SCALA 2.13 COLLECTIONS REWORK

FEBRUARY 25, 2017

WRITTEN BY: *Stefan Zeiger*

In October of 2015 Martin Odersky asked for strawman proposals (https://github.com/lampepfl/dotty/issues/818) for a new collections library design for Scala 2.13, which eventually led to the project that we are currently working on, based on his latest proposal. This was not the first redesign for the Scala collections. The current design (http://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html) was first implemented in Scala 2.8 along with the required improvements to type inference in the Scala compiler. It can generally be considered a success, providing powerful and flexible abstractions that bring together immutable and mutable collections, both sequential and parallel, with a high amount of shared interfaces and implementations. However, it does exhibit some symptoms of second-system syndrome (https://en.wikipedia.org/wiki/Second-system_effect) that have been problematic in practice.

Goals

Before looking at details of these problems and possible solutions in the new design, let's start with the broader goals for the new design:

- Simplify the API for users: This includes doing common operations without `CanBuildFrom`, pruning back the inheritance tree and a better separation of immutable and mutable collection operations.

- Simplify the API for implementors: Implementing a new collection type is far from trivial (http://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html) at the moment. Setting up the implicits to get the desired `CanBuildFrom` instance in all cases (both through static implicits lookup and at run-time) is tricky. Inheriting lots of method implementations by default appears like a benefit at first but you still need to manually check (and possibly override) all methods whose default implementation has unsatisfactory performance for a specific collection.

- Design for better performance: The best algorithms are not very useful if you ignore lower-level performance concerns such as specialization for primitive types or method dispatch in the JVM. For example, replacing Scala collections in Slick's AST (https://github.com/slick/slick/pull/1252/commits/2adb7c36874c41f068176570d3812b674463660e) by a custom collection implementation improved the performance of Slick's query compiler by 25% (from reducing overhead and implementing efficient operations alone, without any use of specialization). While we do not expect the same improvements for a more generic collection library, it does show that there is room for improvement.
- Provide source compatibility with 2.12 where it makes sense but allow breaking changes where required: We expect the majority of collection usage to be compatible between 2.12 and 2.13 and the majority of the remaining incompatibilities to be automatically fixable with ScalaFix (https://scalacenter.github.io/scalafix/).

Traversable and Iterable

Leaving the "generic" abstractions (which encompass both sequential and parallel collections) and the implementation traits (like `TraversableLike` and `IterableLike`) aside, we have `Traversable` at the root of the current collections hierarchy. Its only real subtype that is used by collection implementations is `Iterable` . The difference is that `Traversable` only provides *internal iteration*, i.e. a `foreach` method, whereas `Iterable` gives you the more powerful *external iteration* via an `Iterator` . The `Traversable` abstraction has not carried its weight in the current library and will likely not resurface in the new design. Everything we want to do can be expressed with `Iterable` .

We are also looking at other opportunities to remove collection traits. While each one of them is there for a good reason, their interactions and the sheer number create a huge amount of complexity. For example, this is the class declaration of the standard `List` (http://www.scala-lang.org/api/2.12.1/scala/collection/immutable/List.html) class:

```
sealed abstract class List[+A] extends AbstractSeq[A]
                                with LinearSeq[A]
                                with Product
                                with GenericTraversableTemplate
                                with LinearSeqOptimized[A, List
                                with scala.Serializable {
  ...
}
```

If you traverse all these supertypes you find no less than 37 (!) linear supertypes in total.

CanBuildFrom

One of the most powerful but also most controversial features of the current collections library is `CanBuildFrom`:

```
    def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr
```

This is the standard `map` method as declared in `TraversableLike`. It is so inscrutable to beginners that simplified *use case signatures* were added to the API documentation in order to better convey the intended meaning:

```
    def map[B](f: A => B): $Coll[B]
```

While use case signatures help you when you look at the scaladocs, you still have to deal with the real definitions in IDE code-completion and in compiler errors.

Here is the same method as defined in `IterablePolyTransforms` in the new design:

```
   def map[B](f: A => B): C[B]
```

The method has the expected signature, there is no compile-time or run-time overhead to find the right `CanBuildFrom`, and the type constructor `C` is refined to a concrete collection type in almost every use case, so you see the expected definitions in scaladocs, code-completion and error messages.

Of course, `CanBuildFrom` was added in Scala 2.8 for a good reason. It allows a single definition of a standard method like `map` to work for regular, unconstrained collections and for constrained collections that require an implicit evidence for their element type. For example, we can naturally define `class BitSet extends Set[Int]`, but what does it mean to call `map` on a `BitSet`?

```
val s: BitSet = ...
val c1 = s.map(i => i+1)
val c2 = s.map(i => i.toString)
```

The current Scala collections design makes it possible to not only build a `BitSet` for `c1` and a `HashSet` for `c2` at run-time but also compute these types statically at compile-time. It gets slightly more complicated when you take code such as

```
val s: BitSet = ...
val c1 = (s: Set[Int]).map(i => i+1)
```

You wouldn't expect `c1` to be of type `BitSet` but is it still backed by a `BitSet` at run-time or do you get a default `Set` implementation? In this case it is the latter but there are inconsistencies between static and dynamic lookup of `Builder` types in the current library that can lead to unexpected types or implementations.

As shown above, the new design does not have an implicit evidence parameter for `map`, so you are assured that you always get the same implementation no matter if you statically see your `BitSet` as a `BitSet` or a `Set[Int]`. This "same implementation" in the case of `BitSet` means `Set` though. A collection implementation is free to pick any collection type to build (between what its supertype promises and what it implements itself, of course) as long as it is *unconstrained*. In order to call `BitSet.map` and get another constrained `BitSet` out of it, we need to overload the `map` method:

```
class BitSet extends Set[Int] {
  // inherited:
  //def map[B](f: Int => B): Set[B]

  def map(f: Int => Int): BitSet
}
```

Thanks to an improvement to type inference (https://github.com/scala/scala/pull/5307) it is possible to call the overloaded method with a lambda without explicit type annotations in Scala 2.12:

```
val s: BitSet = ...

// Scala 2.11 and earlier would have required:
val c1 = s.map((i: Int) => i+1)

// This works in 2.12+:
val c2 = s.map(i => i+1)
```

While `BitSet` is a rather esoteric collection type, the same principle of implicit constraints on element types `T` applies to other collection types as well:

- BitSet: `T <:< Int` ("must be an Int")
- All `Map` types: `T <:< (_, _)` ("must be a Tuple2")
- All sorted collections (like `TreeSet`): `Ordering[T]` ("must have an Ordering")
- String (not a collection type per se but it gets many collection methods as extension methods): `T <:< Char` ("must be a Char")

This covers most uses of `CanBuildFrom` in a much simpler way. The most important use cases that cannot be supported by the new design are `collection.breakOut` (for building a different collection type directly without an extra conversion step) and `to` (for converting to a different collection type for which a `CanBuildFrom` is available).

FromIterable

The `to` method still exists in the new design but it is only a decorator around `FromIterable` , the basic abstraction that is implemented by every unconstrained collection type's companion object:

```
1.
trait IterableOps[+A] extends Any {
  def to[C[X] <: Iterable[X]](fi: FromIterable[C]): C[A @unche
    fi.fromIterable(coll)
  ...
}

trait FromIterable[+C[X] <: Iterable[X]] {
  def fromIterable[B](it: Iterable[B]): C[B]
}
```

Note that the `fi` parameter is not implicit (like the `CanBuildFrom` in the current library), so you now call `to` with a value and have the type inferred instead of calling it with a type and having the value filled in by implicit lookup:

```
1.
val s: Set[Int] = ...
val v = s.to(Vector) // this used to be to[Vector]
```

This has the advantage that we can overload `to` to cover maps (and maybe also other constrained collection types) which is not possible in the current design (where `to` only works for unary type constructors).

Views

Views (http://docs.scala-lang.org/overviews/collections/views) in the current collections library are one of the lesser-used features (https://github.com/scala/collection-strawman/issues/21#issuecomment-275861238) yet they add a lot of complexity to the implementation. The new design separates immutable views from mutable views (the latter have not yet been implemented), with only two types of immutable views: indexed and non-indexed. This is a huge simplification over the current design where a view type is specific to a its underlying collection type.

Conceptually a `View` is now a reified operation over an `Iterator`. Like a Java 8 Stream (https://zeroturnaround.com/rebellabs/java-8-streams-cheat-sheet/) it has *terminal* operations which run the operation represented by the current `View` on a fresh `Iterator` (e.g. `foreach and foldLeft`) and *intermediate* operations which create a new `View` (e.g. `filter and flatMap`). This provides clear semantics even when used together with mutable collections: Composing views with intermediate operations is always independent of concurrent modifications to the underlying collection. Only when you call a terminal operation will the current state of the collection be used (by calling `iterator` on it).

Views are also the recommended replacement for `collection.breakOut`. For example,

```
val s: Seq[Int] = ...
val set: Set[String] = s.map(_.toString)(collection.breakOut)
```

can be expressed with the same performance characteristics as:

```
val s: Seq[Int] = ...
val set = s.view.map(_.toString).to(Set)
```

If you combine multiple operations they are executed lazily:

```
val s: Seq[Int] = ...

// First build a filtered Seq, then a mapped Seq, then a Set
val set1 = s.filter(_ > 0).map(_.toString).to(Set)

// Build a Set directly by evaluating filter and map together
val set2 = s.view.filter(_ > 0).map(_.toString).to(Set)
```

Laziness

Aside from views the current collections library also supports lazy collections but they do not get a first class treatment. While we do have `Stream`, it is only lazy in the tail but still strict in the head element and the implementation needs to special-case pretty much everything because the basic abstraction for building new collections (which is used by all of the default implementations that strict collection types can safely inherit) is `CanBuildFrom` / `Builder` which uses strict, push-based collection building.

By combining `FromIterable` with the new `View` design we can turn this around to provide pull-based building which can be used by strict and lazy collections alike. For example, this is the default implementation of `map` in `IterablePolyTransforms` which does not need to be overridden in `LazyList` to provide the expected laziness:

```
  def map[B](f: A => B): C[B] = fromIterable(View.Map(coll, f
```

Any operation that is available on `View` can get a default implementation like this based on rebuilding the current collection type from a `View` operation. Naturally a `LazyList` can be "built" from an `Iterable` by getting an `Iterator` and only pulling elements one by one as they are needed, so we automatically get a lazy implementation of `map`.

Language Integration

You may have wondered about the asymmetry in the default types that are in scope through `Predef` or the `scala` package object:

1.

```
scala> classOf[Set[_]]
res1: Class[Set[_]] = interface scala.collection.immutable.Se

scala> classOf[Map[_, _]]
res2: Class[Map[_, _]] = interface scala.collection.immutable.

scala> classOf[Seq[_]]
res3: Class[Seq[_]] = interface scala.collection.Seq
```

Unlike all other collection types, `Seq` is not the immutable version but the generic one that encompasses both, mutable and immutable sequences. The reason for this is that the Scala specification represents varargs as type `scala.Seq` (and it should not have to rely on any type outside the top-level `scala` package). Since varargs in Java (and on the JVM) are really mutable arrays, the default `Seq` has to allow mutable collections.

In practice though, this kind of array can be treated as quasi-immutable, so we plan to add a new `ImmutableArray` wrapper in the new design which can be used for varags, thereby removing the need for a non-immutable default `Seq` type.

Outlook

We are currently working on the new design as part of SCP-007 (https://github.com/scalacenter/advisoryboard/blob/master/proposals/007-collections.md) which is funded by Scala Center (https://scala.epfl.ch/). At this stage all work is happening in the collection-strawman (https://github.com/scala/collection-strawman) repository with the goal of refining and completing it to the point where we are confident that it can and should become a part of Scala 2.13. The core project team consists of Julien Richard-Foy (from Scala Center), Martin Odersky, Rex Kerr and myself (as a member of the Scala team at Lightbend who maintain the Scala compiler).

If you'd like to get involved, now is the time to weigh in on the discussions that are happening on the pull requests and issues. We also have a Gitter channel (https://gitter.im/scala/collection-strawman) and a Scala Contributors (https://contributors.scala-lang.org/t/ongoing-work-on-standard-collections-redesign/293) discourse discussion thread.

A few topics still need further exploration:

- Currently there is no support for specialization (http://www.scala-notes.org/2011/04/specializing-for-primitive-types/) of collections. It would be nice to allow this in the new design if we can do it without too much of an impact on the majority of non-specialized collections.
- We need a story for parallel collections. They will be moved into a separate module (https://github.com/scala/scala/pull/5603/) in Scala 2.13 as part of the ongoing modularization of the standard library but it is not clear yet how closely they will be integrated into the new design.
- The scala-java8-compat (https://github.com/scala/scala-java8-compat) module provides better integration of collections with Java Streams. Some basic parts like the specialized `Stepper` types (which unify Java Iterators, Scala Iterators and Java Spliterators) may find their way into the standard library.
- Apart from ScalaFix (https://scalacenter.github.io/scalafix/) we should explore other migration options, for example using IntelliJ IDEA's refactoring API or providing compatibility libraries that allow you to cross-build most sources against the old and new collection libraries.

The current roadmap calls for the basic design to be completed in Q1 2017 so we can focus on migration options in Q2 and come to a decision whether to adopt the new design in Scala 2.13. If all goes according to plan, the remaining implementation work should be finished by the end of the year in time for Scala 2.13.0-RC1.

DOCUMENTATION

Getting Started (/documentation/getting-started.html)
API (http://www.scala-lang.org/api/current/index.html#package)
Overviews/Guides (http://docs.scala-lang.org/overviews/)
Tutorials (http://docs.scala-lang.org/tutorials/)
Language Specification (/files/archive/spec/2.12/)

DOWNLOAD

Current Version (/download/)
All Versions (/download/all.html)

COMMUNITY

Community (/community/)
Mailing Lists (/community/index.html#mailing-lists)
Chat Rooms & More (/community/index.html#chat-rooms)
Libraries and Tools (/community/index.html#community-libraries-and-tools)
The Scala Center (https://scala.epfl.ch/)

CONTRIBUTE

How to Help (/contribute)
Report an Issue (/contribute/bug-reporting-guide.html)