Region-based off-heap memory for Scala

Technical report

Denys Shabalin, Martin Odersky

first.last@epfl.ch

February 2, 2015

Introduction

At the moment, the Scala programming language offers a single memory management model for heap-allocated memory: fully automatic garbage collection. In theory, this approach lets the developer completely forget about memory management as GC will automatically deallocate objects for them whenever it considers them to be unreachable. Unfortunately, in practice GC has non-trivial performance trade-offs that might not be acceptable in some applications.

The goal of this project is to explore an alternative memory model: region-based memory. The main idea behind this model is to let developers annotate special scopes in code as regions and provide an APIs to allocate memory that would stay available until the execution leaves the scope. Once execution leaves the scope, all of the memory associated with it can be freed at once in one pass.

Regions are also know as memory pools, arenas or memory contexts and are in fact widely used in the C/C++ code bases as a means to efficiently manage objects with clear explicitly delimited lifetime. This approach is used in a number of high-profile projects such as Apache HTTP server, PostgreSQL and Google Chrome.

Apache memory pools [1]. This API is quite representative of how one might implement regions in C. Due to the simplicity of the language, one has to explicitly demarcate beginning and the end of the region by calls to corresponding `apr_pool_create` and `apr_pool_destroy` functions. In between of these two function calls one can allocate memory in the pool using `apr_palloc`:

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

void main() {
    apr_pool_t *mp;
    apr_pool_create(&mp, NULL);       // create a fresh pool
    for (i = 0; i < 100; ++i) {       // allocate in pool
        struct Point *p = apr_palloc(pool, sizeof(struct Point))
        p->x = i;
        p->y = i * 2;
        hello(p);
    }
    apr_pool_destroy(mp);             // free everything at once
}
void hello(struct Point *p) {         // points are friendly
    printf("Hi, I'm a point (%d, %d)", point->x, point->y)
}
```

Like most things in C, this API is completely memory-unsafe. Dereferencing a pointer allocated in a pool after the pool has been destroyed is an undefined behaviour that might cause application crash or access to uninitialised memory. These types of pointers are also known as dangling pointers.

Language-integrated regions in Cyclone [2]. One of the goals of the Cyclone programming languages was to eliminate the unsafety of C by introducing additional language constructs to simplify static analysis. One of the constructs authors of the language have added was first-class support for region-based memory:

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

void main() {
    region<`r> h;
    for (i = 0; i < 100; ++i) {
        Point *@region(`r) p = rnew(h) Point(i, i * 2);
        hello(p);
    }
}

void hello(Point *@region(`r) p) {
    printf("Hi, I'm a point (%d, %d)", point->x, point->y)
}
```

As one can see regions are now part of the language and can be used to annotate types of the pointers. Such annotations allow the type system to guarantee absence of dangling pointers in the source code based purely on static region analysis done at compiletime. Moreover, these annotations can be often inferred by the compiler.

Encoding regions

One of the secondary goals of the project was to make regions available in Scala without making any changes to the language. With this in mind, we've decided to use a combination of macros (macro annotations [3] and def macros [4]) and value classes [5] to encode regions.

Due to safety/performance/boilerplate trade-off concerns, we've implemented three models of encoding that pick various combinations of trade-offs: unchecked, dynamic and static models.

Unchecked model. In this model, we've not included any safety precautions against dangling pointer dereference. Since we don't impose any runtime

overhead on checks we will use this model as best-case baseline for the performance comparison:

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

object Main extends App {
  Region { implicit r: Region =>
    for (i <- 0 to 99) {
      val p = Point(i, i*2)(r)
      hello(p)
    }
  }
  def hello(p: Point) =
    println(s"Hi, I'm a Point (${p.x}, ${p.y})")
}
```

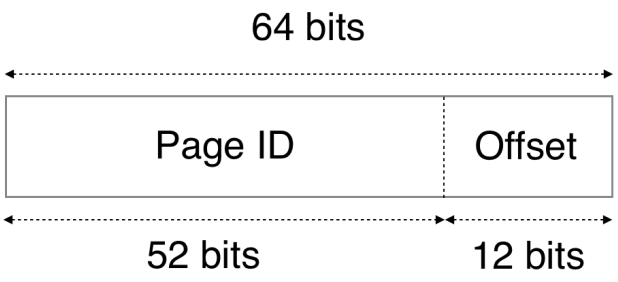
Region scopes are expressed through the `Region { … }` construct which is nothing more than a method call that takes a function in terms of Scala syntax.

Our API provides the @offheap macro-annotation that lets user define classes that can be allocated in regions. This annotation desugars a class into a value class over an off-heap pointer with corresponding field access and instance allocation logic:

```
class Point private (
  private val addr: runtime.Address
) extends AnyVal {
  def x: Int = ...
  def y: Int = ...
}

object Point {
  def apply(x: Int, y: Int)(implicit r: Region): Point = ...
}
```

Here `runtime` calls correspond to the implementation of the region runtime that is going to be discussed in the next section.



Dynamically checked model. This model offers an API equivalent to the one in the previous unchecked model with one major difference: pointers are not actual machine pointers but pointers in virtual address space.
We represent a pointer as a pair of virtual page id and an offset within that page. On every pointer dereference, we are going to verify that the page id corresponds to one of the currently allocated pages. Whenever a pointer dereference over unavailable page happens, we throw an exception rather than going into undefined behaviour.

As we are going to see later in performance comparison, such runtime validation is extremely expensive.

Statically checked model. Lastly, we've implemented a model that ensures absence of dangling pointers at compile-time. The intuition behind this encoding is inspired by Cyclone and ideas of representation of capabilities through implicits.

```
@region(R) class Point(x: Int, y: Int)

object Main extends App {
  implicit val r = Region.open      // r: Region[42]
  for (i <- 0 to 99) {
    val p = Point(i, i*2)           // p: Point[Region[42]]
    hello(p)                        // hello[Region[42]](p)(r)
  }
  r.close
  @region(R) def hello(p: Point[R]) =
    println(s"Hi, I'm a Point (${p.x}, ${p.y})")
}
```

Similarly to Cyclone, we tag each pointer with a region where it was allocated. Each region instantiation has a unique type that lets us guide type inference to automatically infer region types for arguments of methods. Moreover, we require all dereferences of reference types to provide an implicit evidence of the fact that the corresponding region is still open.

The novel side of our approach is the fact that our solution achieves dangling pointer safety without requiring modifications to the host language. We still rely on macro annotations to desugar `@region(R)` annotations. These annotations on methods are desugared into extra type and implicit value parameters:

```
def hello[R <: Region[_]](p: Point[R])(implicit r: R) =
  println(s"Hi, I'm a Point (${p.x(r)}, ${p.y(r)})")
```

Applications of @region to classes augments them with extra type parameter and implicit argument on all methods:

```
class Point[R <: Region[_]] private (
  private val addr: runtime.Address
) extends AnyVal {
  def x(implicit r: R): Int = …
  def y(implicit r: R): Int = …
}
```

The fact that we have an implicit evidence whenever we access a method guarantees that control flow is currently within a given region scope. Therefore it's impossible to dereference a dangling pointer as there would be no implicit evidence available in current scope.

Due to type inference limitations, we can not use scoped syntax any longer and have to let developer perform calls to open and close the region. Due to this change we also need to introduce additional checks in the accessors of the fields of @offheap classes to ensure that region we've got is still open.

Memory management runtime

sun.misc.unsafe. This API is an enabling technology that lets us provide custom memory management on JVM. Among other things, it provides an equivalent of malloc, realloc and free bundled together with type-specific pointer dereference operations:

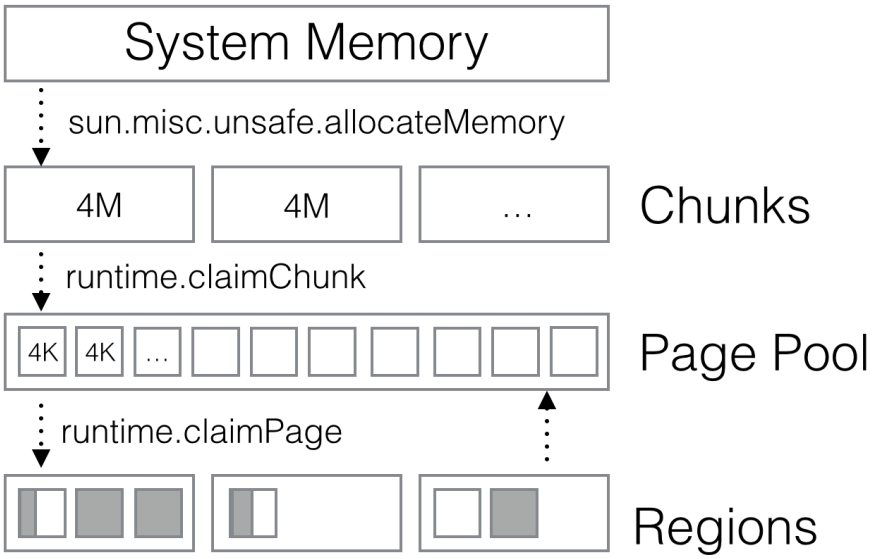
```
package sun.misc

class Unsafe {
    public native long allocateMemory(long bytes)
    public native long reallocateMemory(long address, long bytes)
    public native void freeMemory(long address)
    public native void putByte(long address, byte x)
    public native void putShort(long address, short x)
    ...
    public native byte getByte(long address)
    public native short getShort(long address)
    ...
}
```

We use this API as off-heap memory allocation and access primitives.

Memory re-using runtime. To provide efficient memory allocation and de-allocation we've implemented a runtime that recycles memory allocated from the system. The recycling mechanics of the runtime has been largely inspired by Apache Memory Pools underlying implementation [1].

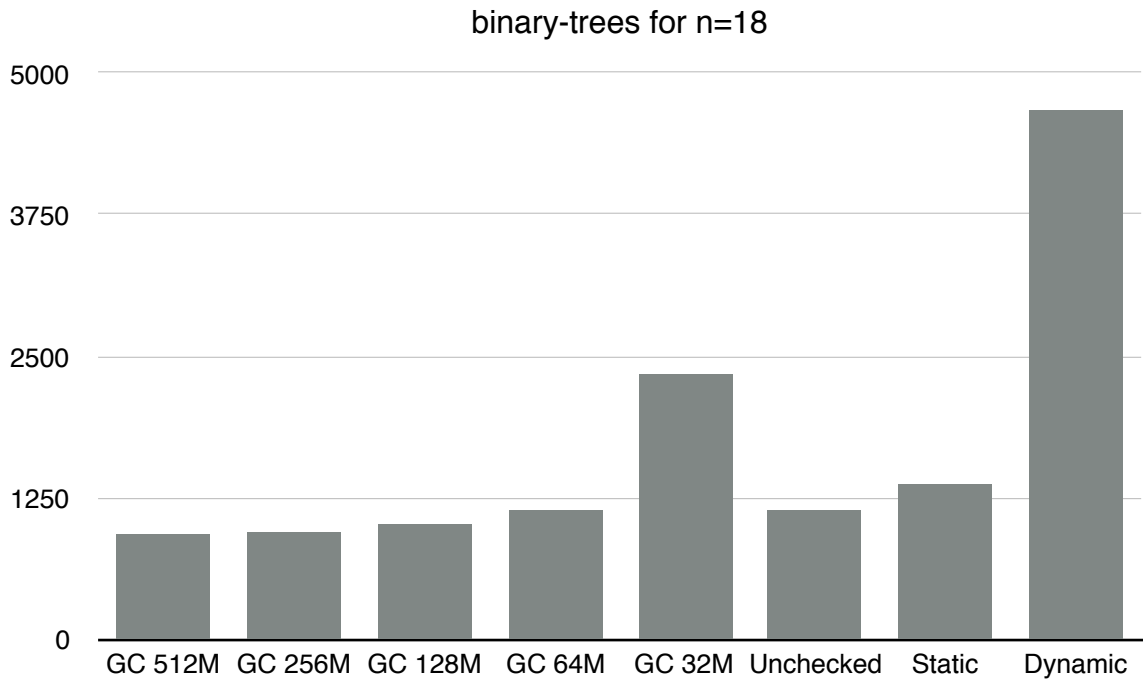
To manage memory, we introduce an additional layer of indirection between system memory and regions that consists of a page pool of pre-allocated memory. This page pool gets filled by occasional 4MB chunk allocations from system memory. Once allocated these chunks are split into 4KB virtual pages, and pointers to those are added to the pool.



In this model, regions consist in a list of pages. The tail of the list contains fully used up pages, and the head of the list services incoming allocations. Once the head is full, a new page is requested from the pool and is then appended to the old head. After the region is closed, the entire list of its pages is returned to the shared pool.

Performance evaluation

To evaluate performance of the *models*, we've implemented we benchmarked them using a widely know memory-taxing binary-trees benchmark [5]. One of the goals of the benchmark is to stress memory management subsystem by repeatedly creating large binary trees and walking them to compute a sum of the underlying nodes.



In addition to benchmarking our models, we've also studied the performance of the current default GC on Oracle JVM's on various heap sizes. This particular benchmark has about 30M worth of allocated objects if objects are allocated on GC heap. GC performs quite well as long as one provides at least twice as much memory as the size of the dataset. As heap size decreases GC starts to perform progressively worse.

Our unchecked model performs similarly to GC with 64M of heap while using only 24M of off-heap memory. Memory savings are caused by the fact that our memory management system neither aligns object nor maintains additional object headers as JVM does.

Statically checked model has some additional overhead over the unchecked one due to implicit argument passing and extra checks needed on those arguments. We believe that most of it can be optimised away by a sufficiently advanced optimisation pass once type inference limitations have been lifted.

Lastly dynamically checked model presents a severe hit to the runtime performance due to hash table lookup on each and every memory access.

Conclusions

In this project, we've presented three encodings of region-based memory in Scala. All of them explore various trade-off one might make: unchecked model offers the best performance at the expense of safety, dynamic model offers additional safety at the expense of performance and, lastly, statically checked model offers safety at expense of additional boilerplate and complexity of the API.

We believe that with some extra work done on improving state of macros and implicits, the static model can become a clear winner out of the three, but at the moment it might be a bit challenging to use for the newcomers.

References

1. Apache Portable Runtime
2. Cyclone
3. Macro annotations
4. Def macros
5. Value classes
6. binary-trees benchmark