Scala BlitzView Presentation

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

June 10, 2014

Table of Contents

Plan

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.

What are Views?

Definition (View)

A non-strict version of one or more collections.

Definition (Non-strictness)

Evaluation is postponed until the result is actually needed.

Non-strictness allows lazy-evaluation for our View operations (transformers). Use-case: lots of successive operations on elements.

What do Views contain?

Views **capture** operations (contract):

- $O(tn)$ memory, stack of transformers
- $O(n)$ operations: efficient computation in a single pass over the inner collection

where t is the number of transformers and n the number of elements.

Definition (Forced View)

The View is said to be *forced* when the postponed computations are performed over all the elements.

Operations

Different type of operations:

Transformers: Postponed and **captured** in Views (no forcing).
e. g. `map` and `filter`. Type: `[a] -> [b]`.

Folders: Last operations, **forcing**.
e. g. `aggregate` and `max`. Type `[a] -> b`.

We focused on a subset of operations (on purpose!)

Plan

Based on existing ideas

Learn from past works:

Scala Collections: Lots of methods like `flatMap` and data structures like `List`, `Array`, `Map`, `Set` with (im)mutable variants. Strict.

Scala Views: Conversion from Scala Collections to Views, same interface. Immutable. Non-strict.

Java 8 Streams: Like Scala Views, unlike Scala Streams. Non-strict. Semi-immutable (not transparent-referenceable[1]).

[1] Can be replaced with its **value** without changing the behavior

What's good and what's not so good

Comparisons:

Scala Collections: +Lots of methods, +Strict, −Non-parallel[2];

Scala Views: −Lots of methods, +Non-strict, −Non-parallel[3];

Java 8 Streams: +OK methods +Non-strict −Non-parallel[4] −Limitations of Generics[5] −Poor usability.

[2] Require explicit conversion `Par`

[3] `Par` has unexpected behavior

[4] Require explicit conversion: `parallelStream`

[5] One per type: `flatMapTo*`

Plan

Issues

What's problematic:

- Scala common interface inherited from Collections;
- Requiring explicit parallel conversion;
- Scala unexpected behavior between Views and Par;
- Java limitations and usability problems;
- Built-in vs "Hackability": extension.

Solution

`BlitzView`, or how we solved them:

- Scala common interface inherited from Collections
  ⇒ Specific interface for Views
- Requiring explicit parallel conversion
- Scala unexpected behavior between Views and Par
  ⇒ Built-in configurable parallelism (on operations)
- Java limitations and usability problems
  ⇒ Scala type system and immutability
- Built-in vs “Hackability”: extension
  ⇒ Separate package and implicits conversions

Problem of interface

Scala common interface inherited from Collections[6]:

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

Other operations are **forcing** like `groupBy`, `intersect`, `sortBy` and the likes.

[6] In fact `Traversable`

Problem of parallelism

Requiring explicit parallel conversion;
unexpected behavior between Views and Par:

```
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.

Solution

```
trait BlitzView[+B] { // some have [A >: B] for covariance
  /* operators */
  def ++/:::(ys: BlitzView[B]): BlitzView[B]
  def ::(y: A): BlitzView[B]

  /* methods (transformers): V -> V (hidden: map + filter) */
  def flatMap[C, U](f: B => U)(implicit ctx: Scheduler, viewable: IsViewable[U, C]): Bli

  /* methods: V -> other array structure */
  def toArray(implicit ct: ClassTag[A], ctx: Scheduler): Array[A]
  def toList(implicit ctx: Scheduler): List[A]

  /* methods: V -> V[constant type] */
  def toInts/Doubles/...(implicit f: Numeric[A]): BlitzView[Int]

  /* methods: V -> 1 (with Optional variants) */
  def aggregate[R](z: => R)(op: (A, R) => R)(reducer: (R, R) => R)(implicit ctx: Schedul
  def reduce(op: (A, A) => A)(implicit ctx: Scheduler): A
  def min/max(implicit ord: Ordering[A], ctx: Scheduler): A
  def sum/product(implicit num: Numeric[A], ctx: Scheduler): A

  /* methods: V -> 1[constant type] */
  def size(implicit ctx: Scheduler): Int
  def count/find/exists/forall(p: B => Boolean)(implicit ctx: Scheduler): Int
}
```

Remarks

Properties:

- Sufficient powerful methods implemented (but no more[7])
- Implicits to configure parallelism (ScalaBlitz[8] `Scheduler`)
- **Modularity** using common implementation trait
- **Extensibility** using implicits (supported collections)

[7] Some says: "less is more"

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

Modularity design

OOP Hierarchy:

- BlitzView trait (interface)
  - BlitzViewImpl trait (interface+implementations) subclasses implement >> and `genericInvoke`
    - BlitzViewC(ollection)
    - BlitzViewVV(iews)
    - BlitzViewFlattenVs
    - BlitzViewO(ption)
    - BlitzViewS(ingleton)
    - (Open to new implementations)
  - (Open to new design)

Design Internals

```
/** BlitzView implementation with a single source par Collection. */
abstract class BlitzViewC[B] extends BlitzViewImpl[B] { self =>
  type A // type of source list
  val xs: Reducable[A] // source list
  def transform: ViewTransform[A, B] // stack of transforms

  override def >>[C](next: ViewTransform[B, C]) = new BlitzViewC[C] {
    type A = self.A
    val xs = self.xs
    def transform = self.transform >> next
  }

  override def genericInvoke[R](op: (B, ResultCell[R]) => ResultCell[R]
    , pstop: ResultCell[R] => Boolean)
    (reducer: (R, R) => R)(implicit ctx: Scheduler): ResultCell[R] =
  {
    val stopper = ViewUtils.toStopper(pstop)_
    xs.mapFilterReduce[R](transform.fold(op), stopper)(reducer)(ctx)
  }
}
```

Design Internals

```
trait ViewTransform[-A, +B] { self =>
  type Fold[A, F] = (A, ResultCell[F]) => ResultCell[F]

  def fold[F](g: Fold[B, F]): Fold[A, F]

  def >>[C](next: ViewTransform[B, C]) = new ViewTransform[A, C] {
    def fold[F](fd: Fold[C, F]): Fold[A, F] =
        self.fold(next.fold(fd))
  }
}

class Map[A, B](m: A => B) extends ViewTransform[A, B] {
  def fold[F](fd: Fold[B, F]): Fold[A, F] =
    (x, acc) => fd(m(x), acc)
}

class Filter[A](p: A => Boolean) extends ViewTransform[A, A] {
  def fold[F](fd: Fold[A, F]): Fold[A, F] =
    (x, acc) => if (p(x)) fd(x, acc) else acc
}
```

Extensibility design

Different types of implicits:

Implicit-extension: Automatic conversion to an extended "proxy" class under certain conditions. Useful to add methods "on-the-fly" depending on type shape. Our use-case: `addFlatten`.

Implicit-evidence: An implicit conversion that requires an (implicit) evidence to. One evidence per supported collection type. Our use-case: `bview`.

Without changing code: easy to add methods to our design, add new supported collections (and View type). Disadvantage: less intuitive ("black-magic" effect).

Implicit-extension

An example of our use-case:

```
val v: BlitzView[BlitzView[Int]] =
  Array[Option[Int]](Some(1), None).map{_.bview}
v.flatten // implicit-extension
```

Behind the scene:

```
implicit def addFlatten[U, B[_] <: BlitzView[_], C[_] <: BlitzView[_]](view: B[C[U]])
  = new ViewWithFlatten[U, B, C](view)

class ViewWithFlatten[U, B[_] <: BlitzView[_], C[_] <: BlitzView[_]]
(val view: B[C[U]]) extends AnyVal
{
  def flatten()(implicit ctx: Scheduler, ct: ClassTag[U]): BlitzView[U] =
    new BlitzViewFlattenVs[U, B, C] {
      val zss = view
    }
}
```

Implicit-evidence

An example of our use-case:

```
Array[Int](1, 2, ..., N).bview
```

Behind the scene:

```
implicit def toViewable[L, A](col: L)(implicit evidence: IsViewable[L, A]) =
  new Viewable[L, A](col)(evidence)

trait IsViewable[L, A] {
  def apply(c: L): BlitzView[A]
}

class Viewable[L, A](col: L)(evidence: IsViewable[L, A]) {
  def bview: BlitzView[A] = evidence.apply(col)
}

/* all supported collections have an evidence */
implicit def arrayIsViewable[T](implicit ctx: Scheduler) =
  new IsViewable[Array[T], T] {
    override def apply(c: Array[T]): BlitzView[T] = {
      View(c.toPar)(new Array2ZippableConvertor)
    }
  }

/* ... */
```

Plan

Future directions

There are more:

- Add new type of Views (Generators?)
- Extend API (keep least surprise)
- Generalize to other parallel libraries
- Add smart memorization
- Macro expansions (further optimization)

Insights

What I have learned.

Thanks

Thanks for your attention!

Prototype on GitHub:

`https://github.com/axel-angel/scala-blitzview`

Includes code samples for free! :)