Scala BlitzView

semester project report

Axel Angel
EPFL, Switzerland
axel.angel@epfl.ch

June 9, 2014

Abstract

Scala is a powerful language which currently provides a built-in implementation for non-strict views with some important shortcomings for the users such as unexpected and unintuitive behavior.

In this work we created a new library, based on Scala Blitz, to provide lightweight, non-strict and parallel-efficient collections. We present the library API design, implementation and how programmers can use and extend it.

1 Introduction

Scala is a powerful and fast-moving language that fuses object-oriented programming with a wide range of functional programming concepts [6]. It runs on the JVM and a lot of efforts were made to stay compatible with Java and its ecosystem as much as possible. Scala itself provides an important number of libraries, for example Scala collection, which implements Lists, Arrays, Maps and Sets with immutable and mutable variants. They are more in accord within the Scala environment than the Java collections, moreover they provide the functional programming concepts like constructors.

A *View* in Scala is a non-strict version of some collection set. *Non-strictness* here is a mean to postpone computations over a collection until the final result is actually needed, this type of view is called a proxy. The View is said to be *forced* when the postponed computations need to be performed over all the elements. A View captures the operations that are postponed over its inner collection in $O(t)$ memory and stacks them to provide efficient computation in a single pass over the collection $O(n)$, where t is the number of transformers and n the number of elements. In practice this is used when multiple operations, such as multiple `map` and `filter`, are called consecutively. In this case, when the programmer uses regular collections, linear-sized $O(n)$ intermediate collections are generated whereas the overhead of Views is only proportional to the number of transformer operations $O(t)$, generally $t << n$. In our design, Views are reusable because they are *immutable*, and performing a new operation actually returns a new View where all previous operations are captured along the new one. Immutability means our

View's internal state never changes, all versions can be reused independently (as we will show later) and what's more important is that Views do not depend on whether the underlying collection is mutable or not. Immutability greatly simplifies the implementation and opens new possibilities for the programmer to combine and reuse Views in his code.

Therefore a View allows programmers to use special optimisations such as merging these operations to compute them all at once for each element of the inner collection. As the operations are done element by element, we can split the inner collections into a dynamic number of chunks and compute the operations in parallel depending on the number of cores of the computer.

The design of the Views API is primordial because it can greatly limit the optimisations, thus influencing the efficiency of the computations as far as deciding whether they can be done in parallel or not. There exists two types of operations over Views:

Transformers: These can be postponed and captured in the View without evaluating (forcing) the elements, e. g. `map` and `filter`. Usually their type is `[a] -> [b]`.

Folders: These are the last operations that actually force the View to be computed and in general return a single element, e. g. `aggregate` and `max`. Usually their type is `[a] -> b`.

In this work, we focused on a powerful subset of the actual Scala collections API to preserve the efficiency of the Views while providing very powerful and functional non-strict collections.

We begin in section 2 by showing what was done in the past, we explain what views are in 3, we then describe our design in section 4. We continue by demonstrating how our Views are used in section 5 and how they can be extended in section 6. Finally we conclude in section 7 with the source code in appendix A.

2 Previous works

Scala and its collection offer a large toolbox of functions taken from functional paradigm such as `flatMap` and `aggregate` in a object-oriented hierarchy of classes with common interfaces. This collection interface is declared in the parent class `Traversable` [4] which is inherited by multiple types of collection in order to provide a common API that operates uniformly on all these different structures transparently for the programmer: whatever he uses is a `List`, an `Array`, a `Map`, a `LinkedList` or any descent of these classes, they all share this common methods. The programmer has to learn and understand it once, then he can use his experience for any of these collections easily: it's intuitive and greatly increases the productivity. The built-in collections in Scala are strict in the sense that all operations are directly computed because Scala is a strict language, although the programmer can specify the `lazy` variable-keyword, this doesn't solve the problem optimally.

Since Scala 2.8, the Views have joined the built-in toolbox to offer non-strict collections using the common interface of collections. They allow to create a proxy over a collection that captures the operations on them until an operation forces their execution. The purpose of the proxy is to change the evaluation strictness of the collections by handling the computation itself when it sees fit. For example a call to `flatMap` over a View returns immediately whereas over a strict collection this may take some time to return. This is done by implementing all methods of the collection interface in a way the operations are remembered and done when necessary. The wrapper is kept to use non-strict operations. Later, then the programmer can force the conversion into a regular collection: this is done by unwrapping the proxy after the computations and filling a regular collection with the result. This design decision has great advantages when it comes to people experienced with Scala collections because there is no external difference between them. Unfortunately it has two important costs for Scala in terms of the implementation and for the programmer who expect consistent and efficiently results. We will develop these aspects in section 3 and how we approached differently the problem.

Independently, Scala added later *Parallel Collections* to convert collections to parallel variants in order to compute the operations with multiple cores. The thin wrapper is specialized on the underlying collection type; most of the types require no change (constant time) for this conversion. The wrapper provides the same interface with the usual collections, thus there is no difference again in the code after the conversion. The programmer applies the methods as usual, then he can call a method to convert back to the regular collections (unwrapping).

Java 8 was recently released with a new toolbox dedicated to functional programming (e. g. lambda functions) and the new package Stream. These concepts allow the programmer to finally manipulate concisely sequences with `flatMap` and the similar functions well known in Scala. The concept of Stream in Java 8 is different of the Streams in Scala. The Java implementation called Stream is an implementation of non-strict Views, they follow the definition we gave earlier. Scala Streams are quite different, they represent infinite and non-strict sequences, they are usually defined recursively and leverage *memoization* (e. g. the Fibonacci sequence). Moreover the Java Stream can be converted to a parallel variant as the Scala Views, the main difference is that Java implemented a specialized version only for Streams and the interface is very different than the usual Java collections. Java Stream and Scala Views have an important number of common methods such as `flatMap`, `find`, `min`/`max` and the like. They both wrap the inner structure and require the programmer to call specific methods to unwrap, such as `toArray`, or when he calls a folding method. They both require to explicitly convert to non-strict versions and then to parallelized variants if needed. The main difference is that Java has severe limitations with return values depending on Generic types: this is visible for all variants of `flatMap` whose name is postfixed by the type explicitly, e. g. `flatMapToInt` and return a specialized type, e. g. `IntStream`. An important problem with Java Stream is the lack of *transparent referenceability*[1] and reusability. One case is after a terminal operation (Folders) on a Stream: it cannot be reused, it is consumed by the operation (side-effect) and can never be reused. Another case is when Streams are combined together: the programmer cannot use the same Stream twice in a row, for example with `concat`. This limits greatly the combinatorial power of Streams as one needs to create a new Stream for each new use whereas a single View can be reused alone and be part

[1]An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program. – Wikipedia

of other Views.

3 Views

We now define the properties of Views and describe the constraints we must satisfy in our API based on the experience of the previous works.

As we said, Views are non-strict collections and they guarantee *constant* time and *constant* memory for transformers. This is possible because the View (the proxy that wraps the underlying collection) remembers all transformers the programmer requested. As the computations are bookmarked into the view's internals, no change is actually made to the inner collection, these only happen in the proxy. In the current Scala infrastructure, it was decided these Views are immutable, thus each time a transformer is applied on it, a new View is returned. Multiple advantages are offered this way: first the programmer can rely on the immutability, for example he can store multiple Views over the same data without worrying about side-effects on his original collection nor his intermediate Views.

Views should be seen as an adaptor over a collection where each element passes through its pipeline made of operations (the transformers) which are computed and are collected by the last operators (the folder) as they pass by. The choice of whether an operation is a transformer or a folder will depend on the internal implementation of the library.

The problem that interests us in this project was to overcome the limitations seen in the current implementation of Views. One such problem is due to the fact Views inherit from the whole collection API which contains all usual operations that were designed to work on strict and mostly on sequential structures. Although operations such as `permutations` and `sorted` make perfectly sense for the usual collections, these operations cannot be efficiently implemented without actually forcing the View. Despite this fact, this kind of operations is implemented in Scala Views but unexpected results can happen sometimes leading to an exception, the surprised programmer should rather not use these methods at all.

```
xs.view.map{x => println(x); x}.sorted
// each number is printed
// it returns a View
```

The programmer may not expect the numbers to be printed now because he didn't request any result yet. However `sorted` is forcing the View behind the scene and create a new sorted View; this is an unexpected and unintuitive behavior. In fact, other operations share the same issue like `groupBy`, `intersect`, `sortBy` and the likes.

There exist other operations that don't play nice when they are used on Views, and this is a problem for programmers who expect the least surprise. For example `flatten` does not return a flatten View but a new List containing the result of the flattening, even if we use Views inside and outside. This operation should have been non-strict as well.

Other important problems with Scala Views arise when we want to combine non-strictness with Par-parallelism. The following is permitted although it doesn't have the same intuitive behavior:

```
val xs = (0 to 1000).par.view
val ys = (0 to 1000).view.par
```

Which one is correct? Are they equivalent? In fact they are not, worse, the first loses its parallelism, while the second loses its non-strictness. From these problems we can see there is a lack of coordination between these APIs. Scala collections offer too many methods that cannot be efficiently implemented or that do not make sense in a non-strict context. Moreover the combination of both Views and Par should be done in a unified way to avoid these problems. We have noticed a similar bad interaction between Views and Stream in Scala, it would be better for the View API to disallow such uses.

4 Design

In this work we propose an alternative implementation of Scala Views that solves the issue of coordination between available methods and efficiency in non-strict and parallelized context.

The first design decision we made is to create a new interface, a trait, that does not contain problematic methods. There are different types of such methods: some are inherently sequential (e. g. `reduceRight`), some require forcing the View (e. g. `ordered`), some are inefficient anyway (e. g. `permutations`) and some are possible but trickier to implement (e. g. `takeWhile`).

We focused our prototype on the most important ones:

- `[a] -> [b]`: `map`, `filter` which are transformers.
- `[a] -> b`: `aggregate` is the most important. It is the building block for the other folders such as `min`, `sum`, `find`, `exists`, `count` which are folders.

The transformers are represented by the trait `ViewTransform[-A, +B]`, in our internal implementation; this is inspired by the work of Martin Odersky in a prototype [5] (inspired by Reducers of Rich Hickey [2]). It represents a function from A to B where A is contravariant and B covariant. This trait is used to pipeline operations when we are folding: we first apply the transformers, then we apply the given folder (this is the purpose of the method `fold`). The important property of these transformers is that they are recursive: a transformer can contain another transformer and so on; this is done with `>>`. In our design, there are three types of transformers: `Map`, which applies a function on each element, `Filter`, which drops elements according to the given predicate function, and `Identity`, which is the identity transformer (this is used at the bottom of the stack).

Here is the hierarchy of classes for our design:

`BlitzView`: The top trait that describes the available operations (transformers and folders) on all Views. It contains all the methods we just discussed above.

`BlitzViewImpl`: The trait below that contains most of our implementation. Anyone is free to create a new implementation next to it, see section 6. This trait inherits `BlitzView` and provides the common implementation of all methods for subclasses in terms of the method `genericInvoke`. The children classes then must only implement this method to inherit all operations of this design.

`BlitzViewC`: The View that contains a single underlying collection. This is the class that is used as a proxy closest to a wrapped collection and the one that actually captures operations in a stack. It inherits `BlitzViewImpl`.

`BlitzViewVV`: The View that concatenates two Views together. When a new transformer is added, it is passed to the two inner Views. This is the result of `++` on two Views. It inherits `BlitzViewImpl`.

`BlitzViewFlattenVs`: The View that contains a list of View and concatenates the elements together in a single flattened View. When a new transformer is added, it is passed to every inner Views. It inherits `BlitzViewImpl`.

A second important design choice we made is to use ScalaBlitz[2] for the actual computations. This library offers first-class collections in terms of performance because it was designed for efficiency by using parallelism and specialized code to avoid unnecessary boxing. We won't cover the details of the internal algorithms but it suffices to say the computations are dynamically spread among the workers according to the programmer policy. This is the concept of *work stealing*, which is implemented in this library; more details can be found in recent papers [3].

The library itself provides the usual high-level operations on the major collections we need (`Array`, `Range`, `Map` and `Set`). Although we could have used multiple calls for transformers, then reducers, we decided to extend ScalaBlitz with a new method (`mapFilterReduce`) that we used to implement our internal methods (such as `genericInvoke`). This new function combines a `flatMap` (map and flatten at once) and a general `fold` in a single step. In practice, that means folding a View only require a single iteration over the underlying collection, each element is only used once. This property stays true even as the number of transformers increase, this won't be true for regular collection transformers without optimisation. Moreover, in our design the programmer gains parallelism for free, this is fully integrated by the use of Scala Blitz whose algorithm can be configured: by importing some careful chosen `implicits` (which form the *context*).

[2]Homepage: http://scala-blitz.github.io/

4.1 Implicits

An interesting part of Scala API design consists of using implicits methods or values [8]. They can help to augment classes of certain shapes with more operations and sometimes they can provide a way to construct values given a number of possible underlying representations. We will now show how we used both of these two mechanisms to design a fluent and powerful API for the programmer. The first type augments specific type shape and is referred as *implicit-extensions*, while the latter type, where we construct a class based on the representation, is referred as *implicit-evidence* (like a proof).

In our design implicit-extensions are used to allows the programmer to flatten a View, by just calling `flatten` on it. Even though there is no such method anywhere in our public API, the programmer can call it when it makes sense to do so. The requirement is that the View contains itself Views inside and as such there is no particular class that provides `flatten`, it's just a plain `BlitzView[B]` where `B` is the type of the elements. In this case `B` has a special shape: `B = BlitzView[C]`, and the implicit-extension is triggered when the user calls `flatten`. The Scala compiler searches for an implicit conversion, then, provided the requirements are satisfied, it's applied automatically to produce what we designed: a special hidden class that has the `flatten` method, in our case `ViewWithFlatten`. To implement such implicit conversion we need an implicit method that specifies the constraint and the conversion (see addFlatten). Then the call to flatten (now on ViewWithFlatten) returns a special View instance that concatenates all its inner Views, in our case BlitzViewFlattenVs.

The second part with implicit-evidence is used to create Views, that happens when the programmer calls bview on any supported collection. We support an interesting subset of Scala Blitz collections (see above) but we decided to evict Lists because they cannot be used efficiently in a parallel context, and it's easy for users to convert them to Array anyway. We created a "proof-performer" that, given a suitable evidence, can convert a collection to a View; here the evidence is an implicit value (of type IsViewable). The "proof-performer" is an implicit conversion, called toViewable. There is at least one evidence per collection we support, each one requiring an implicit Scala Blitz context (to decide how to parallelize the collection) and some requiring an implicit ClassTag (to decide how to pack in Arrays). When applied, the "proof-performer" returns an ephemeral instance of Viewable whose sole purpose is to augment the collection with the bview method. In practice our "proof-performer" is called only when the bview method itself is called, thus the class Viewable is only an internal detail of our implementation.

5 Usability

We now talk about the programmer's perspective when using our View implementation: how to create Views, how usual operations are performed and the extent of possibilities with our prototype.

Let's first take an example to illustrate the creation and use of Views:

```
val xs = (0 to 10).toArray
val v = xs.bview
val u = v.map(_ + 10)
```

The user already familiar with Scala built-in Views will notice the similarity: the only difference is bview instead of view. One needs to import our package: collection.views.Scope._ and a Scala Blitz context[3].

The programmer has the guarantee xs will never be affected by the actions he is performing on the Views, here v or u. Moreover u is independent of u and both can be used as many times as needed.

The very nice properties of Views are important because they increase the possible use cases. For example Views can be used with mutable collections (without any special treatment): just create a View on an underlying mutable HashMap for example, this can be seen as a view in SQL, over any table (collection) where the View is always synchronized with the underlying data changes. In our prototype, the View stays up to date because the computation is always done from scratch each time.

```
import collection.mutable.HashMap
val m = HashMap((1,2))
val v = m.bview.map{case (x,y) => x+y}
v.toArray // Array(3)
v.sum // 3
m += ((3,4))
v.toArray // Array(7, 3)
v.sum // 10
```

Let's take a concrete example, let's say we have a collection of departments, each containing people. We want to compute the ratio of people having certain properties, like people over a certain age union[4] whose name begins with an A:

```
case class Person(n: String, a: Int)
// xs: MSet[(String, MSet[Person])]
```

[3]e. g. par.Scheduler.Implicits.sequential

[4]Note that in our example we use ++ which counts duplicates twice

```
val v = xs.map(_._2.bview).bview.flatten
val vf = v.filter(_.a > 30)
      ++ v.filter(_.n[0] == 'A')
vf.size / v.size
```

There are multiple remarks necessary to understand the purpose of this code. First we used `MSet`, a mutable `Set` so we can modify the collection as the number of people come and go. Second we explicitly require the programmer to convert inner collections to Views, which is necessary if the user wants to `flatten` the structure; this is to avoid unnecessary work (for example when the programmer does not need to use View inside) and to avoid problematic implicit conversions (some conversions change the type which wouldn't be desirable all the time). Third there is a major difference with the built-in Scala Views shown here: `flatten` does return a View whose inner Views are flattened, that means the `flatten` operation keeps our non-strict semantic; all the other operations following it (here `size`) are on the always-synchronized Views. The programmer can continue to update the `MSet` and still use `v` and `vf` to get the desired result.

Scala Blitz offers different schedulers for parallelism based on work-stealing such as `Sequential` (no parallelism), `ForkJoin` (kernel pool), and even the programmer can create new ones. The scheduler is an implicit that can be imported or explicitly passed to the methods of our Views.

6 Extensibility

We now present how a programmer can extend our hierarchy to create new implementation or new classes and how well it is integrated seamlessly.

The programmer can create a new class under `BlitzViewImpl` that implements a certain shape of Views. The advantage of creating a class that inherit our implementation is there are only two methods to implement:

- `>>`: this method must save the provided transformer into its state, depending on the case it should propagate this to the children Views.
- `genericInvoke`: this method is responsible for the application of the transformers followed by the folding. This method is called by all others in the public API, this allowed us to keep children classes very thin where most of the implementation resides in the common heritage (`BlitzViewImpl`).

Then, to use it, the programmer can create a new implicit-extension if this should be used for certain shapes of Views.

We take a toy example: let's implement a View that contains a single element: `BlitzViewS`[5]. We use `BlitzViewC` as code base, having a hidden underlying type `A` (for its source) and a transformers stack `transform`, and now we need to store a single element `x` (instead of a collection `xs`). Our implementation of `>>` stays the same whereas `genericInvoke` needs to only apply the transform on `op` (as usual) but on the single element to return it (we don't need to fold here). The trick here is to pass an empty `ResultCell` as our folder's second argument (it plays the role of an *identity* element).

We can now create either an implicit-evidence for the singleton type, for example an Int:

```
import collection.views.ViewTransforms._
implicit def intIsViewable =
  new isViewable[Int, Int] {
    override apply(i: Int) =
      new BlitzViewS[Int] {
```

[5]We provide this implementation in the code repository

```
        type A = Int
        val x = i
        def transform = new Identity()
    }
}
```

The programmer can now write: `5.bview` as expected. There is an other similar implementation, provided as a more interesting example, for `Option` in `BlitzView0`, this allows the programmer to `flatten` `BlitzView[Option[T]]` like with regular collections.

Implicits allow the programmer to plug directly its own classes as first-citizens of our prototype without changing our prototype.

7 Conclusion

Views are a powerful concepts that can improve the efficiency and productivity when they are well implemented. The programmer can apply multiple transformers consecutively, reuse and pass around the intermediate Views, fold them multiple times. All these operations are very efficient because we use non-strictness, optional and configurable parallelism. We limited the scope of available operations (transformers and folders) on purpose to the ones that can be efficiently implemented. This may seem very restrictive but most of the usual methods on collections are still offered, moreover the programmer can always switch back to regular collections and call whatever he wanted, may it be `permutations` or `dropWhile`. In our current prototype these methods were not implemented due to the reasons above but this may change in the future, as this is more easily done than for built-in Scala Views. Due to the fact that our prototype is a separate library, it can make progress on its own, be updated more regularly and we avoid the problem of built-in packages: even with an old version of Scala, one can use our prototype by just importing it.

An other important aspect of libraries is the public part, the methods available to the final programmer. Scala Views decided to stick with the collections API but it was not designed for non-strict Views, it resulted in certain undesirable behavior and unnecessary problems. Additionally, separating Scala Views from Parallel Collection has caused additional confusion as it is often unclear whether they can or cannot work together.

One major insight is that generally APIs cannot be changed without breaking most of the programmer code, and now Views decided to use the Scala collections API without built-in parallelism. The extent of the API problem is even wider than before but hopefully there are multiple ways to improve the situation: changing the public API and breaking code (not acceptable for Scala), or by creating standalone libraries that provide better APIs by learning from the errors of the past designs.

The prototype we presented here is a design that overcomes most of the difficulties that current Views suffer: standalone and suitable API, lightweight wrappers in a small codebase, configurable parallelism[6] and powerful extensibility using implicits.

7.1 Future work

Future work can start to expand the number of Views type, the available methods in the API, generalize to different parallel libraries, add smart memorization and use macro expansions aggressively.

Currently our prototype implements cer-

[6]Our prototype does depend on Scala Blitz but in such a way it's easily changeable.

tain underlying type: `C` (collections), `VV` (concat two Views), `FlattenVs` (flatten Views), `S` (singleton), `O` (`Option`). There is room for improvements by adding more types, for example generators (functions that return the elements lazily).

We implemented methods such as `flatMap`, `map`, `filter`, etc. and there are still other methods missing in our library like `tail`, `partition`, `zip`, `takeWhile` that may have efficient implementations in our model. Especially `tail` that could be implemented with `head` with a lazy `reverse` operation. `partition` and `zip` can certainly be implemented with a `map` followed by a special folder, the tricky part is the interaction of elements from multiple Views simultaneously. For methods like `takeWhile`, the problem is tricky if we want an implementation that works efficiently in parallel, due to the fact that chunks depend on neighbor results, it may be possible but it was not in the scope of our prototype.

As we said, our prototype use the parallelism of Scala Blitz by calling certain methods specialized to the underlying collections. Our dependency is more explicit than possible and we could abstract away the information to avoid this issue[7].

Our current prototype does not cache the result of the folders nor the folders. Memoization could be trivially added after transformer computation for immutable underlying collections at the expanse of linear $O(n)$ memory overhead for our Views and more complex codebase. With mutable underlying collections, this is trickier as it can change anytime, there may exist ways to ask the underlying collection whether it changed, so we could propagate a dirty flag in our Views. Unfortunately this may not be possible, moreover it could add unforeseen issues to our prototype if not well thought[8].

Our last proposition for improvement is macros. They were introduced recently, in Scala 2.10, to allow metaprogramming, writing code that generates code at compile-time without sacrificing modularity [1]. Scala Blitz use them extensively to propagate types statically and work on unboxed types when possible, this has an important impact on performance. We could leverage macros in our prototype to pass statically more information to Scala Blitz so it can create specialized code, thus we would benefit from the maximal performance offered by its design. Furthermore we could use macros to simplify consecutive transformers by inspecting their code to merge them, e. g. `map{_+1}.map{_+1}` into `map{_+2}`. Macros open up new dimensions of specialized optimization by code inspection.

A Source code

The prototype is available on GitHub, commit `v0.1`: `https://github.com/axel-angel/scala-blitzview` (with code samples). We would like to thank Dmitry Petrashko for his continuous great support, Martin Weber for proof-reading, Martin Odersky and the LAMP laboratory for their inspiring and great researches.

References

[1] Eugene Burmako. Scala macros: let our powers combine!: on how rich syn-

[7] We could store the `Scheduler` but it proved to be a mistake in other libraries, thus for now the signatures depend on it.

[8] This posed some issues in Scala Views because it is not controllable by the programmer.

tax and static types work with metaprogramming, 2013.

[2] Rich Hickey. Reducers: Library and model for collection processing. http://clojure.com/blog/2012/05/08/reducers-a-library-and-model-for-collection-processing.html, 2012. [Online; 2014-06-04].

[3] Aleksandar Prokopec Tiark Rompf Phil Bagwell Martin Odersky. A generic parallel collection framework. Technical report, EPFL Lausanne, Switzerland, 2010.

[4] Martin Odersky. Scala 2.8 collections. Technical report, EPFL Lausanne, Switzerland, 2009.

[5] Martin Odersky. scalax: Parallel views. https://github.com/odersky/scalax, 2013. [Online; last commit 6f74549e].

[6] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[7] Martin Odersky and Adriaan Moors. Fighting bit rot with types, 2009.

[8] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *SIGPLAN Not.*, 45(10):341–360, October 2010.