# Abstract

Scala is a powerful language which currently provides a built-in library for non-strict Views with some important shortcomings for the users such as unexpected and unintuitive behaviors. 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 function programming concepts [**?**]. It runs on the JVM and as such it stays compatible with Java and its ecosystem. 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 post-pone 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 computation need to be performed over all the elements. A View captures the operations that are postponed over its inner collection in constant memory $O(1)$ and stacks them to provide efficient computation in a single pass over the collection $O(n)$.

In practice this is used when multiple operations, such as multiple `map` and `filter`, are called consecutively. As Views are usually *immutable*, as in our design, performing a new operation actually returns a new View where all previous operations are captured along the new one. Immutability greatly simplifies the implementation and open new possibilities for the programmer to combine and reuse Views in his code.

Therefor a View allows us 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 and not. There exists two types of operations over Views:

**Transformers:** These can be postponed and captured in the View without evaluating (forcing) the elements, eg: `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 returns a single element, eg: `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.

# 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` [?] 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 increase the productivity. The built-in collections in Scala are strict in the sense 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 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 force it. The purpose of the proxy is to change the evaluation strictness of the collections by handling the computation itself when it see fits. 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 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 the section and how we approached differently the problem.

Independently, Scala added later a way to convert collections to parallel variants in order to compute the operations with multiple cores. The thin wrapper is specialized depending on the underlying type, most of the types requires 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 (eg: 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 first is the Java implementation of the non-strict Views we discussed above whereas Scala Streams are infinite non-strict sequences, usually defined recursively. 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 Generics type: this is visible for all variants of `flatMap` whose name is postfixed by the type explicitly, eg: `flatMapToInt` and return a specialized type, eg: `IntStream`. An important problem with Java Stream is the lack of transparent referenceability and reusability: 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. This limits greatly the combinatorial power of Streams as one needs to create new Stream for each new use whereas a single View can be reused alone and be part 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 internals, no change is actually made to the inner collection, these only happen in the proxy. In the current Scala infrastructures, 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 interest 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` makes perfectly sense for the usual collections, these operations cannot be efficiently implemented in the case of Views. Despite this fact, this kind of operations is available in current Views but they are not correctly implemented which can make it crash, leading to an unexpected behavior:

```
val xs = 0 to 3
xs.permutations.toArray
// correct result
xs.view.permutations.toArray
// UnsupportedOperationException
```

There exists other operations that doesn't play nice when they are used on Views and this is a problem for programmer 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 op-

eration should have been non-strict as well.

Other important problems with Scala Views arise when we want to combine non-strictness of Views with parallelism of Par together. The following is permitted although one version doesn't make sense:

```
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 equivalent, worse, the second version seem to loose its non-strictness.

From these problems we can see there is a leak of coordination between the collection API and the way we can construct parallelized Views. Scala collections offer too many methods that cannot be efficiently implemented or that does not make sense in a non-strict context, and the use of both Views and Par together should be done in a unified way to avoid these problems.

# 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 (eg: `reduceRight`), some require forcing the View (eg: `ordered`), some are inefficient anyway (eg: `permutations`) and some are possible but trickier to implement (eg: `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 from the work of Martin Odersky in a prototype [**?**]. 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 feature of these transformers are they are recursive: a transformer can contain an other transformer and so on, this is the purpose of `>>`. 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` used at the bottom of the stack.

Here is the structure of the objects hierarchy for our Views:

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

`BlitzViewImpl` : contains the trait for 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 a method `aggInternal`. The children classes then must only implement this method to inherit all operations of this design.

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

`BlitzViewVV` : is the View that concatenates two Views together. This would be the result of `++` on two Views. It inherits `BlitzViewImpl`.

`BlitzViewFlattenVs` : is the View that contains a list of View and concatenate the elements together in a single flattened View. It inherits `BlitzViewImpl`.

A second important design choice we made is to use ScalaBlitz[1] 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. Some novel ideas developed in a paper [**?**] such as *work stealing* are implemented in this library. We won't cover the details of the internal algorithms but suffices it to say the computation are dynamically dispatched among the processors according to the programmer policy.

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 augment ScalaBlitz with a new method (`mapFilterReduce`) that we used to implement our internal methods (such as `aggInternal`). This new function combines a `flatMap` (map and reduce 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*).

An interesting part of Scala API design contains implicits methods or values [**?**]. 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 representation. 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 is referred as *implicit-extensions* and the second type where the 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 to represent this, 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 kicks when the user calls `flatten`. The Scala compiler searches for an implicit conversion then, provided the requirements are satisfied, it's applied behind the scene to produce what we designed: a special hidden class that has the `flatten` method, in our case `ViewWithFlatten`. To implement

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

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 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 requires an implicit Scala Blitz context (to decide how to parallelize the collection) and some require an implicit `ClassTag` (to decide how to pack in `Array`s). 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 is itself called, thus the class `Viewable` is only an internal detail of our implementation.

# 5 Usability

We now talk about the programmer perspective when using our View implementation: how to create Views, how usual operations are performed and the extend 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[2].

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 time 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 are always done from scratch each time.

```
import collection.mutable.HashMap
val m = HashMap((1,2))
val v = m.view.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 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[3] people whose name begins with an A.

```
case class Person(n: String, a: Int)
```

[2] For example `par.Scheduler.Implicits.sequential`

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

```
// xs: MSet[(String, MSet[Person])]
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 depart. Second we explicitly require the programmer to convert inner collections to Views, 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 which inner Views are flattened, that means the `flatten` operation keeps our non-strict semantic, all 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.

`aggInternal:` 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 shape of Views.

We take a toy example: let's implement a View that contains a single element: `BlitzViewS`[4]. We use `BlitzViewC` as code base, it has: a hidden underlying type `A` (for its source), 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 `aggInternal` 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 provide an empty `ResultCell` to our folder 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] {
```

[4]We provide this implementation in the code repository

```
    override apply(i: Int) =
      new BlitzViewS[Int] {
        type A = Int
        val x = i
        def transform = new Identity()
      }
}
```

The programmer can now write: `5.bview` as expected. There is a similar but more interesting example of implementation for `Option` in `BlitzViewO`, it can be `flatten` as for regular collections.

# 7 Conclusion

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

# References

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

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

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

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

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