Fighting Bit Rot with Types (Experience Report: Scala Collections)

M. Odersky[1], **A. Moors**[2*]

[1] EPFL, Switzerland
martin.odersky@epfl.ch

[2] K.U.Leuven, Belgium
adriaan.moors@cs.kuleuven.be

ABSTRACT. We report on our experiences in redesigning Scala's collection libraries, focussing on the role that type systems play in keeping software architectures coherent over time. Type systems can make software architecture more explicit but, if they are too weak, can also cause code duplication. We show that code duplication can be avoided using two of Scala's type constructions: higher-kinded types and implicit parameters and conversions.

1 Introduction

Bit rot is a persistent problem in most long-running software projects. As software systems evolve, they gain in bulk but lose in coherence and clarity of design. Consequently, maintenance costs increase and adaptations and fixes become more complicated. At some point, it's better to redesign the system from scratch (often this is not done and software systems are left to be limping along because the risk of a redesign is deemed to high).

At first glance it seems paradoxical that bits should rot. After all, computer programs differ from other engineering artefacts in that they do not deteriorate in a physical sense. Software systems rot not because of rust or material fatigue, but because their requirements change. Modifying a software system is comparatively easy, so there's a low threshold to accepting new requirements, and adaptations and extensions are common. However, if not done right, every such change can obscure the original architectural design by introducing a new special case.

Two aspects of software systems tend to accelerate bit rot: lack of explicit design and code duplication. If the design of a system is not made explicit in detail it risks being undermined by changes down the line, in particular from contributors who are new to the system. Code duplication, on the other hand, is problematic because necessary adaptations might apply to one piece of code but might be overlooked in a duplicate.

In this paper we explore how a strong static type discipline affects bit rot, using the Scala collection library as a case study. A collections library is interesting because it provides a wide variety of operations, spread over several different interfaces of collections, and over an even larger number of implementations. While there is a high degree of commonality

*Supported by a grant from the Flemish IWT.

Foundations of Software Technology and Theoretical Computer Science (Kanpur) 2009.
Editors: Ravi Kannan and K. Narayan Kumar; pp 427–451
Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.
Digital Object Identifier: 10.4230/LIPIcs.FSTTCS.2009.2338

among collection interfaces and implementations, the details vary considerably. Thus, extracting the commonalities is at the same time necessary and non-trivial.

At first glance, a static type system looks like a good basis for a robust collections library because it can make design decisions explicit and checkable. On the other hand, if the static type system is not flexible enough to capture some common pattern, it might force conceptually sharable code to be repeated at each type instance. In the Scala collections we experienced both of these effects. The first Scala collection library was designed with a standard repertoire of generics and nominal inheritance and subtyping, close to what is found in Java or C#. This made a number of constraints explicit, but forced some code to be duplicated over many classes. As the number of contributors to the code base grew, this duplication caused a loss of consistency, because additions were either not done in the most general possible context, or necessary specialisations in subclasses were missed.

We recently set out to redesign the collection libraries with the aim of obtaining at the same time better architectural coherence and better extensibility. The redesign makes critical use of two advanced forms of polymorphism available in Scala: higher-kinded types and implicit parameters and conversions. Higher-kinded types allow to abstract over the constructor of a collection, independently of its element type. Implicits give a library author the means to define new type theories which are adapted to the domain at hand. Both played important roles in cleaning up the collections design.

In this paper we explain the architecture of the original collections library, and how we addressed its shortcomings in the new Scala 2.8 collections. We then present the architecture of Scala 2.8 collection framework, and show how it can be extended with new kinds of collections. We also explain how higher-kinded types and implicits help in making the new design explicit and checkable and in keeping extensions uniform and concise.

Related work The generalisation of first-order polymorphism to a higher-order system was a natural step in lambda calculus [6, 18, 2]. This theoretical advance has since been incorporated into functional programming languages. For instance, the Haskell programming language [8] supports higher-kinded types, and integrates them with type classes [9], the Haskell approach to ad-hoc polymorphism. However, to the best of our knowledge, Scala is the only object-oriented language to integrate support for higher-kinded types. We call this feature "type constructor polymorphism" [13]. Altherr et al. have proposed integrating this into Java [4].

Implicits serve two purposes in Scala: they allow for retroactive extension using the "pimp-my-library" pattern [15], and they extend the language with support for ad-hoc polymorphism. Implicits are the minimal addition to an object-oriented language that is required to encode Haskell's type classes, and thus support that style of ad-hoc polymorphism. They are more local than type classes in that the applicability of an implicit is controlled by scope rules, similarly to the modular type class proposal for ML [5]. A type-class like extension has also been proposed for Java [21].

Ad-hoc polymorphism is similar to parametric polymorphism in the sense that it allows operations to be applicable at varying types, except that, whereas parametrically polymorphic operations are truly indifferent to the concrete type that they are applied to, ad-hoc polymorphic operations take the specific type into account and vary their behaviour accordingly. Java's static overloading is a minimal implementation of this abstraction mechanism, whereas Haskell type classes [20] allow for expressing much richer abstractions.

The literature on the design of collection frameworks has traditionally concentrated on the Smalltalk language. The "blue book" [7] contains a description of Smalltalk's original collection hierarchy. Cook [3] analyses the interfaces inherent in that library which are often not expressed directly in Smalltalk's single-inheritance hierarchy. Ducasse and Schärli describe the use of traits to refactor the Smalltalk collection libraries [1]. Our experience confirms their conclusion that composition of traits is an important asset in the design of such complex libraries. Scala traits differ from their formulation [19] in that Scala traits combine aspects of symmetric trait composition with aspects of linear mixin composition. Nevertheless, the applicability of both forms of traits for modelling collections stays the same. Of course, Smalltalk is dynamically typed, so none of the previously cited related works addresses the question how to type collections statically. Naftalin and Wadler describe Java's generic collections [14], which are largely imperative, and do not offer higher-order functional operations, so that they pose less challenges to the type system.

Structure of the paper Section 2 gives a quick introduction of the parts of Scala necessary to understand the examples in the rest of this paper. Section 3 presents the original collection framework as it existed before the redesign and highlights its shortcomings. The next two sections introduce key abstractions that form the foundation the new collections library. Section 4 shows how to reduce code duplication by abstracting over the representation type of the collection, as well as over how to traverse and build it. Section 5 refines this to abstractions over type constructors. However, neither approach suffices. Section 6 illustrates that we need ad-hoc polymorphism — piece-wise defined type functions — and introduces implicits as a solution. Section 7 discusses in detail how implicits express piece-wise defined type functions and integrates them with builders. Section 8 outlines the Scala 2.8 collections hierarchy, and shows how new collection implementations can be integrated in the framework, illustrating the kind of code re-use that is achieved. Section 9 explains how the pre-existing primitive classes for arrays and strings can be integrated in the collections framework. Section 10 concludes.

2 Syntactic Preliminaries

In Scala [16, 17], a class can inherit from one other class and several other traits. A trait is a class that can be composed with other traits using mixin composition. Mixin composition is a restricted form of multiple inheritance, which avoids ambiguities by linearising the graph that results from composing traits that are themselves composites of traits. The difference between an abstract class and a trait is that the latter can be composed using mixing inheritance[†]. We will use "class" to refer to traits and classes alike, but, for brevity, we will use `trait` instead of `abstract class` in listings.

Identifiers in Scala may consist of symbolic as well as regular identifier characters. Method calls like `xs.++(ys)` or `xs.take(5)` have more lightweight equivalents: `xs ++ ys` and `xs take 5`.

[†]The restrictions imposed on traits to allow mixin composition are not relevant for this paper.

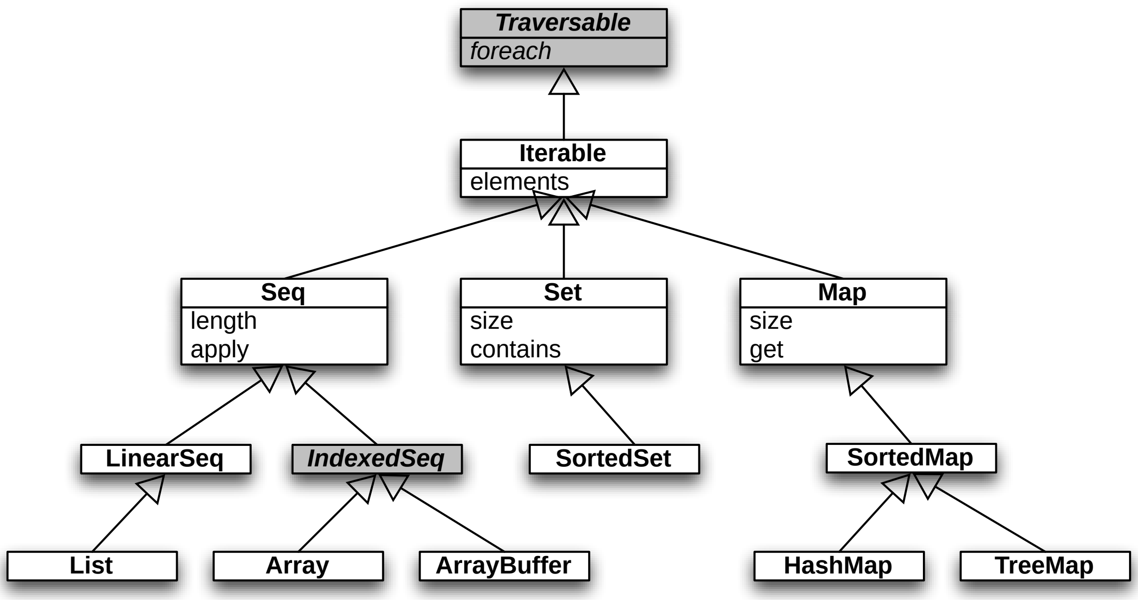

Figure 1: Some Scala Collection Classes (2.8-specific classes are shaded)

Functions are first-class values in Scala. The type of functions from T to U is written $T \Rightarrow U$. A function literal is also written with an infix "⇒ ", e.g. `(x: Int)⇒ x + 1` for anonymous successor function over type `Int`. Type inference often allows to elide the argument type of a function literal, as in `x ⇒ x + 1`. Alternatively, and even shorter, the parameter position may be marked with an underscore, as in `(_ + 1)`. Internally, functions are represented as objects with `apply` methods. For instance, each of the three above function literals is expanded to the object

```
new Function1[Int, Int] {
  def apply(x: Int) = x + 1
}
```

Conversely, function application notation `f(e)` is available for every object `f` with an `apply` method, and is in each case equivalent to `f.apply(e)`.

3 Status Quo

Scala collections are characterised by four properties: they are *object-oriented*, *optionally persistent*, *generic*, and *higher-order*.

Object-oriented Collections form a hierarchy, sharing common operations in base traits. Figure 1 gives an outline of the collections hierarchy as it existed in Scala until version 2.7, including some of the new classes from Scala 2.8, which have been shaded. At the top of the original hierarchy is trait `Iterable`, which represents a collection by means of an `elements` method that allows iterating over its elements. Specialisations of `Iterable` are

`Set` for sets, `Map` for maps, and `Seq` for sequences. Some of these abstractions have further specialisations.

For example, the classes `SortedSet` and `SortedMap` represent sets or maps which are *sorted*, meaning that their iterators return their elements in the natural order of the element type. Each collection abstraction has multiple implementations. Sequences in trait `Seq` can be linked lists, arrays, list buffers, array buffers, or priority queues, to name but a few. The class hierarchy gives rise to a subtyping relation (<:) between collections. For instance, `Set` is a subtype of `Iterable`, so that a set can be passed wherever an iterable is expected.

Most operations on collections are represented as methods in the collection classes. For instance, to retrieve an iterator for the elements in `c`, one calls `c.iterator`. The length of a sequence `s` can be queried using `s.length`, and `s(i)` (short for `s.apply(i)`) returns its `i`'th element.

Optionally persistent Most collection abstractions in the library exists in two forms: mutable and immutable. Immutable collections are also called "persistent"; they offer operations that create new collections from existing ones incrementally, leaving the original collections unchanged. For instance, `xs ++ ys` creates a new sequence which consists of all elements of sequences `xs`, followed by all elements of sequence `ys`. The sequences `xs` and `ys` remain unchanged. Or, `m + (k -> v)` creates a new map that augments map `m` with a new key/value binding. The original map remains again unchanged. Mutable collections introduce operations that change the collection in place. For instance, `m.update(k, v)` updates a map at key `k` with the new value `v` (this can be expressed shorter as `m(k) = v`).

Generic Most collections are parametric in the type of their elements. For instance, the type of lists with pairs of integers and strings as elements is `List[(Int, String)]`, and `Map[String, List[String]]` represents a map that takes keys of type `String` to values of type `List[String]`. The interaction between subtyping and generics is controlled by variance annotations. Most persistent collection types are covariant, whereas all mutable collections are nonvariant.

Variance defines a subtyping relation over parameterised types based on the subtyping of their element types. For example, **`class`** `List[+T]` introduces the type constructor `List`, whose type parameter is *covariant*. This means that `List[A]` is a subtype of `List[B]` iff `A` is a subtype of `B`. With a *contravariant* type parameter, this is inverted, so that **`class`** `OutputChannel[-T]` entails that `OutputChannel[A]` is a subtype of `OutputChannel[B]` iff `A` is a *supertype* of `B`. Without an explicit variance annotation, type arguments must be equal for the constructed types to be comparable.

Some collections restrict their type parameter. Sets backed by red-black trees, for example, are only defined for element types that can be ordered.

Higher-order Many operations on collections take functions as arguments. Examples are: `c.foreach(f)`, which applies the side-effecting function `f` to each element in `c`, the collection of the elements in `c` that satisfy the predicate `p` can be computed as `c.filter(p)`,

```
trait Iterable[+A] {
  def filter(p: A ⇒ Boolean): Iterable[A] = ...
  def partition(p: A ⇒ Boolean) = (filter(p(_)), filter((!p(_))))
  def map[B](f: A ⇒ B): Iterable[B] = ...
}

trait Seq[+A] extends Iterable[A] {
  override def filter(p: A ⇒ Boolean): Seq[A] = ...
  override def partition(p: A ⇒ Boolean) = (filter(p(_)), filter((!p(_))))
  override def map[B](f: A ⇒ B): Seq[B] = ...
}
```

Listing 1: Some methods of the `Iterable` and `Seq` traits

and `c.map(f)` produces a new collection with the same size as `c`, where each element is the result of applying `f` to the corresponding element of `c`.
These operations are defined uniformly for all collections. When they return a collection result, it is usually of the same class as the collection on which the operation was applied. For instance if `xs` is a list then `xs map (_ + 1)` would yield another list, but if `xs` was an array, the same call would again yield an array. The following interaction with Scala REPL shows that this relationship holds for static types as well as computed values.

```
scala> val xs = List("hello", "world", "!")
xs: List[java.lang.String] = List(hello, world, !)

scala> xs map (_.length)
res0: List[Int] = List(5, 5, 1)

scala> val ys = Array("hello", "world", "!")
ys: Array[java.lang.String] = Array(hello, world, !)

scala> ys filter (_.length > 1)
res1: Array[java.lang.String] = Array(hello, world)
```

Base traits like `Iterable` offer the same operations as their concrete implementations, but with the base trait as result type. For instance, the following REPL interactions show that applying `map` on an `Iterable` will yield `Iterable` again as the static result type (even though the computed value is a subtype).

```
scala> val zs: Iterable[String] = xs
zs: Iterable[String] = List(hello, world, !)

scala> zs map (_.length)
res2: Iterable[Int] = List(5, 5, 1)
```

Ideally, a collections framework should also be highly *extensible*. It should be easy to add new kinds of collections, or new implementations of existing collections. However, the combination of genericity and immutable higher-order operations makes it difficult to achieve good extensibility. Consider the `filter` method of trait `Iterable` in Listing 1, which must be specialised in `Seq` so that it returns a `Seq`. Every other subclass of `Iterable` needs a similar re-implementation. In the original collection library such implementations had to be provided explicitly by the implementer of a collection class.

Methods that could in principle be implemented uniformly over all collections also need to be re-implemented. Consider for example the `partition` method of Listing 1, which splits a collection into two sub-collections of elements according to whether they satisfy a predicate `p`. This method could in principle be implemented just once in `Iterable`. However, to produce the correct static return type, `partition` still has to be re-implemented for every subclass, even though its definition in terms of `filter` is the same in each class.

The problem becomes even more challenging with an operation like `map`, also shown in Listing 1. The `map` method does not return exactly the same *type* as the type it was invoked on. It preserves the *type constructor*, but may apply it to a different element type.

Overall, these re-implementations pose a significant burden on collection implementers. Taking sequences as an example, this type of collection supports about a hundred methods, of which 20 return the collection type itself as some part of its result, like `filter` and `partition` do, and of which another 10 return the collection type constructor at a different element type, like `map` does. Every new collection type would have to re-implement at least these 30 methods.

In practice, this made maintaining and extending the library quite difficult. As the collection implementation evolved and the number of its contributors increased, it lost more and more of its consistency. Some operations would be added only to a specific subclass, even though they could in principle apply to more general collection types such as `Iterable`. Sometimes, a specific implementation would fail to re-implement some of the methods of the general collection class it inherited from, leading to a loss of type precision. We observed a pronounced "broken windows" effect: classes of the library that already contained ad-hoc methods would quickly attract more such methods and become more disorganised, whereas classes that started in a clean state tended to stay that way. Over the course of some years the coherence of the collection design deteriorated to a state where we felt a complete redesign was needed.

The intention was that the collection library redesign should largely keep to the original APIs in order to maintain a high degree of backwards compatibility, and also because the basic structure of these APIs proved to be sound. At the same time, the redesign should provide effective guards against the kind of bit rot that affected the previous framework. In the rest of this paper we explain how this goal was achieved and which of Scala's more advanced type constructs were instrumental in this.

4 Abstracting over the Representation Type

To avoid code duplication, collection classes such as `Traversable` or `Seq` inherit most of their concrete method implementations from an implementation trait. These implementation traits, which are denoted by the `Like` suffix, form a shadow hierarchy of the client-facing side of the collections that were depicted in Figure 1. For example, `SeqLike` is the implementation trait for `Seq` and `TraversableLike` underlies `Traversable`.

Listing 2 outlines the core implementation trait, `TraversableLike`, which backs the new root of the collection hierarchy, `Traversable`. The type parameter `Elem` stands for the

```
package scala.collection
trait TraversableLike[+Elem, +Repr] {
  protected[this] def newBuilder: Builder[Elem, Repr] // deferred
  def foreach[U](f: Elem ⇒ U)                         // deferred

  def filter(p: Elem ⇒ Boolean): Repr = {
    val b = newBuilder
    foreach { elem ⇒ if (p(elem)) b += elem }
    b.result
  }
}
```

Listing 2: An outline of **trait** `TraversableLike`

```
package scala.collection.generic
class Builder[-Elem, +To] {
  def +=(elem: Elem): this.type = ...
  def result(): To = ...
  def clear() = ...
  def mapResult[NewTo](f: To ⇒ NewTo): Builder[Elem, NewTo] = ...
}
```

Listing 3: An outline of the `Builder` class.

element type of the traversable whereas the type parameter `Repr` stands for its representation. An actual collection class, such as `List`, can simply inherit the appropriate implementation trait, and instantiate `Repr` to `List`. Thus, clients of `List` never see the type of the underlying implementation trait. There are no constraints on `Repr`, so that it might be instantiated to a type that is not a subtype of `Traversable`. Therefore, classes outside the collections hierarchy such as `String` and `Array` can still make use of all operations defined in this implementation trait.

The two fundamental operations in `Traversable` are `foreach` and `newBuilder`. Both operations are deferred in class `TraverableLike` to be implemented in concrete subclasses. The `foreach` operation takes a function parameter that is applied to every element in the traversable collection. The result of the function paraneter is ignored, so functions are applied for their side effect only. The `newBuilder` operation creates a "builder" object, from which new collections can be constructed. All other methods on of `Traversable` access the collection using `foreach`. If they construct a new collection, they always do so through a builder.

Listing 3 presents a slightly simplified outline of the `Builder` class. One can add an element `x` to a builder `b` with `b += x`. There's also syntax to add more than one element at once, for instance `b += (x, y)` to add the two elements `x` and `y`, or `b ++= xs` to add all elements in the collection `xs`. The `result()` method returns a collection from a builder. The state of the builder is undefined after taking its result, but it can be reset into a new empty state using `clear()`. Builders are generic in both the element type `Elem` and in the type `To` of collections they return.

Often, a builder can refer to some other builder for assembling the elements of a collection, but then would like to transform the result of the other builder, to give a different type, say. This task is simplified by the method `mapResult` in class `Builder`. For instance, assuming a builder `bldr` of `ArrayBuffer` collections, one can turn it into a builder for `Array`s like this:

```
bldr mapResult (_.toArray)
```

Given these abstractions, the trait `TraversableLike` can define operations like `filter` in the same way for all collection classes, without compromising efficiency or precision of type signatures. First, it relies on the `newBuilder` method to create an empty builder that's appropriate for the collection at hand, then, it uses `foreach` to traverse the existing collection, appending every `elem` that meets the predicate `p` to the builder. Finally, the builder's `result` is the filtered collection.

5 Abstracting over the Collection Type Constructor

While abstracting over the representation type suffices to factor out exactly what varies in `filter`'s result type across the collection hierarchy, it cannot capture the variation in `map`'s result type. Recall that `map` is an operation that derives a collection from an existing one by applying a user-supplied function to each of its elements. For example, if the given function `f` goes from `String` to `Int`, and `xs` is a `List[String]`, `xs map f` should yield a `List[Int]`. Likewise, if `ys` is an `Array[String]`, then we expect `ys map f` to produce an `Array[Int]`.

To provide a precise abstract *declaration* of `map` at the top of the collection hierarchy – let alone a single *implementation* – we must refine the technique we developed in the previous section. To make concrete only what distinguishes the individual subclasses, we must be able to abstract over precisely what varies in these examples, and not more. Thus, we cannot simply abstract over the representation type, as the variation (of `map`'s result type) across the hierarchy is restricted to the type *constructor* that represents the collection – it does not fix the type of its elements. The element type depends on the function supplied to `map`, not on `map`'s location in the collection hierarchy. In other words, abstracting over the representation type is too coarse, since we must be able to vary the element type in the `map` method.

More concretely, we need to factor out the type constructors `List` and `Array`. Thus, instead of abstracting over the representation type, we abstract over the collection type constructor. Abstracting over type constructors requires higher-order parametric polymorphism, which we call *type constructor polymorphism* in Scala [13]. This higher-order generalisation of what is typically called "genericity" in object-oriented languages, allows to declare type parameters, such as `Coll`, that themselves take (higher-order) type parameters, such as `x` in the following snippet:

```
trait TraversableLike[+Elem, +Coll[+x]] {
  def map[NewElem](f: Elem ⇒ NewElem): Coll[NewElem]
  def filter(p: Elem ⇒ Boolean): Coll[Elem]
}
```

Now, `List[T]` may extend `TraversableLike[T, List]` in order to specify that mapping or filtering a list again yields a list, whereas the type of the elements depends on the operation. Of course, `filter`'s type can still be expressed as well.

Thus, with type constructor polymorphism, we can give a single declaration of `map` that can be specialised without redundancy in `List` and `Array`. Moreover, as discussed in Section 7, we can even provide a single implementation, where the only variation between the different concrete subclasses is how to build that concrete collection.

However, important corner cases in the collection hierarchy exhibit variations that are less uniform than the above examples. In turns out type constructor polymorphism is too uniform to express the required ad-hoc variations. The next section discusses the general case and presents the kind of polymorphism that our design hinges on.

6 Ad-hoc Polymorphism with Implicits

The examples from the previous section led us to believe that `map`'s result type is a simple "straight-line" function from the concrete collection type (e.g., `List` or `Array`) and the type of the transformed elements to the type of the resulting collection. We assumed we could simply apply the type constructor of the generic class that represents the collection to the result type of the mapped function, as mapping a function from `Int` to `String` over an `Array` of `Int`s yields an `Array[String]`.

We shall collect the variations in `map`'s type signature using a triple of types that relates the original collection, the transformed elements, and the resulting collection. Type constructor polymorphism is restricted to type functions of the shape `(CC[_], T, CC[T])`, for any type constructor[‡] `CC` and any type `T`. This section discusses several important examples that deviate from this pattern, and introduces implicits as a way of expressing them.

The regularity of transforming arrays and lists breaks down when we consider more specialised collections, such as a `BitSet`, which must nonetheless fit in our hierarchy. Consider the following interaction with the Scala REPL:

```
scala> BitSet(1,2,3) map (_ + 1)
res0:  scala.collection.immutable.BitSet = BitSet(2, 3, 4)
```

With a little bit of foresight in `Iterable`, we can capture this pattern. However, it quickly goes awry when we consider an equally desirable transformation:

```
scala> BitSet(1,2,3) map (_.toString+"!")
res1:  scala.collection.immutable.Set[java.lang.String] = Set(1!, 2!,
   3!)
```

Because the result type of `toString` is `String` and not `Int`, the result of the map cannot be a `BitSet`. Instead a general `Set[String]` is returned. One might ask why the second `map` should be admitted at all. Could one not restrict `map` on `BitSet` to mappings from `Int` to `Int`? In fact, such a restriction would be illegal because `BitSet` is declared to be a subtype of `Set[Int]` (and there are good modelling reasons why it should be). `Set[Int]` provides a `map` operation which takes arbitrary functions over `Int`, so by the Liskov substitution principle [10] every subtype of `Set[Int]` must provide the same operation.

This means that our type function for calculating `map`'s result type must now include the following triples: `(BitSet, Int, BitSet)`, and `(BitSet, String, Set[String])`, and in fact, for every type `T` different from `Int`, `(BitSet, T, Set[T])`. A type function

‡More precisely, for any type constructor `CC` with one unbounded type parameter.

that includes only the first triple `(BitSet, Int, BitSet)` can be expressed using type constructor polymorphism, but the other ones are out of reach.
Finally, consider transforming maps:

```
scala> Map("a" -> 1, "b" -> 2) map { case (x, y) ⇒ (y, x) }
res2:  scala.collection.immutable.Map[Int,java.lang.String] = Map(1 ->
    a, 2 -> b)

scala> Map("a" -> 1, "b" -> 2) map { case (x, y) ⇒ y }
res3:  scala.collection.immutable.Iterable[Int] = List(1, 2)
```

The first function swaps two arguments of a key/value pair. The result of mapping this function is again a map, but now going in the other direction. In fact, the original yields the inverse of the original map, provided it is invertible. The second function, however, maps the key/value pair to an integer, namely its value component. In that case, we cannot form a `Map` from the results, but we can still form an `Iterable`, which is the base trait of `Map`.

The irregular triples `(Map[A, B], (A, B)⇒(B, A), Map[B, A])` and — assuming `T` is not `(A, B)` — `(Map[A, B], (A, B)⇒T, Iterable[T])` summarise these type signatures, for arbitrary types `A`, `B`, and `T`.

Instead of admitting these ad-hoc type relations between the type of the collection, the transformation and the result, we could restrict `map` to recover the regularity that is supported by type constructor polymorphism. However, in doing so, we must respect the Liskov substitution principle. This requires somehow "announcing" these restrictions abstractly in the top-level type, `Iterable`. Expressing these restrictions quickly becomes unwieldy so that this is not a viable alternative.

Shoehorning the collection hierarchy into what is supported by type constructor polymorphism would lead to an imprecise interface, code duplication, and thus, in the long term, bit rot. To avoid these problems, we shall use the type system to express the required piece-wise defined type functions precisely.

Piece-wise defined type functions are reminiscent of Java's static overloading, as an individual case ("piece") of the type function corresponds to an overloaded method. However, Java's static overloading can only express fairly trivial piece-wise defined type functions, rendering it unsuitable for our purposes. Haskell's type classes [20] provide a sufficiently expressive, and principled solution. Scala introduces implicits, which, together with Scala's object-oriented constructs, support ad-hoc polymorphism in much the same way as type classes.

Implicits

The foundations of Scala's implicits are quite simple. A method's last argument list may be marked as implicit. If such an implicit argument list is omitted at a call site, the compiler will, for each missing implicit argument, search for the implicit value with the most specific type that conforms to the type of that argument. For a value to be eligible, it must have been marked with the **`implicit`** keyword, and it must be in the implicit scope at the call site. For now, the implicit scope may simply be thought of as the scope of a regular value, although it is actually broader.

```
abstract class Monoid[T] {
  def add(x: T, y: T): T
  def unit: T
}

object Monoids {
  implicit object stringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = ""
  }
  implicit object intMonoid extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
}
```

Listing 4: Using implicits to model monoids

```
def sum[T](xs: List[T])(implicit m: Monoid[T]): T =
  if(xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail))
```

Listing 5: Summing lists over arbitrary monoids

Listing 4 introduces implicits by way of a simple example. It defines an abstract class of monoids and two concrete implementations, `StringMonoid` and `IntMonoid`. The two implementations are marked with an **`implicit`** modifier. Listing 5 implements a `sum` method, which works for arbitrary monoids. `sum`'s second parameter is marked **`implicit`**. Because of that, `sum`'s recursive call does not need to pass along the `m` argument explicitly; it is instead provided automatically by the Scala compiler.

After having entered the code snippets in Listings 4 and 5 into the Scala REPL, we can bring the implicit values in the `Monoid` object into scope with **`import`** `Monoids._`. This makes the two implicit definitions of `stringMonoid` and `intMonoid` eligible to be passed as implicit arguments, so that one can write:

```
scala> sum(List("a", "bc", "def"))
res0:  java.lang.String = abcdef

scala> sum(List(1, 2, 3))
res1:  Int = 6
```

These applications of `sum` are equivalent to the following two applications, where the formerly implicit argument is now given explicitly.

```
sum(List("a", "bc", "def"))(stringMonoid)
sum(List(1, 2, 3))(intMonoid)
```

Implicits are closely related to Haskell's type classes. Where Scala uses a regular class such as `Monoid`, Haskell would use a type class. Implicit values such as `stringMonoid` and intMonoid correspond to instance declarations in Haskell. Implicit parameters correspond to contexts in Haskell. Conditional instance declarations with contexts in Haskell can be modelled in Scala by implicit functions that themselves take implicit parameters. For instance, here is a function defining an implicit lexicographical ordering relation on lists which have element types that are themselves ordered.

```
implicit def listOrdering[T](xs: List[T])(implicit elemOrd: Ordering[T]) =
  new Ordering[List[T]] {
    def compare(xs: List[T], ys: List[T]) = (xs, ys) match {
      case (Nil, Nil) ⇒ 0
      case (Nil, _) ⇒ -1
      case (_, Nil) ⇒ 1
      case (x :: xs1, y :: ys1) ⇒
        val ec = elemOrd.compare(x, y)
        if (ec != 0) ec else compare(xs1, ys1)
    }
  }
```

7 Implicits for Scala's collections

The most interesting application of implicits in our design of Scala's collections library is in the typing of methods like map, which require expressive ad-hoc polymorphism. We have seen that the result type of BitSet's map method can be specified in terms of triples that relate the source collection, the target element type, and the resulting collection: (BitSet, Int, BitSet), (BitSet, T, Set[T]). These triples define a piece-wise function on types, encoded as the implicit instances of the trait CanBuildFrom in Listing 6.

The listing first defines the trait CanBuildFrom, which takes three type parameters: the Collection type parameter indicates the collection from which the new collection should be built, the NewElem type parameter indicates the new element type of the collection to be built, and the Result type parameter indicates the type of that collection itself. The trait has a single deferred method, apply, which produces a Builder object that constructs a Result collection from NewElem elements.

The listing then shows the map method in class TraversableLike. This method is defined for every function result type B and every collection type To such that there exists an implicit value of CanBuildFrom[Repr, B, To], where Repr is the representation type of the current collection. In other words, the triple (Repr, B, To) must be populated by a CanBuildFrom value. We'll come back to the implementation of map later in this section.

Two such CanBuildFrom values are shown in the companion objects — the objects that are co-defined with the classes of the same name – of classes Set and BitSet. Scala's scope rules for implicits include the companion object of a type in the implicit scope for that type. More precisely, when searching for an implicit value of type T, we consider all types S that form part of T, as well as all the supertypes of any such part S. The companion objects of all these types may contain implicit definitions which are then in the implicit scope for T. For instance, when searching for an implicit value of type CanBuildfrom[BitSet, Int, ?To], the BitSet object is in the implicit scope because BitSet forms part of the type of the requested implicit. The Set object is also in the implicit scope because Set is a superclass of

```scala
trait CanBuildFrom[-Collection, -NewElem, +Result] {
  def apply(from: Collection): Builder[NewElem, Result]
}

trait TraversableLike[+A, +Repr] {
  def repr: Repr = ...
  def foreach[U](f: A ⇒ U): Unit = ...
  def map[B, To](f: A⇒B)(implicit cbf: CanBuildFrom[Repr, B, To]): To = {
    val b = cbf(repr) // get the builder from the CanBuildFrom instance
    for (x <- this) b += f(x) // transform element and add
    b.result
  }
}
trait SetLike[+A, +Repr] extends TraversableLike[A, Repr] {  }
trait BitSetLike[+This <: BitSetLike[This] with Set[Int]] extends SetLike[
    Int, This] {}

trait Traversable[+A] extends TraversableLike[A, Traversable[A]]
trait Set[+A] extends Traversable[A] with SetLike[A, Set[A]]
class BitSet extends Set[Int] with BitSetLike[BitSet]

object Set {
  implicit def canBuildFromSet[B] = new CanBuildFrom[Set[_], B, Set[B]] {
    def apply(from: Set[_]) = ...
  }
}

object BitSet {
  implicit val canBuildFromBitSet = new CanBuildFrom[BitSet, Int, BitSet] {
    def apply(from: BitSet) = ...
  }
}

object Test {
  val bits = BitSet(1, 31, 15)
  val shifted = bits map (x ⇒ x + 1)
  val strings = bits map (x ⇒ x.toString)
}
```

Listing 6: Encoding the CanBuildFrom type-relation for BitSet

BitSet.

Consider now the Test object in Listing 6. It contains two applications of map on the BitSet value bits. In the first case, the implicit parameter of the map method has a type of the form CanBuildfrom[BitSet, Int, ?To] because the collection on which the map is performed is a BitSet and the result type of the new collection is Int. Both shown CanBuildFrom values are in the implicit scope, and both match the type pattern that is searched. In this case, the canBuildFromBitSet value in object BitSet is the more specific of the two, and will be selected.

Implicit resolution uses Scala's standard member resolution rules for overloading in order to disambiguate between several applicable implicits, such as canBuildFromSet and canBuildFromBitSet in the example above. Member resolution orders equivalent members according to where they are defined in the subclassing hierarchy, with definitions in class A preceding over those in class B if A is a subclass of B. This ordering is extended to companion objects, which can be seen as forming a parallel hierarchy to the corresponding class hierarchy, somewhat like meta-classes in Smalltalk.

In the second case, the implicit parameter of the map method has a type of the form CanBuildfrom[BitSet, String, ?To] because the result type of the second function argument is String. In this case, only the implicit value in Set is applicable and will be selected.

Type inference takes the availability of an implicit value into account. Thus, when inferring the type arguments for map, the To type parameter is constrained by the search for the applicable implicit. In the example, shifted gets type BitSet since the implicit value canBuildFromBitSet is selected, and for that to be a valid argument for map's cbf implicit type parameter, its To type parameter must be BitSet. The definition of strings, on the other hand, passes canBuildFromBitSet to the map application, with Set[String] as third type pararameter. Consequently, Set[String] is also the result type of that application.

The second application of map, stored in strings, explains why CanBuildFrom's first type parameter is contravariant[§]: an implicit of CanBuildFrom[BitSet, String, ?To] is required, where ?To is a type inference variable. We have that BitSet is a subtype of Set. By contravariance of CanBuildFrom, this means that CanBuildFrom[Set, String, ?To] is a subtype of CanBuildFrom[BitSet, String, ?To]. Hence, CanBuildFrom[Set, String , ?To] can be substituted for the required CanBuildFrom[BitSet, String, ?To], and ?To is inferred to be Set[String].

Finding Builders at Run Time

We have seen that map can be given a precise type signature in TraversableLike, but how do we implement it? Since map has a value of type CanBuildFrom[From, Elem, To], the idea is to let the implicit canBuildFrom values produce builder objects of type Builder[Elem, To] that construct collections of the right kind.

However, there is one minor snag. Since implicit resolution is performed at compile time, it cannot take dynamic types into account. Nonetheless, we expect a List to be created

[§]The variance of the other type parameters will become apparent in the next section.

when the dynamic type is List, even if the static type information is limited to Iterable. This is illustrated by the following interaction with the Scala REPL:

```
scala> val xs: Iterable[Int] = List(1, 2, 3)
xs:  Iterable[Int] = List(1, 2, 3)

scala> xs map (x ⇒ x * x)
res0:  Iterable[Int] = List(1, 4, 9)
```

If CanBuildFrom solely relied on the triple of types (Iterable[Int], Int, Iterable[Int]) to provide a builder, it could not do better than to statically select a Builder[Int, Iterable[Int]], which in turn could not build a List. Thus, we add a run-time indirection that makes this selection more dynamic.

The idea is to give the apply method of CanBuildfrom access to the dynamic type of the original collection via its from argument. An instance cbf of CanBuildFrom[Iterable[Int], Int, Iterable[Int]], is essentially a function from an Iterable[Int] to a Builder[Int, Iterable[Int]], which constructs a builder that is appropriate for the dynamic type of its argument. This is shortly explained in more detail. We first discuss how map is implemented in terms of this abstraction.

The implementation of map in Listing 6 is quite similar to the implementation of filter shown in Listing 2. The interesting difference lies in how the builder is acquired: whereas filter called the newBuilder method of class TraversableLike, map uses the instance of CanBuildFrom that is passed in as a witness to the constraint that a collection of type To with elements of type B can be derived from a collection with type Repr. This nicely brings together the static and the dynamic aspects of implicits: they express rich relations on types, which may be witnessed by a run-time entity. Thus, static implicit resolution resolves the constraints on the types of map, and virtual dispatch picks the best dynamic type that corresponds to these constraints.

Most instances of CanBuildFrom use the same structure for this virtual dispatch, so that we can implement it in GenericTraversableTemplate, the higher-kinded implementation trait for all traversables, as shown in Listing 7.

Let's see what happens for a concrete call xs.map(f), where f has static type A ⇒ B, and xs's static type is a subtype of GenericTraversableTemplate[A, CC]. The compiler will statically select an instance of CanBuildFrom[CC[A], B, CC[B]] for the implicit argument cbf. The call cbf(**this**) in map will actually be cbf(xs), which, assuming cbf was a standard instance of GenericCanBuildFrom[B], evaluates to xs.genericBuilder[B], and finally xs.companion.newBuilder[B]. Thus, whatever the dynamic type of xs, it must simply implement companion to point to its factory companion object, and implement the newBuilder method there.

8 Scala 2.8 Collections Hierarchy

In this section we give an architectural summary of the 2.8 collections framework and discuss how it can be extended by implementers of new collection classes.

Figure 1 gives an overview of some common collection classes. Classes that were added in the 2.8 framework are shaded in that figure. At the top of the collection hierarchy is now

```
trait GenericCompanion[+CC[X] <: Traversable[X]] {
  def newBuilder[A]: Builder[A, CC[A]]
}

trait GenericTraversableTemplate[+A, +CC[X] <: Traversable[X]] {
  // The factory companion object that builds instances of class CC.
  def companion: GenericCompanion[CC]

  // The builder that builds instances of CC at arbitrary element types.
  def genericBuilder[B]: Builder[B, CC[B]] = companion.newBuilder[B]
}


trait TraversableFactory[CC[X] <: Traversable[X] with
                                  GenericTraversableTemplate[X, CC]]
                                       extends GenericCompanion[CC] {
  // Standard CanBuildFrom instance for a CC that's a traversable.
  class GenericCanBuildFrom[A] extends CanBuildFrom[CC[_], A, CC[A]] {
    def apply(from: CC[_]) = from.genericBuilder[A]
  }
}
```

Listing 7: `GenericCanBuildFrom`

class `Traversable`, which implements all accesses to its data via its `foreach` method. Class `Traversable` is extended by class `Iterable`, which implements all traversals by means of an iterator. `Iterable` is further extended by classes `Seq`, `Set`, and `Map`. Each of these classes has further subclasses that capture some particular trait of a collection. For instances, sequences `Seq` are split in turn into `LinearSeq` for linear access sequences such as lists and `IndexedSeq` for random access sequences such as arrays.

All collection classes are kept in a package `scala.collection`. This package has three subpackages: `mutable`, `immutable`, and `generic`. Most collections exist in three forms, depending on their mutability.

A collection in package `scala.collection.immutable` is guaranteed to be immutable for everyone. That means one can rely on the fact that accessing the same collection value over time will always yield a collection with the same elements.

A collection in package `scala.collection.mutable` is known to have some operations that change the collection in place.

A collection in package `scala.collection` can be either mutable or immutable. For instance, `collection.Seq[T]` is a superclass of both `collection.immutable.Seq[T]` and `collection.mutable.Seq[T]`. Generally, the root collections in package `scala.collection` define the same interface as the immutable collections, and the mutable collections in package `scala.collection.mutable` typically add some destructive modification operations to this immutable interface. The difference between root collections and immutable collections is that a user of an immutable collection has a guarantee that nobody can mutate the collection, whereas users of root collections have to assume modifications by

```
package mycollection
import collection.generic.{CanBuildFrom, GenericTraversableTemplate,
    GenericCompanion, SeqFactory}
import collection.mutable.{Builder, ArrayBuffer}

class Vector[+A](buf: ArrayBuffer[A])
  extends collection.immutable.IndexedSeq[A]
     with collection.IndexedSeqLike[A, Vector[A]]
     with GenericTraversableTemplate[A, Vector] {
  override def companion: GenericCompanion[Vector]  = Vector
  def length = buf.length
  def apply(idx: Int) = buf.apply(idx)
}

object Vector extends SeqFactory[Vector] {
  implicit def canBuildFrom[A]: CanBuildFrom[Vector[_], A, Vector[A]] =
    new GenericCanBuildFrom[A]
  def newBuilder[A]: Builder[A, Vector[A]] =
    new ArrayBuffer[A] mapResult (buf ⇒ new Vector(buf))
}
```

Listing 8: A sample collection implementation.

others, even though they cannot do any modifications themselves.

The `generic` package contains building blocks for implementing various collections. Typically, collection classes defer the implementations of some of their operations to classes in `generic`. Users of the collection framework, on the other hand, should need to refer at classes in `generic` only in exceptional circumstances.

Integrating new collections

The collection framework is designed to make it easy to add new kinds of collections to it. As an example, Listing 8 shows a simple yet complete implementation of immutable vectors.

The `Vector` trait inherits from three other traits. It inherits from `scala.collection.immutable.IndexedSeq` to specify that `Vector` is a subtype of a random access sequence and is immutable. It inherits most of implementations of its methods from the `IndexedSeqLike` trait, specialising the representation type to `Vector[A]`. Finally, `Vector` mixes in `GenericTraversableTemplate`, and instantiates the type parameter that abstracts over the collection type constructor to `Vector`.

Only three abstract methods remain to be implemented. Two of these, `length` and `apply`, are related to querying an existing sequence, while the third, `companion`, is involved in creating new sequences. The method `length` yields the length of the sequence, and `apply` returns an element at a given index. These two operations are implemented in trait `Vector`. For simplicity's sake they simply forward to the same operation of an underlying `ArrayBuffer`. Of course, the actual implementation of immutable vectors is considerably more refined algorithmically, and more efficient.

The third method, `companion`, is declared in `GenericTraversableTemplate`. `Vector` defines it to refer to its companion object, which specifies the `CanBuildFrom` case for `Vector`. This ensures that calling `map` on a `Vector` yields a `Vector`. As discussed in Section 7, the implicit value that populates the `CanBuildFrom` relation on types is an instance of `GenericCanBuildFrom`, which delegates the creation of the `Vector`-specific builder to the `newBuilder` method. This method creates an array buffer (which is a specialised kind of builder), and transforms results coming out of this buffer into instances of `Vector`. That's the minimal functionality required for instances of `GenericTraversableTemplate`.

As an added convenience, the `Vector` object inherits from class `SeqFactory` which makes available a large set of creation methods for vectors.

With the setup as described in Listing 8 the `Vector` class is fully integrated into the collections hierarchy. It inherits all methods defined on indexed sequences and all construction methods for such sequences can be applied to it. The following REPL script shows some of the operations that are supported. First, here are some ways to construct vectors:

```
import mycollection.Vector

scala> val v = Vector(1, 2, 3)
v:  mycollection.Vector[Int] = Vector(1, 2, 3)

scala> val ev = Vector.empty
ev:  mycollection.Vector[Nothing] = Vector()

scala> val zeroes = Vector.fill(10)(0)
zeroes:  mycollection.Vector[Int] = Vector(0, 0, 0, 0, 0, 0, 0, 0, 0,
   0)

scala> val squares = Vector.tabulate(10)(x ⇒ x * x)
squares:  mycollection.Vector[Int] = Vector(0, 1, 4, 9, 16, 25, 36,
   49, 64, 81)

scala> val names = Vector("Jane", "Bob", "Pierre")
names:  mycollection.Vector[java.lang.String] = Vector(Jane, Bob,
   Pierre)

scala> val ages = Vector(21, 16, 24)
ages:  mycollection.Vector[Int] = Vector(21, 16, 24)
```

To continue, here are some operations on vectors.

```
scala> val persons = names zip ages
persons:  mycollection.Vector[(java.lang.String, Int)] =
   Vector((Jane,21), (Bob,16), (Pierre,24))

scala> val (minors, adults) = persons partition (_._2 <= 18)
minors:  mycollection.Vector[(java.lang.String, Int)] =
   Vector((Bob,16))
adults:  mycollection.Vector[(java.lang.String, Int)] =
   Vector((Jane,21), (Pierre,24))
```

```
scala> val adultNames = adults map (_._2)
adultNames: mycollection.Vector[Int] = Vector(21, 24)

scala> val totalAge = ages reduceLeft (_ + _)
totalAge: Int = 61
```

To summarise: To fully integrate a new collection class into the framework one needs to pay attention to the following points:

1. Decide whether the collection should be mutable or immutable.
2. Pick the right base classes for the collection.
3. Inherit from the right template trait to implement most collection operations.
4. If one wants `map` and similar operations return instances of the collection type, provide an implicit builder factory in the companion object.
5. If the collection should have dynamic type adaptation for `map` and operations like it, one should also inherit from `GenericTraversableTemplate`, or implement equivalent functionality.

A simpler scheme is also possible if one does not need bulk operations like `map` or `filter` to return the same collection type. In that case one can simply inherit from some general collection class like `Seq` or `Map` and implement any additional operations directly.

9 Dealing with Arrays and Strings

The integration of arrays into the Scala collection library has turned out to be very challenging. This has mostly to do with the clash between requirements and the constraints imposed by Java and the JVM. On the one hand, arrays play an important role for interoperation with Java, which means that they need to have the same representation as in Java. This low-level representation is also useful to get high performance out of arrays. But on the other hand, arrays in Java are severely limited.

First, there's actually not a single array type representation in Java but nine different ones: one representation for arrays of reference type and another eight for arrays of each of the primitive types `byte`, `char`, `short`, `int`, `long`, `float`, `double`, and `boolean`. Unfortunately, `java.lang.Object` is the most specific common type for these different representations, even though there are some reflective methods to deal with arrays of arbitrary type in `java.lang.reflect.Array`. Second, there's no way to create an array of a generic type; only monomorphic array creations are allowed. Third, arrays only support operations for indexing, updating, and getting their length.

Contrast this with what we would like to have in Scala: Arrays should slot into the collections hierarchy, supporting the roughly one hundred methods that are defined on sequences. And they should certainly be generic, so that one can create an `Array[T]` where `T` is a type variable.

The previous collection design dealt with arrays in an ad-hoc way. The Scala compiler wrapped and unwrapped arrays when required in a process called boxing and unboxing, similarly to what is done to treat primitive numeric types as objects. Additional "magic" made generic array creation work. An expression like **new** `Array[T]` where `T` is a type parameter was converted to **new** `BoxedAnyArray[T]`. `BoxedAnyArray` was a special wrapper class which *changed its representation* depending on the type of the concrete Java array to which it was cast. This scheme worked well enough for most programs but the implementation "leaked" for certain combinations of type tests and type casts, as well as for observing uninitialised arrays. It also could lead to unexpectedly low performance. Some of the problems have been described by David MacIver [11] and Matt Malone [12]. Moreover, boxed arrays were unsound when combined with covariant collections. In summary, the old array implementation technique was problematic because it was a leaky abstraction that was complicated enough so that it would be very tedious to specify where the leaks were to be expected.

The obvious way to reduce the amount of "magic" needed for arrays is to have two representations: one that corresponds closely to a Java array and another that forms an integral part of Scala's collection hierarchy. Implicit conversions can be used to transparently convert between the two representations. A possible downside of having two array types would be that it forces programmers to choose the kind of array to work with. That choice would not be clear-cut: the Java-like arrays would be fast and interoperable whereas the Scala native arrays would support a much nicer set of operations on them. With a choice like this, one would expect different components and libraries to make different decisions, which would result in incompatibilities and brittle, complex code. In a word, an ideal environment for future bit rot.

Fortunately, the introduction of implementation traits in 2.8 collections offers a way out of that dilemma of choice. Arrays can be integrated into this framework using *two* implicit conversions. The first conversion maps an `Array[T]` to an object of type `ArrayOps`, which is a subtype of type `IndexedSeqLike[T, Array[T]]`. Using this conversion, all sequence operations are available for arrays at the natural types. In particular, methods will always yield arrays instead of `ArrayOps` values as their results. Because the results of these implicit conversions are so short-lived, modern VM's can eliminate them altogether using escape analysis, so we expect the calling overhead for these added methods to be essentially zero.

So far so good. But what if we need to convert an array to a real `Seq`, not just call a `Seq` method on it? This is handled by another implicit conversion, which takes an array and converts it into a `WrappedArray`. `WrappedArray`s are mutable, indexed sequences that implement all sequence operations in terms of a given Java array. The difference between a `WrappedArray` and an `ArrayOps` object is apparent in the type of methods like `reverse`: Invoked on a `WrappedArray`, `reverse` again returns a `WrappedArray`, but invoked on an `ArrayOps` object, it returns an `Array`. The conversion from `Array` to `WrappedArray` is invertible. A dual implicit conversion goes from `WrappedArray` to `Array`. `WrappedArray` and `ArrayOps` both inherit from an implementation trait `ArrayLike`. This is to avoid duplication of code between `ArrayOps` and `WrappedArray`; all operations are factored out into the common `ArrayLike` trait.

Avoiding ambiguities. The two implicit conversions from `Array` to `ArrayLike` values are disambiguated according to the rules explained in Section 7. Applied to arrays, this means that we can prioritise the conversion from `Array` to `ArrayOps` over the conversion from `Array` to `WrappedArray` by placing the former in the standard `Predef` object (which is visible in all user code) and by placing the latter in a class `LowPriorityImplicits`, which is inherited by `Predef`. This way, calling a sequence method will always invoke the conversion to `ArrayOps`. The conversion to `WrappedArray` will only be invoked when an array needs to be converted to a sequence.

Integrating Strings. Strings pose similar problems as arrays in that we are forced to pick an existing representation which is not integrated into the collection library and which cannot be extended with new methods because Java's `String` class is **`final`**. The solution for strings is very similar as the one for arrays. There are two prioritised implicit conversions that apply to strings. The low-priority conversion maps a string to an immutable indexed sequence of type `scala.collection.immutable.IndexedSeq`. The high-priority conversion maps a string to a (short-lived) `StringOps` object which implements all operations of an immutable indexed sequence, but with `String` as the result type. The previous collection framework implemented only the first conversion. This had the following undesirable effect:

```
"abc" != "abc".reverse.reverse
```

This unintuitive behaviour occurred because the result of the double `reverse` in previous Scala collections was a `Seq` instead of a `String`, so Java's built-in operation of equality an strings failed to recognise it as equal to the string. In the new collection framework, the high-priority conversion to `StringOps` will be applied instead, so that `"abc".reverse.reverse` yields a `String` and the equality holds.

Generic Array Creation and Manifests. The only remaining question is how to implement generic array creation. Unlike Java, Scala allows an instance creation **`new`** `Array[T]` where `T` is a type parameter. How can this be implemented, given the fact that there does not exist a uniform array representation in Java? The only way to do this is to require additional run-time information which describes the type `T`. Scala 2.8 has a new mechanism for this, which is called a `Manifest`. An object of type `Manifest[T]` provides complete information about the type `T`. Manifest values are typically passed in implicit parameters, and the compiler knows how to construct them for statically known types `T`. There exists also a weaker form named `ClassManifest` which can be constructed from knowing just the top-level class of a type, without necessarily knowing all its argument types. It is this type of runtime information that's required for array creation.

10 Conclusion

As this paper is written we are about to release Scala 2.8 with its new collections library. So it is too early to tell whether the new design withstands bit rot better than the old one did. Nevertheless, we have reasonable grounds for hoping that this will be the case.

The new collection design is far more regular than the old one and makes many aspects of its structure more explicit. Mutability aspects are consistently expressed by placing collections in the right package. Reusable method implementations are separated from client interfaces in implementation classes. This allowed us to have simple and intuitive types like `Seq[String]` or `Map[String, Int]` for clients yet have implementation classes expose their representation as in additional type parameter that can be instantiated as needed by implementers. There is a common universal framework of builders and traversal methods. Code duplication is almost completely absent (There is still a certain amount of duplicated boilerplate code in the definition of so called *views*, which are by-name transforms of existing collections, but these views are typically not extended by third parties). Arrays and strings are cleanly integrated into the collections framework with implicit conversions instead of requiring special compiler support.

Getting this design right was very hard, however. It took us about a year to go from a first sketch to the final implementation. In doing this work, we also encountered some dead ends. Initially, we anticipated that most of the flexibility and opportunities for code-reuse of the framework would come from higher-kinded types. In retrospect this turned out to be a false assumption, because requirements on the element type of collections varied from collection to collection. So common methods on collections had to be defined piece-wise. They would return a specialised collection for some element types, and a more general "fall-back" collection for other element types. In the course of the project, we learned how to use implicits to define these piece-wise functions. More generally, we came to appreciate how implicits can encode rich user-defined type theories. So, in the end higher-kinded types played a smaller role than anticipated and implicits played a much larger role.

Nevertheless, type constructor polymorphism did find a useful application niche in the collections framework, where it came to generate factories for collection classes. This application worked out fine because setting up a factory by inheriting from a factory class which takes higher-kinded type parameters is done on a case-by-case basis. Collections which pose additional constraints on the higher-kinded type parameter can simply choose not to inherit from `TraversableFactory` and implement the required methods themselves. By contrast, implementation classes follow a subtyping hierarchy; any specification made higher up in the hierarchy needs to hold up for all inheriting classes. So the lesson drawn is not that higher-kinded types per se are of limited utility, but that they sometimes interact in awkward ways with a rich subtyping hierarchy. In some sense this is a new facet of the fragile baseclass problem.

Acknowledgments The final architecture and implementation of collections was mainly done by Odersky but several people have shaped the design in important ways. The new libraries would not exist without their contributions. Matthias Zenger wrote Scala's original collection libraries for sets, maps, buffers, and other types. Many of his design decisions have survived the redesign. Some have been generalised, such as his partition into mutable and immutable collection packages, which now applies uniformly for all kinds of collections including sequences. Sean McDirmid added projections to the original collection libraries, a concept which has been taken up in the redesign under the name of views. Adriaan Moors developed higher-kinded types in Scala, which gave the primary motivation for the collection redesign, even though in the end their role is more narrow than originally anticipated. Adriaan was also the first to explore builders as a fundamental abstraction for Scala's collections. David McIver proposed builders as implicit parameters and `Traversable` as a generalisation of `Iterable`. Miles Sabin contributed the bidirectional wrappers that convert between Java collections and Scala collections. Phil Bagwell, Gilles Dubochet, Burak Emir, Erik Engbrecht, Stepan Koltsov, Stéphane Micheloud, Tony Morris, Jorge Ortiz, Paul Phillips, David Pollak, Tiark Rompf, Lex Spoon, and many others have contributed to specific collection classes or made important suggestions for improvements.

Bibliography

[1] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection classes. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA*, pages 47–64. ACM, 2003.

[2] Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. The semantics of second-order lambda calculus. *Inf. Comput.*, 85(1):76–134, 1990.

[3] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *OOPSLA*, pages 1–15, 1992.

[4] Vincent Cremet and Philippe Altherr. Adding type constructor parameterization to Java. *Journal of Object Technology*, 7(5):25–65, June 2008. Special Issue: Workshop on FTfJP, ECOOP 07. http://www.jot.fm/issues/issue_2008_06/article2/.

[5] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 63–70. ACM, 2007.

[6] J.Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These d'Etat, Paris VII, 1972.

[7] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[8] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[9] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *J. Funct. Program.*, 5(1):1–35, 1995.

[10] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM.

[11] David MacIver. Scala arrays, 2008. Blog post at `http://www.drmaciver.com/2008/06/scala-arrays`.

[12] Matt Malone. The mystery of the parameterized array, 2009. Blog post at `http://oldfashionedsoftware.com/2009/08/05/the-mystery-of-the-parameterized-array`.

[13] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In Gail E. Harris, editor, *OOPSLA*, pages 423–438. ACM, 2008.

[14] Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., 2006.

[15] Martin Odersky. Pimp my library, 2006. Blog post at `http://www.artima.com/weblogs/viewpost.jsp?thread=179766`.

[16] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[17] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[18] John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[19] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP*, pages 248–274, 2003.

[20] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

[21] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI : Generalized interfaces for Java. In Erik Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 347–372. Springer, 2007.