

Future-Proofing Collections

from mutable to persistent to parallel

Martin Odersky
Chief Architect, Typesafe



Brief History

- 2003 Scala 1.0 ships with first collection library: Some functional types, such as `List`. No common organization.
- 2005 Redesigned & bootstrapped Scala 2.0 comes with generic imperative + functional collection framework. Traditional design requires large amount of code duplication.
- 2005-2009 Library evolves, several cooks, more duplication,
→ bit rot
- 2009 Scala 2.8 collections released. Radical reduction of code duplication through advanced architecture & types.

The Scala Way of Collections

- De-emphasize destructive updates
- Focus on *transformers* that map collections to collections
- Have a complete range of *persistent* collections

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

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

```
scala> xs map (_ + 1)
res0: Seq[Int] = List(2, 3, 4)
```

```
scala> ys map (_ + 1)
res1: List[Int] = List(2, 3, 4)
```

Collection Properties

- object-oriented
- generic: `List[T]`, `Map[K, V]`
- optionally persistent, e.g. `collection.immutable.Seq`
- higher-order, with methods such as `foreach`, `map`, `filter`.
- **Uniform return type principle:** Operations return collections of the same type (constructor) as their left operand, as long as this makes sense.

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

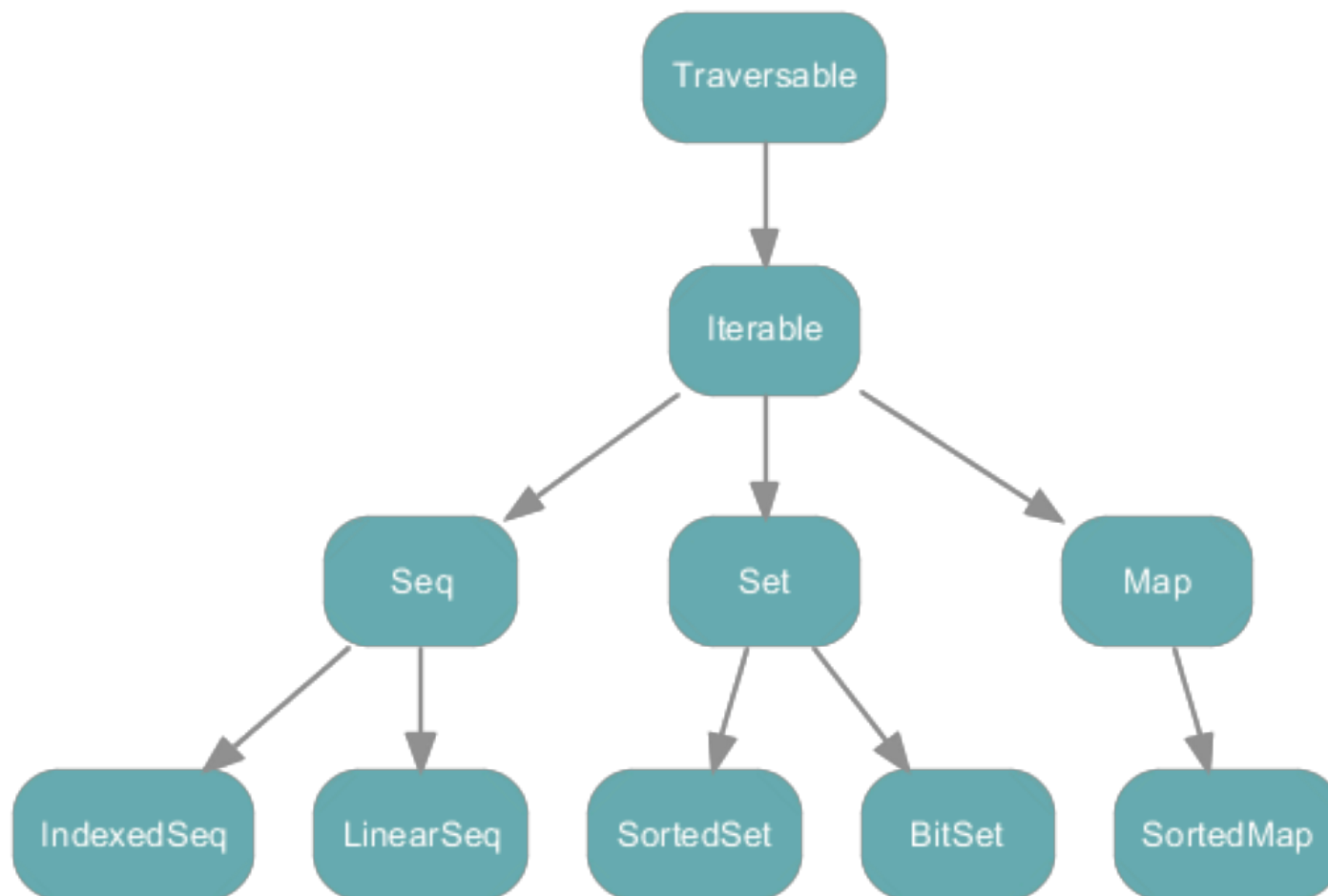
```
scala> val xs: Seq[Int] = ys
xs: Seq[Int] = List(1, 2, 3)
```

```
scala> xs map (_ + 1)
res0: Seq[Int] = List(2, 3, 4)
```

```
scala> ys map (_ + 1)
res1: List[Int] = List(2, 3, 4)
```

This makes a very elegant and powerful combination.

Some General Scala Collections



Constructed automatically using *decodify*.

Using Collections: Map and filter

```
scala> val xs = List(1, 2, 3)
```

```
xs: List[Int] = List(1, 2, 3)
```

```
scala> val ys = xs map (x => x + 1)
```

```
ys: List[Int] = List(2, 3, 4)
```

```
scala> val ys = xs map (_ + 1)
```

```
ys: List[Int] = List(2, 3, 4)
```

```
scala> val zs = ys filter (_ % 2 == 0)
```

```
zs: List[Int] = List(2, 4)
```

```
scala> val as = ys map (0 to _)
```

```
as: List(Range(0, 1, 2), Range(0, 1, 2, 3), Range(0, 1, 2, 3, 4))
```

Using Collections: flatMap and groupBy

```
scala> val bs = as.flatten
bs: List[Int] = List(0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 4)

scala> val bs = ys flatMap (0 to _)
bs: List[Int] = List(0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 4)

scala> val fruit = Vector("apple", "pear", "pineapple")
fruit: Vector[String] = Vector(apple, pear, pineapple)

scala> fruit groupBy (_.head)
res11: scala.collection.immutable.Map[Char,Vector[String]] =
  Map(a -> Vector(apple), p -> Vector(pear, pineapple))
```

Using Collections: For Notation

```
scala> for (x <- xs) yield x + 1           // same as map
res14: List[Int] = List(2, 3, 4)

scala> for (x <- res14 if x % 2 == 0) yield x // ~ filter
res15: List[Int] = List(2, 4)

scala> for (x <- xs; y <- 0 to x) yield y   // same as flatMap
res17: List[Int] = List(0, 1, 0, 1, 2, 0, 1, 2, 3)
```

Using Maps

```
scala> val m = Map(1 -> "ABC", 2 -> "DEF", 3 -> "GHI")  
m: Map[Int, String] = Map(1 -> ABC, 2 -> DEF, 3 -> GHI)
```

```
scala> m(2)  
res0: String = DEF
```

```
scala> m + (4 -> "JKL")  
res1: Map[Int, String] = Map(1 -> ABC, 2 -> DEF, 3 -> GHI, 4 -> JKL)
```

```
scala> m map { case (k, v) => (v, k) }  
res8: Map[String, Int] = Map(ABC -> 1, DEF -> 2, GHI -> 3)
```

An Example

- Task: Phone keys have mnemonics assigned to them.

```
val mnemonics = Map(  
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",  
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")
```

- Assume you are given a dictionary `dict` as a list of words. Design a class `Coder` with a method `translate` such that

```
new Coder(dict).translate(phoneNumber)
```

produces all phrases of words in `dict` that can serve as mnemonics for the phone number.

- Example: The phone number “7225276257” should have the mnemonic
 Scala rocks
as one element of the list of solution phrases.

Program Example: Phone Mnemonics

- This example was taken from:
Lutz Prechelt: An Empirical Comparison of Seven Programming Languages. [IEEE Computer 33](#)(10): 23-29 (2000)
- Tested with Tcl, Python, Perl, Rexx, Java, C++, C
- Code size medians:
 - 100 loc for scripting languages
 - 200-300 loc for the others

Outline of Class Coder

```
class Coder(words: List[String]) {  
  
  private val mnemonics = Map(  
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",  
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")  
  
  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */  
  private val charCode: Map[Char, Char] = ??  
  
  /** Maps a word to the digit string it can represent, e.g. "Java" -> "5282" */  
  private def wordCode(word: String): String = ??  
  
  /** A map from digit strings to the words that represent them,  
   * e.g. "5282" -> Set("Java", "Kata", "Lava", ...) */  
  private val wordsForNum: Map[String, Set[String]] = ??  
  
  /** Return all ways to encode a number as a list of words */  
  def encode(number: String): Set[List[String]] = ??  
  
  /** Maps a number to a list of all word phrases that can represent it */  
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")  
}
```

Class Coder (1)

```
class Coder(words: List[String]) {  
  
  private val mnemonics = Map(  
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",  
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")  
  
  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */  
  private val charCode: Map[Char, Char] =  
  
  /** Maps a word to the digit string it can represent */  
  private def wordCode(word: String): String = ??  
  
  /** A map from digit strings to the words that represent them */  
  private val wordsForNum: Map[String, List[String]] = ??  
  
  /** Return all ways to encode a number as a list of words */  
  def encode(number: String): Set[List[String]] = ??  
  
  /** Maps a number to a list of all word phrases that can represent it */  
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")  
}
```

Class Coder (1)

```
class Coder(words: List[String]) {  
  
  private val mnemonics = Map(  
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",  
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")  
  
  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */  
  private val charCode: Map[Char, Char] =  
    for ((digit, str) <- mnemonics; ltr <- str) yield (ltr -> digit)  
  
  /** Maps a word to the digit string it can represent */  
  private def wordCode(word: String): String = ??  
  
  /** A map from digit strings to the words that represent them */  
  private val wordsForNum: Map[String, List[String]] = ??  
  
  /** Return all ways to encode a number as a list of words */  
  def encode(number: String): Set[List[String]] = ??  
  
  /** Maps a number to a list of all word phrases that can represent it */  
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")  
}
```

Class Coder (2)

```
class Coder(words: List[String]) {  
  
  private val mnemonics = Map(  
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",  
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")  
  
  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */  
  private val charCode: Map[Char, Char] =  
    for ((digit, str) <- m; letter <- str) yield (letter -> digit)  
  
  /** Maps a word to the digit string it can represent, e.g. "Java" -> "5282" */  
  private def wordCode(word: String): String =  
  
  
  /** A map from digit strings to the words that represent them */  
  private val wordsForNum: Map[String, Set[String]] = ??  
  
  /** Return all ways to encode a number as a list of words */  
  def encode(number: String): Set[List[String]] = ??  
  
  /** Maps a number to a list of all word phrases that can represent it */  
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")  
}
```

Class Coder (2)

```
class Coder(words: List[String]) {  
  
  private val mnemonics = Map(  
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",  
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")  
  
  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */  
  private val charCode: Map[Char, Char] =  
    for ((digit, str) <- m; letter <- str) yield (letter -> digit)  
  
  /** Maps a word to the digit string it can represent, e.g. "Java" -> "5282" */  
  private def wordCode(word: String): String =  
    word.toUpperCase map charCode  
  
  /** A map from digit strings to the words that represent them */  
  private val wordsForNum: Map[String, Set[String]] = ??  
  
  /** Return all ways to encode a number as a list of words */  
  def encode(number: String): Set[List[String]] = ??  
  
  /** Maps a number to a list of all word phrases that can represent it */  
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")  
}
```

Class Coder (3)

```
class Coder(words: List[String]) {  
  
  private val mnemonics = Map(  
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",  
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")  
  
  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */  
  private val charCode: Map[Char, Char] =  
    for ((digit, str) <- m; letter <- str) yield (letter -> digit)  
  
  /** Maps a word to the digit string it can represent */  
  private def wordCode(word: String): String =  
    word.toUpperCase map charCode  
  
  /** A map from digit strings to the words that represent them,  
   * e.g. "5282" -> Set("Java", "Kata", "Lava", ...) */  
  private val wordsForNum: Map[String, List[String]] =  
  
  
  /** Return all ways to encode a number as a list of words */  
  def encode(number: String): Set[List[String]] = ??  
  
  /** Maps a number to a list of all word phrases that can represent it */  
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")  
}
```

Class Coder (3)

```
class Coder(words: List[String]) {  
  
  private val mnemonics = Map(  
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",  
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")  
  
  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */  
  private val charCode: Map[Char, Char] =  
    for ((digit, str) <- m; letter <- str) yield (letter -> digit)  
  
  /** Maps a word to the digit string it can represent */  
  private def wordCode(word: String): String =  
    word.toUpperCase map charCode  
  
  /** A map from digit strings to the words that represent them,  
   * e.g. "5282" -> Set("Java", "Kata", "Lava", ...) */  
  private val wordsForNum: Map[String, List[String]] =  
    (words groupBy wordCode) withDefaultValue List()  
  
  /** Return all ways to encode a number as a list of words */  
  def encode(number: String): Set[List[String]] = ??  
  
  /** Maps a number to a list of all word phrases that can represent it */  
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")  
}
```


Class Coder (4)

```
class Coder(words: List[String]) { ...

  /** Return all ways to encode a number as a list of words */
  def encode(number: String): Set[List[String]] =
    if (number.isEmpty)
      Set(List())
    else {
      for {
        splitPoint <- 1 to number.length

        /** Maps a number to a list of all word phrases that can represent it */
        def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
      }
    }
}
```

Class Coder (4)

```
class Coder(words: List[String]) { ...

  /** Return all ways to encode a number as a list of words */
  def encode(number: String): Set[List[String]] =
    if (number.isEmpty)
      Set(List())
    else {
      for {
        splitPoint <- 1 to number.length
        word <- wordsForNum(number take splitPoint)

        /** Maps a number to a list of all word phrases that can represent it */
        def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
      }
    }
}
```

Class Coder (4)

```
class Coder(words: List[String]) { ...

  /** Return all ways to encode a number as a list of words */
  def encode(number: String): Set[List[String]] =
    if (number.isEmpty)
      Set(List())
    else {
      for {
        splitPoint <- 1 to number.length
        word <- wordsForNum(number take splitPoint)
        rest <- encode(number drop splitPoint)
      } yield word :: rest
    }.toSet

  /** Maps a number to a list of all word phrases that can represent it */
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}
```

In Summary

- In the original experiment:
 - Scripting language programs were shorter and often even faster than Java/C/C++ because their developers tended to use standard collections.
- In Scala's solution:
 - Obtained a further 5x reduction in size because of the systematic use of *functional* collections.
- Scala's collection's are also easily upgradable to parallel.

Think collection transformers, not CRUD!

A Solution in Java

By Josh Bloch, author of Java collections

```
PhoneWordGenerator.java
import java.util.*;
import java.io.*;

/**
 * Translate phone numbers to word-sequences that map to the number
 * according to the standard mapping on the keypad (or dial).
 *
 * @author Josh Bloch
 */
public class PhoneWordGenerator {
    /** Map from digit-sequences to the word(s) they map to, e.g. "5278" -> ["kart", "last"] */
    private final Map<String, List<String>> wordsForNumber = new HashMap<String, List<String>>();

    /** Creates a phone word generator for the specified dictionary file */
    public PhoneWordGenerator(String dictFileName) throws IOException {
        // Invert lettersForDigit mapping
        String[] lettersForDigit = { "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };
        char[] digitForLetter = new char[26];
        for (int digit = 0; digit < 10; digit++) {
            String letters = lettersForDigit[digit];
            for (int i = 0; i < letters.length(); i++)
                digitForLetter[letters.charAt(i) - 'a'] = (char) (digit + '0');
        }

        // Process dictionary into a map from digit-sequence to the word(s) it maps to
        BufferedReader br = new BufferedReader(new FileReader(dictFileName));
        String word = null;
        try {
            while ((word = br.readLine()) != null) {
                char[] digitSeq = word.toCharArray(); // Initially contains letter sequence
                for (int i = 0; i < digitSeq.length; i++)
                    digitSeq[i] = digitForLetter[digitSeq[i] - 'a']; // Translate letter sequence to digit
                String number = new String(digitSeq);
                List<String> words = wordsForNumber.get(number);
                if (words == null)
                    wordsForNumber.put(number, words = new ArrayList<String>());
                words.add(word);
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            throw new IllegalArgumentException("Dictionary contains bogus word: " + word);
        } finally {
            br.close();
        }
    }

    /** Returns a shared instance with the standard English dictionary. Initialized on first use. */
    synchronized public static PhoneWordGenerator getInstance() throws IOException {
        return pwg != null ? pwg : (pwg = new PhoneWordGenerator("WORD.LST"));
    }

    private static PhoneWordGenerator pwg;

    /** Returns a list of the sentences represented by the specified number */
    public List<String> phoneWords(String number) {
        return sentences(number.toCharArray(), 0);
    }

    /** Returns a list of sentences represented by digits[start : digits.length) */
    private List<String> sentences(char[] digits, int start) {
        // Start with any single words represented by entire digit sequence
        List<String> result = new ArrayList<String>(words(digits, start, digits.length - start));

        // Then add any "sentences" of consisting of two or more words
        for (int firstSpace = start + 1; firstSpace < digits.length; firstSpace++)
            result.addAll(cartesianProduct(words(digits, start, firstSpace - start), sentences(digits, firstSpace)));

        return result;
    }

    /** Returns a list of single words represented by digits[offset : offset + len) */
    private List<String> words(char[] digits, int offset, int len) {
        List<String> result = wordsForNumber.get(new String(digits, offset, len));
        return result != null ? result : Collections.<String>emptyList();
    }

    /** Returns all sentences consisting of a prefix followed by a suffix */
    private static List<String> cartesianProduct(List<String> prefixes, List<String> suffixes) {
        List<String> result = new ArrayList<String>();
        for (String prefix : prefixes)
            for (String suffix : suffixes)
                result.add(prefix + " " + suffix);
        return result;
    }

    public static void main(String[] args) throws IOException {
        PhoneWordGenerator pwg = PhoneWordGenerator.getInstance();
        Console console = System.console();
        while (true) {
            System.out.print("Phone number: ");
            System.out.println(pwg.phoneWords(console.readLine()));
        }
    }
}

PhoneWordGenerator.java Top L6 (Java/L Abbrev) PhoneWordGenerator.java Bot L48 (Java/L Abbrev)
```

Going Parallel

- In Scala 2.9, collections support parallel operations.
- Two new methods:
 - `c.par` returns parallel version of collection `c`
 - `c.seq` returns sequential version of collection `c`
- A powerful tool for addressing the PPP (popular parallel programming) challenge.
- I expect this to be one of the cornerstone technologies for making use of multicores for the rest of us.

a parallel collection

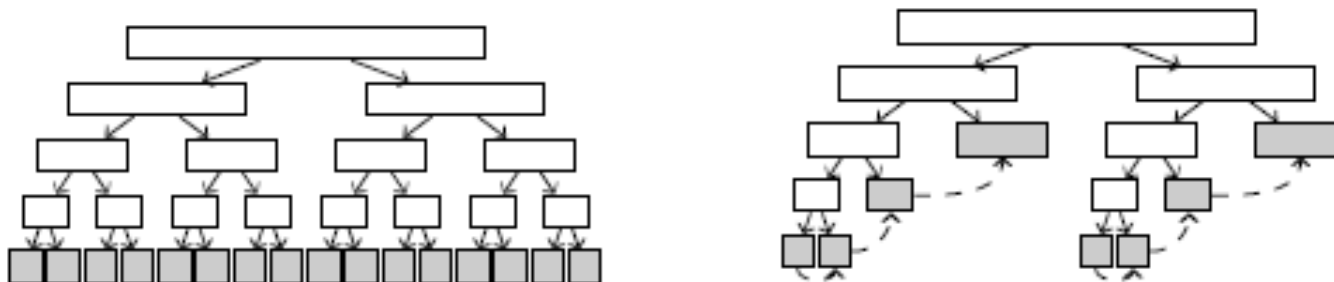
Parallel Coder

```
class Coder(words: ParVector[String]) {  
  
  private val mnemonics = Map(  
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",  
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")  
  
  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */  
  private val charCode: Map[Char, Char] =  
    for ((digit, str) <- m; letter <- str) yield (letter -> digit)  
  
  /** Maps a word to the digit string it can represent */  
  private def wordCode(word: String): String = word.toUpperCase map charCode  
  
  /** A map from digit strings to the words that represent them */  
  private val wordsForNum: Map[String, List[String]] = words groupBy wordCode  
  
  /** Return all ways to encode a number as a list of words */  
  def encode(number: String): Set[List[String]] =  
    if (number.isEmpty) Set(List())  
    else {  
      for {  
        splitPoint <- (1 to number.length).par  
        word <- wordsForNum(number take splitPoint)  
        rest <- encode(number drop splitPoint)  
      } yield word :: rest  
    }.toSet  
}
```

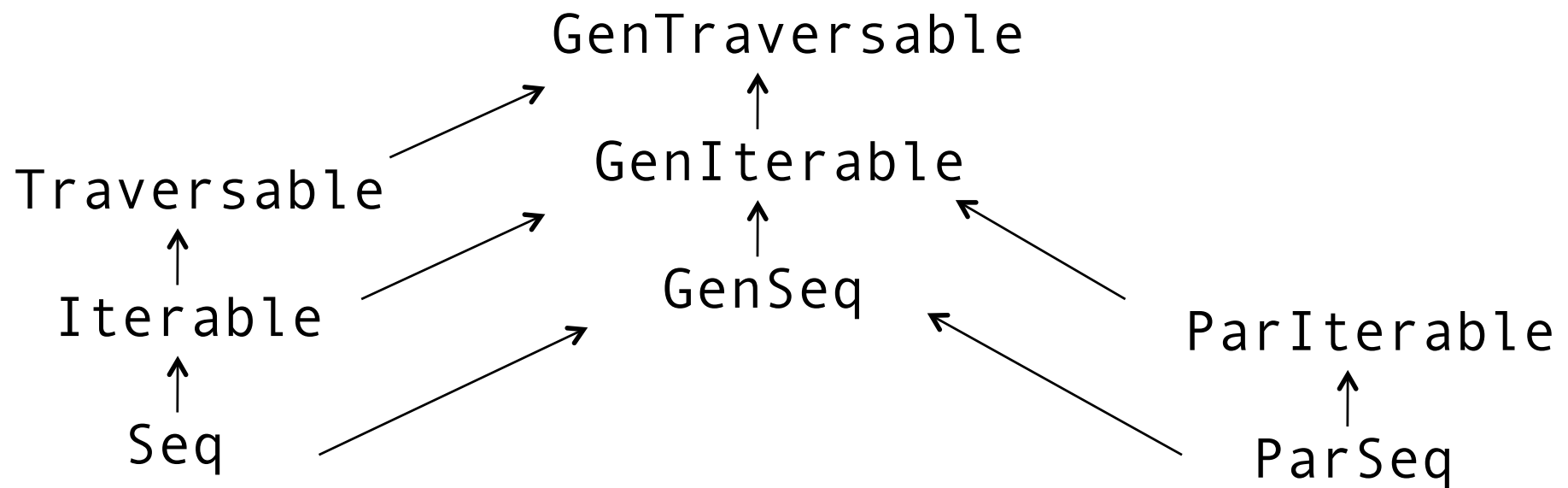
mapping sequential
collection to parallel

Parallel Collections

- Split work by number of Processors
- Each Thread has a work queue that is split exponentially. Largest on end of queue
- Granularity balance against scheduling overhead
- On completion threads “work steals” from end of other thread queues



General Collection Hierarchy



Going Distributed

- Can we get the power of parallel collections to work on 10'000s of computers?
- Hot technologies: MapReduce (Google's and Hadoop)
- But not everything is easy to fit into that mold
- Sometimes 100's of map-reduce steps are needed.
- Distributed collections retain most operations, provide a powerful frontend for MapReduce computations.
- Scala's uniform collection model is designed to also accommodate parallel and distributed.
- *cf.* Cascade (J.Suereth, D.Mahler @ Google)

But how is all this implemented?



Everything is a Library

- Collections feel like they are an organic part of Scala
- But in fact the language does not contain *any* collection-related constructs
 - no collection types
 - no collection literals
 - no collection operators
- *Everything* is done in a library
- *Everything* is extensible
 - You can write your own collections which look and feel like the standard ones

The Uniform Return Type Principle

Bulk operations return collections of the same type (constructor) as their left operand. (DWIM)

This is tricky to implement without code duplication!

In fact, Scala seems to be the only statically typed language that implements this principle!

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

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

```
scala> xs map (_ + 1)
res0: Seq[Int] = List(2, 3, 4)
```

```
scala> ys map (_ + 1)
res1: List[Int] = List(2, 3, 4)
```

Achieved with a clever combination of traits + implicits, with a sprinkling of higher-kinded types. (but all of it safely tucked away)

The Future

Scala's persistent collections are

- easy to use: *few steps to do the job*
- concise: *one word replaces a whole loop*
- safe: *type checker is really good at catching errors*
- fast: *collection ops are tuned, can be parallelized*
- scalable: *one vocabulary to work on all kinds of collections: sequential, parallel, or distributed.*

I see them play a rapidly increasing role in software development.

Working Hard to Keep it Simple

Scala feels radically different for **producers** and **consumers** of advanced libraries.

For the **consumer**:

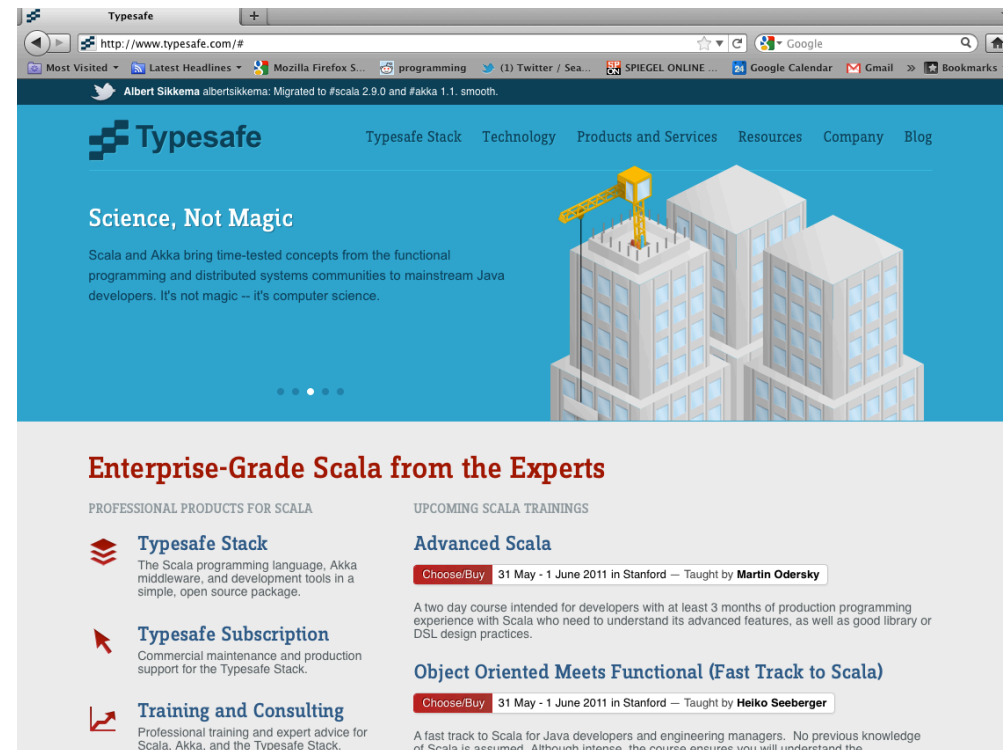
- Really easy
- Things work intuitively
- Can concentrate on domain, not implementation

For the **producer**:

- Sophisticated tool set
- Can push the boundaries of what's possible
- Requires expertise and taste

Thank You

scala-lang.org



typesafe.com

