Bobby Crimi, Alex Ackerman, Richard Green

Scala Array Problems-rough draft

//ATTN. TA’s: This is a work in Progress. Parts are still in outline form as we are trying to determine what data (quotes, supporting code, etc.) to use because we would rather shape our writing around that. Coming together well so far though!

//Paragraph 1: outlines what paper will be about…uses bullet point 1.

In 2008, there was a great deal of discussion regarding the use of arrays in Scala 2.7. To the non-expert eye, it would appear that they were simply the same as Java arrays. However, this is simply not the case, and if used incorrectly, they can negatively impact performance. The key issues with Scala 2.7 stemmed from the fact that ideally, Scala arrays would have the same representation as in Java so that the data could be used interchangeably. Unfortunately, there are a number of reasons why the 2.7 arrays were unable to do this in light of Java’s particularly low-level representation. First off, Java has multiple array representations which is redundant and can cause a certain amount of ambiguity. Also, while 2.7 technically did have constructors for arrays of a generic type, these constructors caused some issues, such as the fact that various data types would not function as well as using uninitialized arrays. Lastly, there were very few effective methods for array indexing and manipulation. This paper will continue to examine these issues, potential fixes, and what was actually implemented in subsequent releases of Scala in order to avoid some of these problems.

//Paragraph 2/3…use bullet 2. “Ways this can be solved”.

Throughout the Scala Improvement Process, there were a few ways that were presented as possibilities for solving the aforementioned issues with arrays in Scala 2.7. One such way was to create two different implementations of array. In one implementation, there would be the Java representation for interoperation. This implementation would have all of the same traits as an array in Java and have a quick runtime. The other implementation would be the Scala representation for use in the collection hierarchy. One advantage of this second implementation would be that it would have all of the easy-to-use methods for such things as indexing and manipulation. In an ideal world, these two implementations should be interchangeable, with the first one having the performance boost, and the second one being more flexible. However, issues arise when thinking about what would occur with this method when very large pieces of code are being used. Because of the fact that a developer would have to determine which implementation of the array to use, discrepancies could occur if people who are working on the same code choose to use opposite implementations. This could become very complicated as the various chunks of code from different programmers are brought together to complete a program.

Another potential solution that was discussed involved wrapping Java-like arrays in Scala arrays. Therefore, the native arrays would go through a conversion that would essentially make them a part of Scala’s collection hierarchy. By doing this, the uncertainty that arises from having two separate implementations is avoided. In this context, native arrays would have the same implementation as new Scala arrays; however, they would still have the same performance capabilities as Java Arrays.

//Finish this paragraph…remember to talk about string/RichString

In the Scala 2.8 version of Scala Collections, there is a new collections framework which “accompanies collection traits such as Seq with implementation traits that abstract over the representation of the collection.” This allows the base trait to inherit its essential operations from the abstract trait that is instantiated depending on the representation type. Using this new framework, scala arrays can get the speed of the java arrays without sacrificing its place in the collection hierarchy.  
Scala arrays are integrated into new framework via two implicit conversions. The first maps Array[T] to an object type ArrayOps, which will allows programmers to call any Seq method on an array. The returns of these methods will then yield arrays instead of ArrayOps values. And since the intermediate step, converting to ArrayOps, is so brief, modern VM's can reduce the calling overhead to essentially zero. (<http://docs.scala-lang.org/sips/completed/scala-2-8-arrays.html>)  
 If the programmer wants to convert an array into a Seq, another implicit conversion is preformed: the conversion between an array and a WrappedArray. WrappedArrays “are mutable vectors that implement all vector operations in terms of a given Java array”. The operations on a WrappedArray then return a WrappedArray, instead of an Array like ArrayOps does. And then there is an implicit conversion between WrappedArray and Array. Both ArrayOps and WrappedArray inherit from ArrayLike, to avoid the duplication of common code, like operations.

Given the two implicit conversions, there is a balancing as to which is called. The conversion to ArrayOps has precedence, but in some cases an array needs to be converted to a sequence, instead of a sequence method just being called on it. In that case, the WrappedArray conversion is used. This precedence is defined by the policy that: “When comparing two different applicable alternatives of an overloaded method or of an implicit, each method gets one point for having more specific arguments, and another point for being defined in a proper subclass. An alternative 'wins' over another if it gets a greater number of points in these two comparisons.” As ArrayOps is placed in the Predef object, and WrappedArray is in the LowPriorityImplicits class inherited from Predef, WrappedArray will only be used if it's definitely needed.