DOI:10.1145/2591013

Scala unifies traditionally disparate programming-language philosophies to develop new components and component systems.

BY MARTIN ODERSKY AND TIARK ROMPF

Unifying Functional and Object-Oriented Programming with Scala

THOUGH IT ORIGINATED as an academic research project, Scala has seen rapid dissemination in industry and open source software development. Here, we give a high-level introduction to Scala and look to explain what makes it appealing for developers. The conceptual development of Scala began in 2001 at École polytechnique fédérale de Lausanne (EPFL) in Switzerland. The first internal version of the language appeared in 2003 when it was also taught in an undergraduate course on functional programming. The first public release was in 2004, and the 2.x series in 2006, with slightly redesigned language and a new compiler, written completely in Scala itself. Shortly thereafter, an ecosystem of open-source software began to form around it, with the Lift Web framework as an early crystallization point. Scala also began to be used in industry. A well-known adoption was Twitter, which aimed (in 2008) to rewrite its own message queue implementation in Scala. Since then, much of its core software has been written in Scala. Twitter has contributed back to open source in more than 30 released projects[24] and teaching materials.[25] Many other companies have followed suit, including LinkedIn, where Scala drives the social graph service, Klout, which uses a complete Scala stack, including the Akka distributed middleware and Play Web framework, and Foursquare, which uses Scala as the universal implementation language for its server-side systems. Large enterprises (such as Intel, Juniper Networks, and Morgan Stanley) have also adopted the language for some of their core software projects.

Gaining broad adoption quickly is rare for any programming language, especially one starting in academic research. One can argue that at least some of it could be due to circumstantial factors, but it would still be interesting to ponder what properties of the language programmers find so attractive. There are two main ingredients: First, Scala is a pragmatic language. Its main focus is to make developers more productive. Productivity needs access to a large set of libraries and tools and is why Scala was designed from the start to interoperate well with Java and run efficiently on the JVM. Almost all

» key insights

- **Scala shows that functional and object-oriented programming fit well together.**
- **This combination allows a smooth transition from modeling to efficient code.**
- **Scala also offers an impressive toolbox for expressing concurrency and parallelism.**

Java libraries are accessible from Scala without needing wrappers or other glue code. Designing Scala libraries so they can be accessed from Java code is also relatively straightforward. Moreover, Scala is a statically typed language that compiles to the same bytecodes as Java and runs at comparable speed.[6]

Several necessary compromises followed from the interoperability goal; for instance, Scala adopts Java's method overloading scheme, even though one could say multi-methods in the style of Fortress[1] would have been cleaner. Another is that Scala allows `null` pointers, called by their originator, Tony Hoare, the "billion-dollar mistake." This is again important for interoperability with Java. However, in pure Scala code `null` pointers are usually avoided in favor of the standard `Option` type.

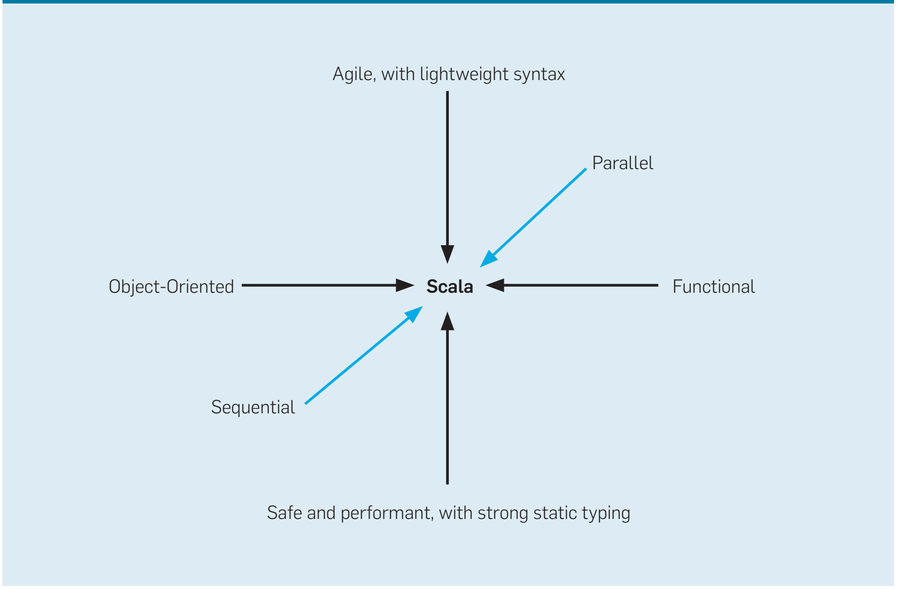
The second ingredient for Scala's impressive adoption history is that it rides, and to a certain degree drives, the emerging trend of combining functional and object-oriented programming. Functional programming has emerged since the mid-2000s as an attractive basis for software construction. One reason is the increasing importance of parallelism and distribution in computing. In the world of software development where updates require logs or replication for consistency, it is often more efficient to use replayable operations on immutable data instead; prominent examples are parallel toolkits like Akka[18] and Spark.[2]

Scala unifies areas of computing that were traditionally disparate (see the figure here). Its integration of functional and object-oriented concepts leads to a scalable language, in the sense that the same concepts work well for very small, as well as very large, programs. Scala was voted the most popular scripting language on the JVM at the JavaOne conference 2012. This was surprising, as scripting languages are usually dynamically typed, whereas Scala has an expressive, precise static type system, relying on local type inference[14,16] to obviate

Scala stairs at École polytechnique fédérale de Lausanne inspired the Scala logo (http://www.scala-lang.org) designed by Gilles Dubochet.

PHOTOGRAPH BY GILLES DUBOCHET

Scala as unifier.


Agile, with lightweight syntax
Parallel
Object-Oriented
Scala
Functional
Sequential
Safe and performant, with strong static typing

the need for most annoying type annotations. The type system and good performance characteristics of its implementation make it suitable for large mission-critical back-end applications, particularly those involving parallelism or concurrency.

Every piece of data in Scala is conceptually an object and every operation a method call. This is in contrast to functional languages in the ML family that are stratified into a core language and a module system. Scala's approach leads to an economy of features, keeping the language reasonably small in spite of its multi-paradigm nature. The functional object system also enables construction of high-level, flexible libraries that are easy for programmers to use; for instance, its hierarchy of collection classes provides a uniform framework[12] for sequences, sets, and maps that systematically cover multiple dimensions, from immutable to mutable and sequential to parallel[17] and (for sequences) from strict to lazy evaluation.

Here, we demonstrate Scala through a series of examples, starting with simple program fragments for how it identifies features from functional and object-oriented programming, then how its object model absorbs common concepts from module systems to achieve modularity and abstraction. This also demonstrates how Scala programmers can progress from a simple, high-level model to efficient sequential and parallel implementations of the model.

Combining Features

Scala combines features from the object-oriented and functional paradigms in new and interesting ways. As an example of the object-oriented style, consider the following definition of a simple class of Persons:

```
class Person(val name:
String, val age: Int) {
   override def toString =
   s"$name ($age)"
}
```

An instance of this class include fields name and age and provides an overridden toString implementation that returns a String describing the object. The syntax s"..." denotes an interpolated string that can contain computed expressions following the escape character $. As an example of the functional style, here is a way to split a list of persons according to their age:

```
val persons: List[Person] = ...
val (minors, adults) = per-
sons.partition(_.age < 18)
```

The partition method takes a sequence and a predicate and returns a pair of sequences, one consisting of elements that satisfy the predicate, the other of elements that do not. The notation _.age < 18 is shorthand for the anonymous function x => x.age < 18.

This is just one of many ways one can act on collections of objects through powerful purely functional combinators. Compared to equivalent Java code, the Scala version is much more succinct. While brevity is not universally good, the difference in cognitive overhead in this example is striking.

Instead of objects and inheritance, traditional functional languages often have algebraic data types that can be decomposed with pattern matching. An algebraic data type consists of a fixed number of alternatives, each with a different constructor. Scala allows pattern matches over class instances, obviating the need for another fundamental type constructor besides classes while keeping the language simpler and more uniform.

As an example, consider the task of writing a type named Try that contains either a returned value or an exception. A standard usage of a type like Try is to communicate exceptions across thread or machine boundaries. Consider a small use case: Assume we have a function checkAge that checks whether a person is of legal age and throws an exception if not. We may want to use Try like this:

```
Try {
   checkAge(person)
   fetchRestrictedContent()
}
```

At a glance, this code looks similar to familiar try/catch exception handling. However, the catch block is missing, and the result of the whole expression is either a Success instance carrying the result of fetchRestrictedContent or a Failure carrying an exception. While try/catch decouples the handling of error conditions from the location of the error in the program, Try also decouples the handling from the time the error occurred and its physical location; if failures happen on, say, a Web server in response to a client request, we can collect all failures relating to the request, batch them up, and send them back to the client.

Here is a possible definition of type Try followed by an explanation of the Try { ... } syntax:

```
trait Try[T] {
 def get: T
}
```

```
case class Success[T](value: T)
extends Try[T] {
 def get = value
}
case class Failure[T](ex: Ex-
ception) extends Try[T] {
 def get = throw ex
}
```

A "trait" is a generalization of Java's interface that can contain both abstract and concrete methods. The trait Try in this code defines a single abstract method get; we add more methods to it later. The trait is parameterized with a type parameter T meant to indicate the type of any returned value.

Two subclasses extend Try: Success and Failure. The case modifier in the class definitions enables pattern matching and also adds convenience methods to these classes. The Success class takes as a parameter the returned value, and the Failure class takes as parameter the thrown exception.

Consider the following client code that processes a Try value:

```
val x: Try[Int] = ...
x match {
  case Success(v) =>
    println(s"OK: $v")
  case Failure(ex: IOException)
=>
    println(s"I/O error")
  case Failure(ex) =>
    println(s"Other error $ex")
}
```

Pattern matching in Scala is done in match expressions, which are conceptually generalizations of the switch statement in C and Java. A selector value (the x to the left of match) is matched against a number of cases, each consisting of a pattern followed by an arrow => and an expression that defines the result value in case the pattern matches. The expression here includes three patterns: The first matches any value of form Success(x) where x is arbitrary; the second matches any value of form Success(ex) where ex is of type IOException; and the third matches all other Failures. In all three, the result expression is a println invocation. The return type of println is Unit, which is inhabited by the single atomic value ().

Scala's integration of functional and object-oriented concepts leads to a scalable language, in the sense that the same concepts work well for very small, as well as very large, programs.

The pattern-matching syntax and semantics is standard for a functional language. New is that pattern matching applies to object types, not algebraic data types; for instance, the second pattern matches at the same time on a Failure alternative with an embedded IOException value. Dealing with open types (such as exceptions) has been a headache for standard functional pattern matching, which applies only to closed sums with a fixed number of alternatives. Scala sidesteps this issue by matching on object hierarchies instead. Having alternatives like Success and Failure be first-class types (as opposed to only Try) gives the dual benefits of uniformity and expressiveness.

A possible criticism against this scheme is that the open-world assumption means one cannot conclusively verify that a pattern match is exhaustive, or contains a pattern for every possible selector value. Scala caters to this case by allowing sealed classes. We thus could have declared Try like this:

```
sealed trait Try[T] ...
```

In this case all immediate subclasses of Try need to be defined together with it, and exhaustiveness checking for Try expressions would be enabled; that is, a sealed trait with some extending case classes behaves like an algebraic data type.

Another criticism levied against Scala's scheme is that a set of case class definitions tends to be more verbose than the definition of an algebraic data type. This is true but must be weighed against the fact that in a typical real-world system the amount of type definitions should be small compared to the amount of code that operates on the types.

We have seen the definition of type Try but still lack a convenient way of creating Try values from expressions that can throw exceptions, as in, say:

```
Try { readFile(file) }
```

We can achieve this through the following definition:

```
object Try {
def apply[T](expr: => T) =
```

Code section 1: Graph signature.

```
trait Graphs {
  type Node
  type Edge
  def pred(e: Edge): Node
  def succ(e: Edge): Node
  type Graph <: GraphSig
  trait GraphSig {
    def nodes: Set[Node]
    def edges: Set[Edge]
    def outgoing(n: Node): Set[Edge]
    def incoming(n: Node): Set[Edge]
    def sources: Set[Node]
    def topSort: Seq[Node]
    def subGraph(nodes: Set[Node]): Graph =
      newGraph(nodes, edges filter (e =>
        (nodes contains pred(e)) &&
        (nodes contains succ(e))))
  }
  def newGraph(nodes: Set[Node], edges: Set[Edge]): Graph
}
```

```
  try Success(expr)
  catch {
    case ex: Throwable =>
Failure(ex)
  }
}
```

It includes several noteworthy points, and what gets defined is an object named Try. An object is a singleton instance that implements a set of definitions; in the example, Try implements a method apply. Object definitions are Scala's replacement of the concept of static members, which, despite being a common feature, are decidedly non-object-oriented. Instead of defining static values in a class, we define a singleton object containing the same values. A common pattern in Scala combines an object definition and a class or trait definition with the same name in a single source file. In this case, the object members are treated like the static members of a Java class.

The Try object defines a method apply that takes a type parameter T and a value parameter expr of type => T that describes a computation of type T. The type syntax => T describes "by-name" parameters. As indicated by the syntax, such parameters can be thought of as functions without an argument list. Expressions passed to them will not be evaluated at the call site; instead, they will be evaluated each time the parameter is dereferenced in the called function. Here is an example:

```
Try.apply(x / y)
```

In this case, the expression x / y will be passed unevaluated into the apply method. The apply method will then evaluate it at the point where the expr parameter is referenced as part of its Success result. If y is zero, this evaluation yields a division-by-zero error that will be caught in the enclosing try/catch expression and a Failure value will be returned.

The Try expression can be written even shorter, like this:

```
Try(x / y)
```

Or, if blocks of multiple statements are used, like this:

```
Try { ... }
```

This code makes it look as if Try is a function that can be applied to an argument and that yields a value of type Try. It also makes use of another cornerstone of Scala: Every object with an apply method can be used as a function value. This is Scala's way of making functions first class; they are simply interpreted as objects with apply methods.

Scala includes function types, written in double-arrow notation (such as Int => String is the type of functions from Int to String). But this type is simply a syntactic abbreviation for the object type Function1[Int, String]. Function1 is defined as a trait along the following lines:

```
trait Function1[S, T] {
 def apply(x: S): T
}
```

One characteristic of good functional programming style is the definition of combinator libraries that take a data type and compose it in interesting ways. What are useful combinators for Try values? One obvious choice is the combinator onSuccess, which takes a Try value and runs another Try, returning function on its result, if the value was a success. A failure, on the other hand, would be returned directly.

The classic object-oriented way of implementing apply would rely on dynamic dispatch: Define an abstract method in trait Try and one implementing method in each subclass. Alternatively, we can use pattern matching, defining the onSuccess operator as a single method on trait Try:

```
sealed trait Try[T] {
 def onSuccess[U](f: T =>
Try[U]): Try[U] = this match
{
   case Success(x) => f(x)
   case failure => failure
 }
 ...
}
```

The onSuccess combinator method can be used like this:

```
Try(x/y).onSuccess(z => Try(1/z))
```

Here, the argument to the onSuccess call is an anonymous function that takes its parameter z to the left of the double arrow => and returns the result of evaluating the expression to the right of the arrow. Evaluation of the whole expression first divides x by y and, if successful, returns a Success value containing the inverse of the result z. Any arithmetic exceptions lead to Failure results. The code is thus equivalent to:

```
Try { val z = x/y; 1/z } ===
Try(1/(x/y))
```

More generally, x.onSuccess (z=>Try(f(z))) is equivalent to Try(f(x.get)). Both notational variants are useful for various purposes. The onSuccess combinator is most useful when we want to combine several operations to feed into each other, without assigning explicit names to the intermediate results; onSuccess corresponds to the pipe operator in Unix-like shells. We can define an exec command to launch processes:

```
def exec(cmd: String): (in:
Buffer) => Try[Buffer] = ...
```

And use onSuccess to connect inputs and outputs, given some fixed input data stdin:

```
stdin.onSuccess(exec("ls"))
  .onSuccess(exec("grep ^.*\.scala"))
  .onSuccess(exec("xargs scalac"))
```

The simple all-or-nothing failure model is different from actual OS pipes; here, each command either succeeds after producing its output or fails without producing output. There is no failure after producing partial output, and output cannot be read by the next process until the previous process has terminated with success or failure. However, these features are easy to add with a level of indirection. Those familiar with the Haskell school of functional programming[8] will notice that Try implements the exception monad and onSuccess "bind" operation.

Syntactically, most Scala programmers would write the expression a bit differently, like this:

```
(stdin onSuccess exec("ls")
       onSuccess exec("grep
^.*\.scala")
       onSuccess exec("xargs
scalac"))
```

This style makes use of another pervasive Scala principle: Any binary infix operator is expanded to a method call on one of its arguments. Most operators translate to a method call with the left operand as receiver; the only exception are operators ending in :, which are resolved to the right. An example is the list cons operator ::, which prepends an element to the left of a list, as in 1::xs.

The main benefit of this Scala convention is its regularity. There are no special operators in the Scala syntax. Even operators (such as + and -) are conceptually method calls. So we could redefine onSuccess using the pipe char | as follows:

```
stdin | exec("ls") |
exec("grep ^.*\.scala") |
exec("xargs scalac")
```

Scaling Up

The previous section explored the combination of functional and object-oriented programming in a limited example where we defined one concrete type and several operations. Here, we show how Scala also applies to larger program structures, including interfaces, high-level models for specifications, and lower-level implementations.

The task is to find a general model for graphs. Abstractly, a graph is a structure consisting of nodes and edges. In practice, there are many kinds of graphs that differ in the types of information that are attached to the nodes and edges; for example, nodes might be cities and edges roads, or nodes might be persons and edges relationships.

We would like to capture what graphs have in common using one abstract structure that can then be augmented with models and algorithms in a modular way.

Code section 1 shows a trait for graphs. Simple as it is, the definition of this trait is conceptually similar to how the social graph is modeled in Scala at LinkedIn.

The trait has two abstract type members: Node and Edge. Scala is one of the few languages where objects can have not only fields and methods but also types as members. The type declarations postulate that concrete subclasses of the trait Graphs will contain some definitions of types Node and Edge. The definitions themselves are arbitrary, as long as implementations of the other abstract members of Graphs exist for them.

Next come two abstract method definitions specifying that every Graph will define methods pred and succ that take an Edge to its predecessor and successor Node. The definition of the methods is deferred to concrete subclasses.

The type of Graph itself is then specified. This is an abstract type that has as upper bound the trait GraphSig; that is, we require that any concrete implementation of type Graph conforms to the trait. GraphSig defines a set of fields and methods that apply to every graph; the two essential ones are the set of nodes and the set of edges. Furthermore, here are three convenience methods:

- For each node the set of outgoing and incoming edges;
- The set of sources, or nodes that do not have incoming edges; and
- A method subGraph that takes a set of nodes and returns a new graph consisting of these nodes and any edges of the original graph that connect them.

To show some more interesting algorithmic treatment of graphs, we also include a method topSort for the topological sorting of an acyclic graph. The method returns in a list a total

Code section 2: Graph model.

```
abstract class GraphsModel extends Graphs {
  class Graph(val nodes: Set[Node], val edges: Set[Edge])
      extends GraphSig {
    def outgoing(n: Node) = edges filter (pred(_) == n)
    def incoming(n: Node) = edges filter (succ(_) == n)
    lazy val sources = nodes filter (incoming(_).isEmpty)
    def topSort: Seq[Node] =
      if (nodes.isEmpty) List()
      else {
        require(sources.nonEmpty)
        sources.toList ++
        subGraph(nodes -- sources).topSort
      }
  }
  def newGraph(nodes: Set[Node], edges: Set[Edge]) =
    new Graph(nodes, edges)
}
```

Code section 3: Graph implementation.

```
abstract class GraphsImpl extends Graphs {
  class Graph(val nodes: Set[Node],
              val edges: Set[Edge]) extends GraphSig {
    private val outEdges, inEdges =
      new mutable.HashMap[Node, Set[Edge]] {
        override def default(key: Node) = Set()
      }
    for (e <- edges) {
      inEdges(succ(e)) += e
      outEdges(pred(e)) += e
    }
    def outgoing(n: Node) = outEdges(n)
    def incoming(n: Node) = inEdges(n)
    def topSort: Seq[Node] = {
      val indegree = new mutable.HashMap[Node,Int]
      val sorted   = new mutable.ArrayBuffer[Node]
      for (x <- nodes) {
        indegree(x) = inEdges(x).size
        if (indegree(x) == 0) sorted += x
      }
      var frontier = 0
      while (frontier < sorted.length) {
        for (e <- outEdges(sorted(frontier))) {
          val x = succ(e)
          indegree(x) -= 1
          if (indegree(x) == 0) sorted += x
        }
        frontier += 1
      }
      sorted
    }
  }
  def newGraph(nodes: Set[Node], edges: Set[Edge]) =
    new Graph(nodes, edges)
}
```

ordering of nodes in the graph consistent with the partial ordering implied by the graph edges; that is, nodes are ordered in the list in such a way the predecessor of every graph edge appears before its successor.

Unlike Java interfaces, Scala traits can have abstract, as well as concrete, members; to illustrate, trait GraphSig defines subGraph as a concrete method, whereas the other methods and fields are left abstract.

Finally, Graphs declares an abstract factory method newGraph that constructs an instance of type Graph from a set of nodes and a set of edges. Note because no concrete definition of Graph is given, newGraph cannot be defined as a concrete method at the level of Graphs because one does not know at this level how to construct a Graph. The method must be abstract.

How can Graphs be implemented? A wide range of solutions is possible. We start with a high-level model given in code section 2.

GraphsModel defines two concrete members: the Graph class and the newGraph factory method.

The Graph class takes as parameters the sets of nodes and edges that make up the graph. Note both parameters are prefixed with val. This syntax turns each parameter into a class field that serves as concrete definition of the corresponding parameter in GraphSig. With nodes and edges given, the incoming and outgoing methods can be defined as simple filter operations on sets. As the name implies, a filter operation on a collection forms a new collection that retains all elements of the original collection that satisfy a certain predicate. Note the shorthand method syntax used in these predicates where the underscore marks a parameter position; for instance (pred(_) == n) is shorthand for the anonymous function x => pred(x) == n.

Here are two other commonly used collection transformers:

- xs map f, which applies a function f to each element of a collection xs and forms a collection of the results; and
- xs flatMap f, which applies the collection-valued function f to each element of a collection xs and combines its results in a new collection using a concat or union operation.

The next definition defines sources as a lazy val, meaning the defining expression for sources will not be executed at object initialization but the first time somebody accesses the value of sources (possibly never). However, after the first access, the value will be cached, so repeated evaluations of sources are avoided. Lazy evaluation is a powerful technique for saving work in purely functional computations. The Haskell programming language makes laziness the default everywhere. Since Scala can be both imperative and functional, it has instead opted for an explicit lazy modifier.

The topSort method is a bit longer but still straightforward. If the graph is empty, the result is the empty list; otherwise, the result consists of a list containing all sources of the graph, followed by the topological sort of the subgraph formed by all other nodes. The ++ operation concatenates two collections. The implementation in code section 2 follows this specification to the word and is therefore obviously correct.

One case that still needs consideration is what to return if the graph contains cycles. No topological sorting can exist in it. We model this case through a require clause in the topSort method, specifying that every non-empty graph to be sorted must have a non-empty set of sources. If this requirement does not hold at any stage of the recursive algorithm, a graph with only cycles is left, and require will throw an exception. For implementations in the following sections, a restriction to acyclic graphs is no longer explicitly checked. Rather, we assume it as part of the implicit contract of topSort.

The final method to define is newGraph, trivial once the definition of Graph is fixed.

At the level of GraphsModel, there is still no definition of the Node and Edge types; they can be arbitrary. Here is a concrete object myGraphModel that inherits from `GraphsModel`, defines the `Node` type to be a `Person`, and defines the `Edge` type to be a pair of persons.

```
object myGraphModel extends
GraphsModel {
  type Node = Person
  type Edge = (Person, Person)
  def succ(e: Edge) = { val (s,
p) = e; s }
  def pred(e: Edge) = { val (s,
p) = e; p }
}
```

The situation where edges are pairs of nodes, for whatever the definition of node is, appears quite common. To avoid repetitive code, we can factor out this concept into a separate trait like this:

```
trait EdgesAsPairs extends
Graphs {
  type Edge = (Node, Node)
  def succ(e: Edge) = { val (s,
p) = e; s }
  def pred(e: Edge) = { val (s,
p) = e; p }
}
```

Trait `EdgesAsPairs` extends `Graphs` with a definition of `Edge` as a pair of `Node` and the corresponding definitions of `pred` and `succ`. Using this trait we can now shorten the definition of `myGraphModel` as follows:

```
object myGraphModel extends
GraphsModel with EdgesAsPairs {
    type Node = Person
}
```

The combination of two traits with the `with` connective is called "mixin composition." An important aspect of mixin composition in Scala is that it is multi-way; that is, any trait taking part in the composition can define the abstract members of any other trait, independent of the order in which the traits appear in the composition. For instance, the `myGraphModel` object in the composition defines the type `Node`, referred to in `EdgesAsPairs`, whereas the latter trait defines `Edge`, referred to in `GraphsModel`. In this sense, mixin composition is symmetric in Scala. Order of traits still matters for determining initialization order, resolving super calls, and overriding concrete definitions. As usual, they are defined through a linearization of the mixin composition graph.[11]

More efficient implementation. The `GraphsModel` class is concise and correct by construction but would win no speed record. Code section 3 presents a much faster implementation of `Graphs`, satisfying the same purely functional specification as `GraphsModel` yet relies on mutable state internally. This illustrates an important aspect of the Scala "philosophy": State and mutation are generally considered acceptable as long as one keeps the state local. If no one can observe state changes, it is as if they did not exist.

The idea to speed up graph operations in `GraphsImpl` is to do some preprocessing. Incoming and outgoing edges of a node are kept in two maps—`inEdges` and `outEdges`—initialized when the graph is created. The type of these maps is a mutable `HashMap` from `Node` to `Set[Edge]`, defining a default value for keys that were not entered explicitly; these keys are assumed to map to the empty set of nodes.

The first statements in the body of class `Graph` in code section 3 define and populate the `inEdges` and `outEdges` maps. Class initialization statements in Scala can be written directly in the class body; no separate constructor is necessary. This has the advantage that immutable values can be defined directly using the usual `val x = expr` syntax; no initializing assignments from within a constructor are necessary.

The final interesting definition in code section 3 is the one for `topSort` implemented as an imperative algorithm using a `while` loop that maintains at each step the in-degree of all nodes not yet processed. The algorithm starts by initializing the `indegree` map and the output buffer `sorted`. The output list initially contains the Graphs's sources, or the nodes with in-degree zero. When an element is added to the output list, the stored in-degree value of all its successors is reduced.

Whenever the in-degree of a node reaches zero, the node is appended to the output list. This is achieved through a single `while` loop that traverses the (growing) `sorted` buffer from the left, with variable `frontier` indicating the extent of nodes already processed, while elements are simultaneously added at the right. Intuitively, the `while` loop progresses the same way as the recursive calls in code section 2.

Going parallel. The `GraphsImpl` implementation from code section 3 is efficient on a single processor core. While this may be good enough for small- and mid-size problems, once input data exceeds a certain size we want to use all available processing power and parallelize the algorithm to run on multiple CPU cores.

Parallel programming has a reputation for being difficult and error-prone. Here, we consider a few alternatives of parallelizing the `topSort` algorithm and see how Scala's high-level abstractions allow us to restructure the algorithm to make good use of the available resources.

Scala provides a set of parallel collection classes. Each of them (such as `Seq`, `Set`, and `Map`) has a parallel counterpart (such as `ParSeq`, `ParSet`, and `ParMap`), respectively. The contract is that operations on a parallel collec-

Code section 4: Parallel `topSort` using `AtomicInteger`.

```
import java.util.concurrent.atomic.AtomicInteger
def topSort: Seq[Node] = {
  val indegree = nodes.map(n =>
    (n, new AtomicInteger(inEdges(n).size))).toMap
  def sort(frontier: ParSet[Node]): ParSeq[ParSet[Node]] =
    if (frontier.isEmpty)
      ParSeq()
    else
      frontier +: sort (
        frontier.flatMap(x =>
          outgoing(x).par.map(succ).filter(y =>
            indegree(y).decrementAndGet == 0)))
  sort(sources.par).flatten.seq
}
```

tion class (such as ParSeq) may be executed in parallel, and transformer operations (such as map and filter) will again return a parallel collection.

This way, a chain of operations myseq.map(f).map(g) on a ParSeq object myseq will synchronize after each step, but f and g on their own may be applied to elements of the collection in parallel. Methods .par and .seq can be used to convert a sequential into a parallel collection and vice versa.

A first cut at parallelizing topSort could start with the imperative code from code section 3 and add coarse-grain locks to protect the two shared data structures—the indegree map and the result buffer. Unfortunately, however, this approach does not scale, as every access to one of these objects will need to acquire the corresponding lock. The second attempt would be to use locks on a finer-grain level. Rather than protect the indegree map with one global lock, we could use a ConcurrentHashMap or roll our own by storing AtomicInteger objects instead of Ints in the map; the corresponding code is shown in code section 4. The AtomicIntegers add a layer of indirection, each of which can be changed individually through an atomic decrementAndGet operation without interfering with the other nodes.

The second point of contention is the result buffer. In code section 4 we switch back to a more functional style. The topSort method now takes the frontier of the currently processed graph as argument and returns a sequence of sets of node. Each set represents a cut through the graph of nodes at the same distance from one of the original sources. The +: operation is the generalization of the list cons operator to arbitrary sequences; it forms a new sequence from a leading element and a trailing sequence. The sequence of sets is flattened (in parallel) at the end of the method.

While this implementation is a nice improvement over coarse-grain locking, it would not exhibit very good performance for many real-world inputs. The problem is that, empirically, most graph problems are scale-free; the degree distribution often follows a power law, meaning the graph has a low diameter (longest distance between two nodes), and most nodes have only a few connections, but a few nodes have an extremely large number of connections. On Twitter, for example, most people have close to zero followers, even though a few celebrities have tens of millions; for example, in February 2014, Charlie Sheen had 10.7 million followers, Barack Obama 41.4 million, and Justin Bieber 49.6 million.

If we were to run an algorithm over cyclic graphs but implemented it in a style similar to the one in code section 4 on Twitter's graph, then for every one of Justin Bieber's almost 50 million followers, we would have to execute compare-and-swap operations on the very AtomicInteger used to hold Bieber's indegree. Individual hubs in the graph thus become bottlenecks and impede scalability.

This means we need to try a different strategy. The key is to restructure the access patterns so no two threads ever write to the same memory location concurrently. If we achieve that, we can remove the synchronization and thus the bottlenecks.

The new code is in code section 5. Like the previous version in code section 4, we use a recursive method but instead of a Map that holds AtomicInteger objects use a separate Counters data structure to hold the indegrees. Instead of directly decrementing the indegree of each target node we encounter, we first group the edges by their successor field, like this:

```
val m = frontier.
flatMap(outgoing).groupBy(succ)
```

The flatMap and groupBy operations are executed as two data-parallel steps since frontier is a ParIterable. We can use this more general type here instead of ParSet in code section 5 because the groupBy operation already ensures unique keys. The result m of the operation is a ParMap that maps each node in the next frontier to the set of its incoming edges. The algorithm proceeds by performing another parallel iteration over its elements that adjusts the indegree counters.

The key innovation is that all parallel operations iterate over subsets of nodes, and all writes from within parallel loops go to the indegree map at the loop index. Since loop indices are guaranteed to be disjoint, there can be no write conflicts if the Counters implementation is designed to

Code section 5: Parallel topSort using groupBy.

```
def topSort = {
  val indegree = new Counters(nodes.par)(inEdges(_).size)
  def sort(frontier: ParIterable[Node]): ParSeq[ParIterable[Node]] =
    if (frontier.isEmpty)
      ParSeq()
    else
      frontier +: sort {
        val m = frontier.flatMap(outgoing).groupBy(succ)
        for ((s, es) <- m if indegree.decr(s, es.size) == 0) yield s
      }
  sort(sources.par).flatten.seq
}
```

Performance evaluation.

# Nodes	Listing 2	Listing 3	Listing 4	Listing 5
2,000	27.5056	0.0082	0.0454	0.1006
20,000	—	0.1150	0.1714	0.1686
200,000	—	1.9078	1.3472	1.0096

Running time in seconds for code sections 2–5 on graphs of various sizes. Graphs have 10x as many edges as nodes. The optimized implementations are orders of magnitude faster than the straightforward model. Parallelization adds overhead for small graphs but yields speedup up to 1.9x for large graphs.

allow concurrent writes to disjoint elements. In this case, no fine-grain synchronization is necessary because synchronization happens at the level of bulk operations.

The `Counters` class is a general abstraction implemented like this:

```
class Counters[T](base:
ParSet[T])(init: T => Int) {
  private val index = base.
zipWithIndex.toMap
  private val elems = new
Array[Int](index.size)
  for (x <- base)
elems(index(x)) = init(x)
  def decr(x: T, delta: Int):
Int = {
    val idx = index(x)
    elems(idx) -= delta
    elems(idx)
 }
}
```

The constructor takes a parallel set base and an initializer `init` and uses `zipWithIndex` to assign a unique integer index to each element in `base`. The following `toMap` call turns the result set of (`T`, `Int`) pairs into a parallel map `index` with type `ParMap[T, Int]`. The actual counters are stored in an integer array `elems`. The method `decr`, which decrements the counter associated with object `x`, first looks up the index of `x`, then modifies the corresponding slot in `elems`. Counters for different elements can thus be written to concurrently without interference.

The result buffer handling in code section 5 is similar to the previous implementation in code section 4. Nodes are added en bloc, and the result buffer is flattened (in parallel) at the end of method `topSort`.

We have thus explored how Scala's concepts apply to larger program structures and how the language and the standard library support a scalable development style, enabling programmers to start with a high-level, "obviously correct" implementation that can be refined gradually into more sophisticated versions.

A quick performance evaluation is outlined in the table here. For a small graph with just 2,000 nodes and 20,000 edges, the naïve implementation takes 27 seconds on an eight-core Intel X5550 CPU at 2.67 GHz, whereas the fast sequential version (code section 3) runs in under 0.01 seconds. Compared to the optimized sequential version, parallelization actually results in up to 10x slower performance for small graphs but yields speedups of up to 1.9x for a graph with 200,000 nodes and two million edges. All benchmarks were run 10 times; the numbers reported in the table are averages of the last five runs. Input graphs were created using the R-Mat algorithm,[5] reversing conflicting edges to make the graphs acyclic. The overall achievable performance and parallel speedups depend a lot on the structure of the input data; for example, picking a sparser input graph with two million nodes and only 20,000 edges yields speedups of up to 3.5x compared to the optimized sequential version.

The key is to restructure the access patterns so no two threads ever write to the same memory location concurrently.

Finally, how can we convince ourselves that the efficient implementations actually conform to the high-level model? Since all versions are executable code with the same interface, it is easy to implement automated test suites using one of the available testing frameworks.

Conclusion

We have offered a high-level introduction to Scala, explaining what makes it appealing to developers, especially its focus on pragmatic choices that unify traditionally disparate programming-language philosophies (such as object-oriented and functional programming). The key lesson is these philosophies need not be contradictory in practice. Regarding functional and object-oriented programming, one fundamental choice is where to define pieces of functionality; for example, we defined `pred` and `succ` on the level of `Graphs` so they are functions from edges to nodes. A more object-oriented approach would be to put them in a bound of the edge type itself; that is, every edge would have parameterless `pred` and `succ` methods. One thing to keep in mind is that this approach would have prevented defining `type Edge = (Node, Node)` because tuples do not have these methods defined. Neither of the variants is necessarily better than the other, and Scala gives programmers the choice. Choice also involves responsibility, and in many cases novice Scala programmers need guidance to develop an intuitive sense of how to structure programs effectively. Premature abstraction is a common pitfall. Ultimately though, every piece of data is conceptually an object and every operation is a method call. All functionality is thus a member of some object. Research branches of the language[19] go even further, defining control structures (such as conditionals, loops, and pattern matching) as method calls.

The focus on objects and modularity makes Scala a library-centric language; since everything is an object, everything is a library module. Consequently, Scala makes it easy for programmers to define high-level and efficient libraries and frameworks—important for scaling programs from small scripts to large software systems. Its syntactic flexibility, paired with an expressive type system, makes Scala a popular choice for embedding domain-specific languages (DSLs). The main language constructs for component composition are based on traits that can contain other types, including abstract ones, as members.[13] Scala's traits occupy some middle ground between mixins[3] and Schärli's traits.[22] As in the latter, they support symmetric composition so mutual dependencies between traits are allowed, but, as with traditional mixins, Scala traits also allow stackable modifications that are resolved through a linearization scheme. Another important abstraction mechanism in Scala is implicit parameters that let one emulate the essential capabilities of Haskell's type classes.[15]

Performance scalability is another important dimension. We have seen how we can optimize and parallelize programs using libraries included in the standard Scala distribution. Clients of the graph abstraction did not need to be changed when the internal implementation was replaced with a parallel one. For many real-world applications this level of performance is sufficient. However, we cannot expect to squeeze every last drop of performance out of modern hardware platforms, as with dedicated graph-processing languages (such as GraphLab[10] and Green Marl[7]). With a bit more effort, though, programmers can achieve even these levels of performance by adding runtime compilation and code generation to their programs. Lightweight modular staging (LMS)[20] and Delite[4,9] are a set of techniques and frameworks that enable embedded DSLs and "active" libraries that generate code from high-level Scala expressions at runtime, even for heterogeneous low-level target languages (such as C, CUDA, and OpenCL). DSLs developed through Delite have been shown to perform competitively with hand-optimized C code. For graph processing, the Opti-Graph DSL[23] (embedded in Scala) performs on par with the standalone language Green Marl. Many Scala features are crucial for LMS and Delite to implement compiler optimizations in a modular and extensible way.[21]

Scala's blend of traditionally disparate philosophies provides benefits greater than the sum of all these parts. ■

References

1. Allen, E.E., Hallett, J.J., Luchangco, V., Ryu, S., and Steele, G.L., Jr. Modular multiple dispatch with multiple inheritance. *In Proceedings of the 2007 ACM Symposium on Applied Computing*, Y. Cho, R.L. Wainwright, H. Haddad, S.Y. Shin, and Y.W. Koo, Eds. (Seoul, Mar. 11–15). ACM Press, New York, 2007, 1117–1121.
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., and Zaharia, M. A view of cloud computing. *Commun. ACM 53*, 4 (Apr. 2010), 50–58.
3. Bracha, G. and Cook, W.R. Mixin-based inheritance. In *Proceedings of OOPSLA/ECOOP*, A. Yonezawa, Ed. (Ottawa, Oct. 21–25). ACM Press, New York, 1990, 303–311.
4. Brown, K.J., Sujeeth, A.K., Lee, H., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques* (Galveston Island, TX, Oct. 10–14), IEEE Computer Society Press, 2011, 89–100.
5. Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT: A recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining*, M.W. Berry, U. Dayal, C. Kamath, and D.B. Skillicorn, Eds. (Lake Buena Vista, FL, Apr. 22–24). SIAM, 2004, 442–446.
6. Fulgham, B. *The Computer Language Benchmark Game*, 2013; http://benchmarksgame.alioth.debian.org/
7. Hong, S., Chafi, H., Sedlar, E., and Olukotun, K. Green-marl: A DSL for easy and efficient graph analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (London, Mar. 3–7). ACM Press, New York, 2012, 349–362.
8. Hudak, P. *The Haskell School of Expression: Learning Functional Programming Through Multimedia.* Cambridge University Press, Cambridge, U.K., 2000.
9. Lee, H., Brown, K.J., Sujeeth, A.K., Chafi, H., Rompf, T., Odersky, M., and Olukotun, K. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro 31*, 1 (Jan.-Feb. 2011), 42–53.
10. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J.M. Graphlab: A new framework for parallel machine learning. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence*, P. Grünwald and P. Spirtes, Eds. (Catalina Island, CA, July 8–11). AUAI Press, 2010, 340–349.
11. Odersky, M. *The Scala Language Specification, Version 2.9.* EPFL, Lausanne, Switzerland, Feb. 2011; http://www.scala-lang.org/docu/manuals.html
12. Odersky, M. and Moors, A. Fighting bit rot with types (experience report: Scala collections). In *Proceedings of the Annual Conference on Foundations of Software Technology and Theoretical Computer Science, Vol. 4 of LIPIcs Leibniz International Proceedings in Informatics*, R. Kannan and K.N. Kumar, Eds. (Kanpur, India, Dec. 15–17). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009, 427–451.
13. Odersky, M. and Zenger, M. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, R.E. Johnson and R.P. Gabriel, Eds. (San Diego, Oct. 16–20). ACM Press, New York, 2005, 41–57.
14. Odersky, M., Zenger, M., and Zenger, C. Colored local type inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, C. Hankin and D. Schmidt, Eds. (London, Jan. 17–19). ACM Press, New York, 2001, 41–53.
15. Oliveira, B.C.d.S., Moors, A., and Odersky, M. Type classes as objects and implicits. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, W.R. Cook, S. Clarke, and M.C. Rinard, Eds. (Reno, NV, Oct. 17–21). ACM Press, New York, 2010, 341–360.
16. Pierce, B.C. and Turner, D.N. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, D.B. McQueen and L. Cardelli, Eds. (San Diego, Jan. 19–21). ACM Press, New York, 1998, 252–265.
17. Prokopec, A., Bagwell, P., Rompf, T., and Odersky, M. A generic parallel collection framework. In *Proceedings of the 17th International Conference on Parallel Processing, Vol. 6853 of Lecture Notes in Computer Science*, E. Jeannot, R. Namyst, and J. Roman, Eds. (Bordeaux, France, Aug. 29–Sept. 2). Springer, New York, 2011, 136–147.
18. Roestenburg, R. and Bakker, R. *Akka in Action.* Manning Publications Co., Shelter Island, NY, 2013.
19. Rompf, T., Amin, N., Moors, A., Haller, P., and Odersky, M. Scala-virtualized: Linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* (Sept. 2013),1–43.
20. Rompf, T. and Odersky, M. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM 55*, 6 (June 2012), 121–130.
21. Rompf, T., Sujeeth, A.K., Amin, N., Brown, K., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., and Odersky, M. Optimizing data structures in high-level programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, R. Giacobazzi and R. Cousot, Eds. (Rome, Italy, Jan. 23–25). ACM Press, New York, 2013, 497–510.
22. Schärli, S. Ducasse, Nierstrasz, O., and Black, A.P. Traits: Composable units of behaviour. In *Proceedings of the 17th European Conference on Object-Oriented Programming, Vol. 2743 of Lecture Notes in Computer Science*, L. Cardelli, Ed. (Darmstadt, Germany, July 21–25). Springer, New York, 2003, 248–274.
23. Sujeeth, A.K., Rompf, T., Brown, K.J., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., and Olukotun, K. Composition and reuse with compiled domain-specific languages. In *Proceedings of the 27th European Conference on Object-Oriented Programming, Vol. 7920 of Lecture Notes in Computer Science*, G. Castagna, Ed. (Montpellier, France, July 1–5). Springer, New York, 2013, 52–78.
24. Twitter. Open source projects; http://twitter.github.com
25. Twitter. Scala-school!; http://twitter.github.com/scala.school

Martin Odersky (martin.odersky@epfl.ch) is a professor of computer science at EPFL in Lausanne, Switzerland, co-founder of Typesafe, creator of the Scala language, and a fellow of the ACM.

Tiark Rompf (tiark.rompf@epfl.ch) is a researcher at Oracle Labs and EPFL in Lausanne, Switzerland.