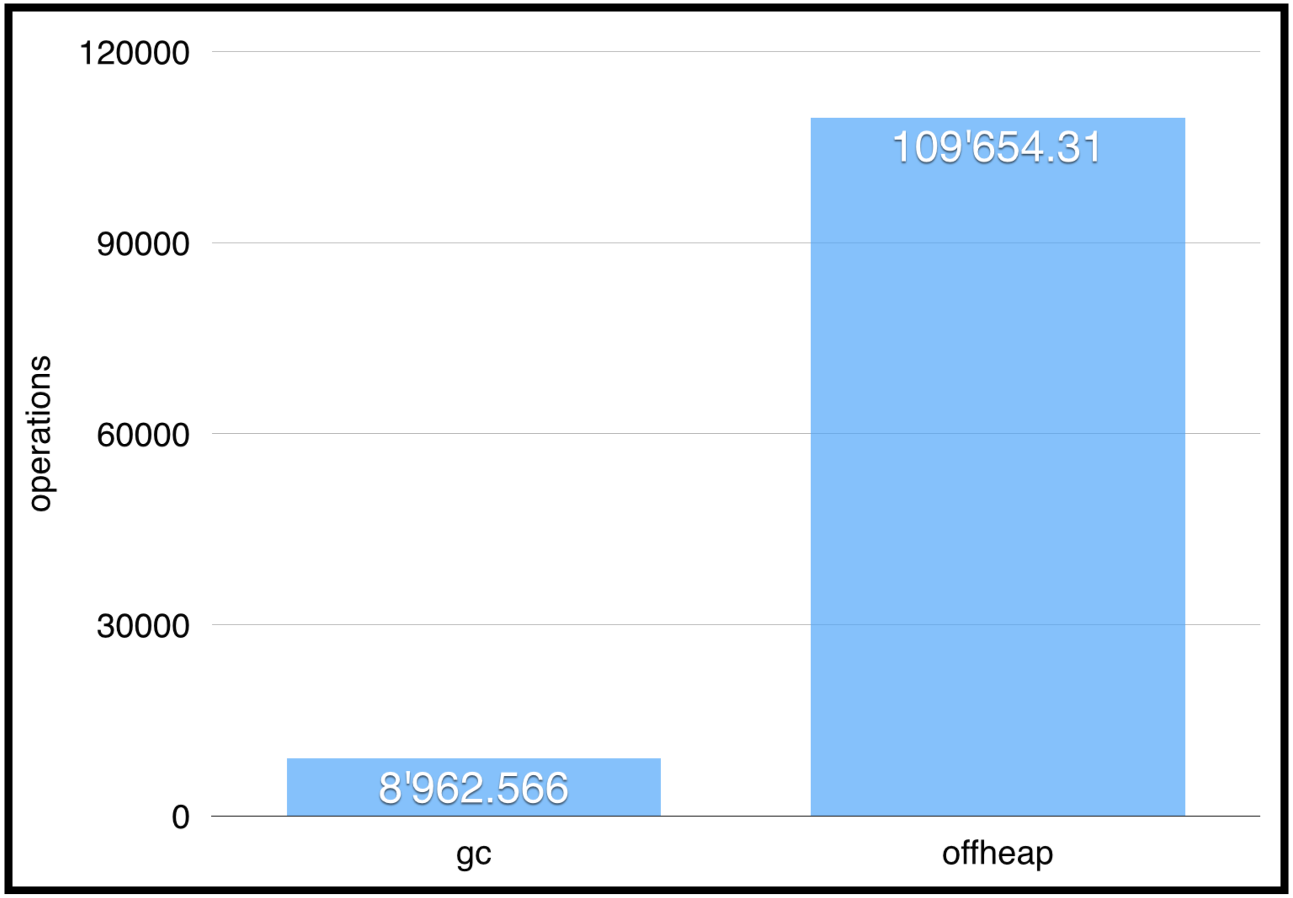
Performance: field read/write



Performance: array read/write



Performance: array map



Under the hood

Macros all the way

@data and @enum are macro annotations.

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

Macros all the way

```scala
class Point private(val addr: Addr) extends AnyVal {
  def x: Int = Memory.getInt(addr)
  def y: Int = Memory.getInt(addr)
  ...
}
object Point {
  def apply(x: Int, y: Int)(implicit a: Allocator): Point = {
    val addr = a.allocate(8)
    Memory.putInt(addr,      x)
    Memory.putInt(addr + 4L, y)
    new Point(addr)
  }
  ...
}
```

Macros all the way

Array operations are blackbox macros.

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

Macros all the way

```
{
  val len = arr.length
  val narr = Array.uninit[Int](len)
  var i = 0
  while (i < len) {
    narr(i) = arr(i) * 2
    i += 1
  }
  narr
}
```

Can I try it today?

Yes! See the sample sbt project for configuration & setup:

https://github.com/densh/scala-offheap-example

Future work

- jemalloc allocator support (contributed by @arosenberger)
- static memory safety guarantees
- ...

See our github issues for comprehensive list.

Summary

scala-offheap offers nice, easy to use and efficient abstractions to deal with off-heap memory.

Questions?

github.com/densh/scala-offheap