MapReduce with Aggregators for Scala Collections*

Hendrik van Antwerpen
hendrik@van-antwerpen.net

Draft

Abstract

Functional programming techniques and models like MapReduce have become increasingly popular for distributed processing of large data sets. Google's Sawzall DSL with its use of aggregators is a descendant of this model. We present an implementation of the latter following Lämmal's formalisation. Our implementation provides map-reduce and generic aggregators that integrate seamlessly with the Scala collections library.

1 Introduction

An increasingly popular model for processing big data sets is MapReduce (?). This model lends itself well for distribution and parallel data processing. One descendant of this model is found in Google's Sawzall (?), using aggregators as a key abstraction. In ? a rigorous description of these two models is provided. The Sawzall concept of aggregators is identified as a generalized form of monoids. Their properties allow easy combination of parallel computed partial results, possible in a hierarchical way. Because monoids for common data types like lists, maps and tuples are readily available, developers need think less about the reduction of intermediate results and can focus on the map part of the process.

Sawzall is very liberal in the output it accepts from map functions. The output can be a monoid, a collection of the monoid or an element if the monoid is a collection itself. To capture all these, Lämmel defines aggregators as generalized monoids:

```
class Monoid m => Aggregator e m | e -> m
  where
    minsert :: e -> m -> m
```

To our knowledge no implementation of this approach is available in Scala. Therefore we implemented a version of aggregators for Scala and used those to implement a map-reduce function that integrates seamless with the Scala collection library.

*The content of this paper was earlier published as part of the report 'Visualizing Project Relations on GitHub'.

2 Aggregators

To implement the Sawzall model, we will start with the key abstraction, the aggregator. An aggregator is defined by the following trait:

```scala
trait Aggregator[A,B] {
  def zero: A
  def insert(a: A, b: B): A
}
```

Because there is no readily available monoid type in Scala and we will use aggregators only, we dropped the requirement that type `A` has to be a monoid. We use the term monoid aggregator for any `Aggregator` where `A` and `B` are the same type, because this behaves the same as a monoid of that type would.

We defined the `minsert` function in Scala as `a:A |<| b:B : A`. Whenever this operator is used, the compiler infers which `Aggregator[A,B]` to use. All Scala **`implicit`** rules apply, so if no `Aggregator` is found there's compile time error. If multiple options are found the programmer has to specify manually which one to use.

We started with a basic monoid aggregator for sums, which serves as the default aggregator for integers in the rest of the examples. The implementation looks like:

```scala
implicit def SumMonoid =
  new Aggregator[Int,Int] {
    override def zero = 0
    override def insert(a: Int, b: Int) = a + b
  }
```

The next step is implementing collection aggregators. Generally every collection type has two aggregators. One for the monoid case and one for the element insert case. For the case of lists, the signatures look like

```scala
implicit def ListMonoid: Aggregator[List[A],List[A]] = ...
implicit def ListAggregator: Aggregator[List[A],A] = ...
```

Implementing this for a lot of collection types would be a daunting task. Luckily the design of the generic Scala collections library (?) comes to great help here. By using the `CanBuildFrom[Coll,Elem,Result]` infrastructure and higher-order generics, we only have to implement two aggregators for any kind of `GenSet`, for any kind of `GenSeq` and for any kind of `GenMap`.

As an example, lets look at the aggregators for `GenSeq`, shown in listing 1. The two cases (monoid and element) are implemented separately. Although the implementation is relatively straightforward, some things are note-worthy. In the monoid case, we are more liberal in our input than the type we produce. Every collection implements the `GenTraversableOnce[A]` interface for it's element types and allows easy appending of such a collection. Apart from the case `List [Int] |<| List[Int]` this also allows us to do things like `List[Int] |<| Set[Int]`. Apart from being liberal in the collection type we insert, we are also covariant in the collections element type. This is intuitive for programmers, when a collection takes elements of type `A`, one can also insert elements of a subtype `B <: A`. Because type inferences considers the type `Elem` to be invariant, we added the extra `InElem <:Elem` to allow any subtype as well.

For `Map` there was some extra work to be done. Although `Map[K,V] <:Traversable [(K,V)]` the compile would not infer the aggregator when a map was provided, Listing 1: Aggregators for sequences

```scala
implicit def GenSeqAggregator[Repr[X] <: GenSeq[X], Elem, InElem <:
    Elem]
                              (implicit bf: CanBuildFrom[Nothing,Elem,
                                  Repr[Elem]]) =
  new Aggregator[Repr[Elem],InElem] {
    override def zero: Repr[Elem] = bf().result
    override def insert(s: Repr[Elem], e: InElem) =
      (s :+ e).asInstanceOf[Repr[Elem]]
  }

implicit def GenSeqMonoid[Repr[X] <: GenSeq[X], Elem, In[X] <:
    GenTraversableOnce[X], InElem <: Elem]
                          (implicit bf: CanBuildFrom[GenSeq[Elem],Elem,
                              Repr[Elem]]) =
  new Aggregator[Repr[Elem],In[InElem]] {
    override def zero: Repr[Elem] = bf().result
    override def insert(s1: Repr[Elem], s2: In[InElem]) =
      (s1.++(s2)(bf))
  }
```

like `Map[Int,Int] |<| Map[Int,Int]`. Therefore another monoid aggregator was implemented for this specific case, although still general for any `Map` type.

This brings the count for our collection aggregators to seven and covers all cases. Some examples to show what we can do with it:

```scala
// Int |<| Int : Int
1 |<| 2 // = 3

// List[Int] |<| Set[Int] : List[Int]
List(1,2) |<| Set(3) // = List(1,2,3)

// SortedSet[Int] |<| Int : SortedSet[Int]
SortedSet(2,3) |<| 1 // = SortedSet(1,2,3)

// simple word count coming up!
// Map[String,Int] |<| (String,Int) : Map[String,Int]
Map("aap"->1,"noot"->2) |<| ("noot",1) // = Map("aap"->1,"noot"->3)
```

A nice property of monoids is that they can be structurally combined. For example a tuple with two monoid values is itself a monoid. In our aggregator world, we happen to have similar properties. Aggregators for `TupleN` (see listing 2) and `Map` where implemented to require their tuple values and map value to be aggregators themselves. This is a difference with Lämmels description, where the nested values are strict monoids. Our approach introduces a lot of flexibility in the elements that can be inserted. When creating deep structures, on every level we have the choice to insert either a value or a collection. Some examples should show the flexibility this gives us:

```scala
// count individual and total words
// (Int,Map[String,Int]) |<| (Int,(String,Int)) : (Int,Map[String,Int])
(3,Map("aap"->1,"noot"->2)) |<| (1,("aap",1)) // = (4,Map("aap"->2,"
    noot"->2))

// or (Int,Map[String,Int]) |<| (Int,Map[String,Int]) : (Int,Map[String
    ,Int])
```

Listing 2: Aggregator for tuple

```
implicit def Tuple2Aggregator[A1,A2,B1,B2]
             (implicit a1: Aggregator[A1,B1],
                       a2: Aggregator[A2,B2]) =
  new Aggregator[(A1,A2),(B1,B2)] {
    override def zero = (a1.zero,a2.zero)
    override def insert(a: (A1,A2), b: (B1,B2)) =
      (a1.insert(a._1, b._1),a2.insert(a._2, b._2))
  }
```

Listing 3: Aggregator for collections of elements

```
implicit def GroupAggregator[Coll,In[X] <: GenTraversableOnce[X],Elem]
                            (implicit va: Aggregator[Coll,Elem])=
  new Aggregator[Coll,In[Elem]] {
    override def zero = va.zero
    override def insert(a: Coll, as: In[Elem]) =
      (a /: as)( (c,e) => va insert (c,e) )
  }
```

```
(1,Map("aap"->1)) |<| (3,Map("aap"->1,"noot"->2)) // = (4,Map("aap
    "->2,"noot"->2))
```

One aspect of the Sawzall aggregators is not addressed yet: the possibility to insert a collection of elements. For this case we implemented an aggregator similar to Lämmel's `Group` aggregator, shown in listing 3. Note that this works again at every level of nested types, so this allows us to do things like:

```
// Int |<| List[Int] : Int
3 |<| List(2,5) // = 10

// Map[Int,Int] |<| List[(Int,List[Int])] : Map[Int,Int]
Map(1->1) |<| List((1,List(2,3)),(7,List(42))) // = Map(1->6,7->42)
```

Our design allows a lot of freedom in the shape of the elements inserted into a collection. What type of elements can be inserted is dictated by the defined aggregators and fully inferred by the compiler. The aggregators for the Scala collections are very generic and in most cases the developer doesn't have to care about how to merge collections. Monoid aggregators are implemented for string concatenation and integer summing, but others are very easy to implement; one implicit function and an implementation of the `Aggregator` trait is enough.

3 MapReduce

Using the developed aggregators, a map-reduce library was implemented. We want the map-reduce functionality to be as easy to use as the standard collection functions like `map` or `filter`. To a high degree this can be achieved by enriching libraries, a process similar to defining type classes in Haskell (see **?** for details).

Using again the generic design of the collections library and type inference allows us to write map-reduce by only specifying the expected result type. The full implementation is shown in listing 4. Our first implementation defined `def mapReduce[ResultColl,ResultElem](f: Elem =>ResultElem)`. Unfortunately it is not

Listing 4: MapReduce for Scala collections

```scala
object MapReduce {

  class GenMapReducer[Elem](as: GenTraversableOnce[Elem]){
    def mapReduce[ResultColl] = new MapReducer[ResultColl]
    class MapReducer[ResultColl] {
       def apply[ResultElem](f: Elem => ResultElem)
                (implicit p: Aggregator[ResultColl,ResultElem])
                : ResultColl =
             (p.zero /: as)( (c,a) => p.insert(c, f(a)) )
    }
  }

  implicit def mkMapReducable[Elem](as: GenTraversableOnce[Elem]) =
    new GenMapReducer[Elem](as)

}
```

possible in Scala to provide some of the type parameters but have others inferred. This forced us to repeat the return type of the function when specifying the result type of the `mapReduce` call. Introducing the intermediate `MapReducer` object solved this problem. The result type of mapReduce was specified on the call, but the return type of the function could be inferred for the `apply` call. Because parentheses can be omitted when a function has no arguments, this resulted in a syntax identical to a case without the intermediate object, but with one less type parameter. A word count example similar to one Lämmel gives now looks like:

```scala
val docs: List[String] = ...
def wordcount(doc: String): List[(String,Int)] = doc.split(" ").toList.
    map( w => (w,1) )
val wc = docs.mapReduce[Map[String,Int]]( wordcount )
```

The performance of our map-reduce is in the range as hand written code using e.g. `foldLeft`. Most of the work is done through inference by the compiler. At runtime the aggregators use folds and collection operations just as you would in handwritten code. There a little overhead of some function calls, but all the reduction details are abstracted away, resulting in less repetition and simpler data transformation functions.

It is possible that multiple aggregators for the same type exist, for example a sum and a product on numbers. In such a case it is very easy to specify which one to use. Here is a small example:

```scala
def SumMonoid: Aggregator[Int,Int] = ...
def ProductMonoid: Aggregator[Int,Int] = ...

val words: List[String] = ...
val sumOfWordLengths = words.mapReduce( w => w.size )(SumMonoid)
val prodOfWordLengths = words.mapReduce( w => w.size )(ProductMonoid)
```

As you can see the result type is not required, because it is inferred from the aggregator.

4 Discussion

This fully implements the Sawzall map-reduce model as it is formalized in **?**. The use of aggregators instead of monoids gives us even more freedom in the output of our map functions than Lämmel's model does. The integration with the Scala collections library makes using map-reduce very easy for programmers used to that idiom.

One library that contains similar concepts is the Scalaz[1] library. There are two main differences in approach. First Scalaz tries to provide a very rich and complete library of functional concepts, we tried to focus only on the MapReduce model, keeping the concepts simple. This should make it easier for users to understand and apply the library as well as to implement new aggregators. For our library, one has to implement an `Aggregator` only, where in Scalaz one has to implement a `Zero` and a `SemiGroup`. Second our design allows for much more flexibility than the strict `Monoid` approach of Scalaz allows. It is possible that the `Reducer` from Scalaz addresses the last point, but not enough information was found to be able to judge that.

[1]See `http://code.google.com/p/scalaz/`