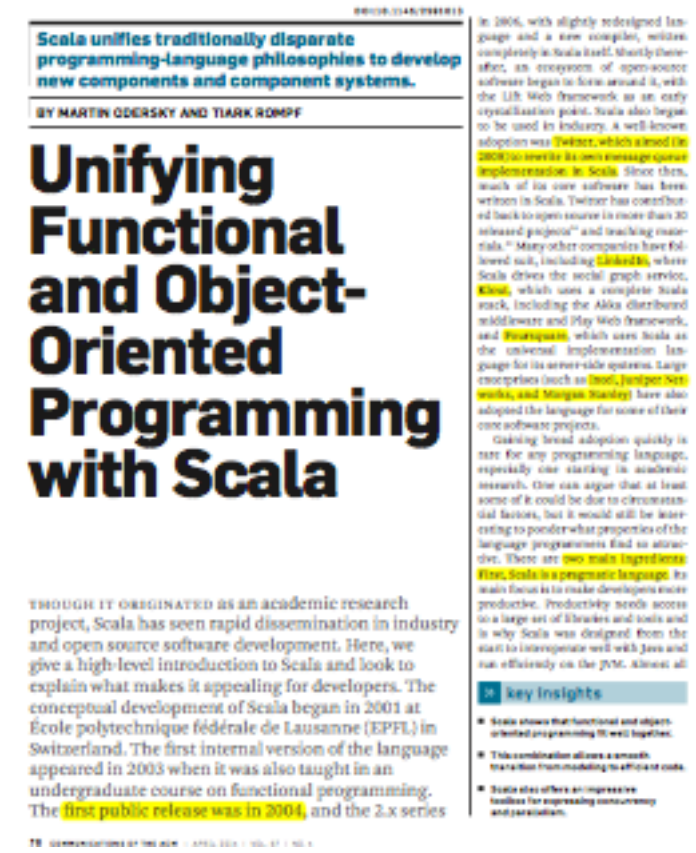
Unifying functional and object-oriented programming with Scala

齐琦， 海南大学
计算机发展前沿系列课程

原文引用

- Odersky, M. and T. Rompf (2014). "Unifying functional and object-oriented programming with Scala." Communications of the ACM **57**(4): 76–86.



contributed articles

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

key insights

SCALA的发展

- Conceptual development by Martin Odersky—2001 at EPFL
- First internal version – 2003
- First public release – 2004
- 2.x series – 2006
  - Slightly redesigned language
  - A new compiler, written completely in Scala itself
- Shortly thereafter
  - Open-source software, (Lift Web framework)
  - In industry
    - Twitter(2008),rewrote its message queue in Scala, and much of its core software; contributed open-source and teaching materials (30 projects)
    - LinkedIn, Scala for its social graph service
    - Klout, uses Akka and Play Web framework
    - Foursquare, for its server-side systems
    - Other large enterprises: Intel, Juniper Networks, and Morgan Stanley

Reason of attractiveness

- Scala is a pragmatic language
  - Focus is to make developers more productive
  - Statically typed, compiles to the same bytecodes as Java, and runs at comparable speed on the JVM
  - Compromises followed from the interoperability
    - Adopts Java's method overloading scheme, even though exists better ones
    - Null pointers though avoided in favor of Option type
- Rides and drives to some degree, on the emerging trend of combining functional and object-oriented programming
  - FP's emergence
    - Increasing importance of parallelism and distribution in computing
    - Re-playable operations on immutable data, instead of requiring logs or replication to updates
  - Integration leads to scalable (the same concepts work well for very small, as well as very large, programs)

Another attractiveness: Type system

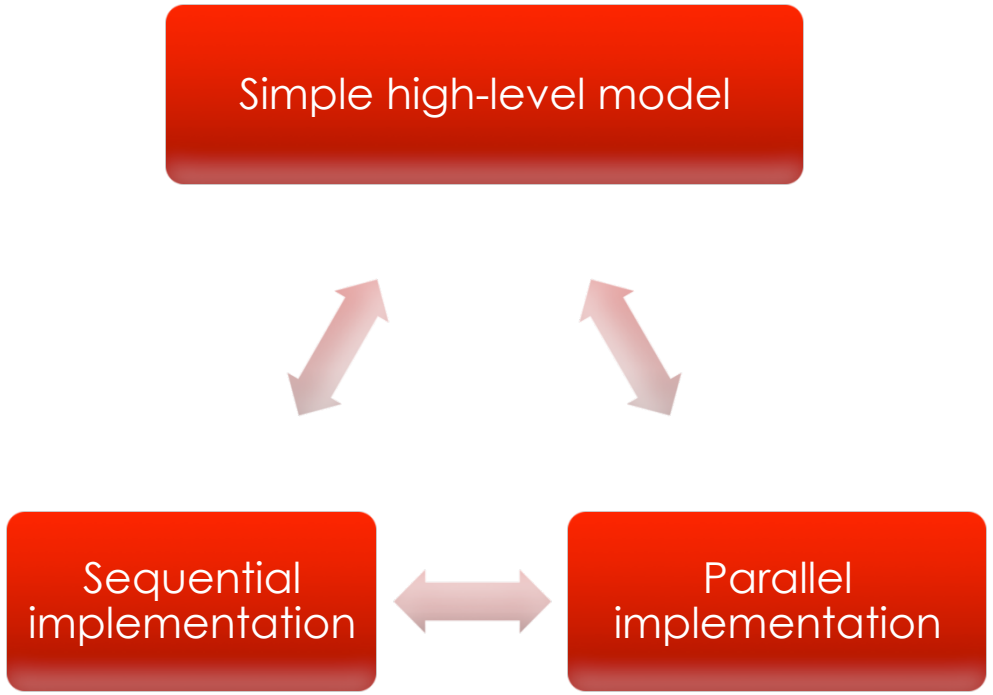
- Static type system
  - Voted the most popular scripting language on the JVM at JavaOne conference 2012, Surprisingly
  - Scripting languages usually dynamically typed, whereas Scala expressive, precise static type system, local type inference, avoid need most annoying type annotations
  - Suitable for large mission-critical back-end applications
  - Particularly those involving parallelism or concurrency

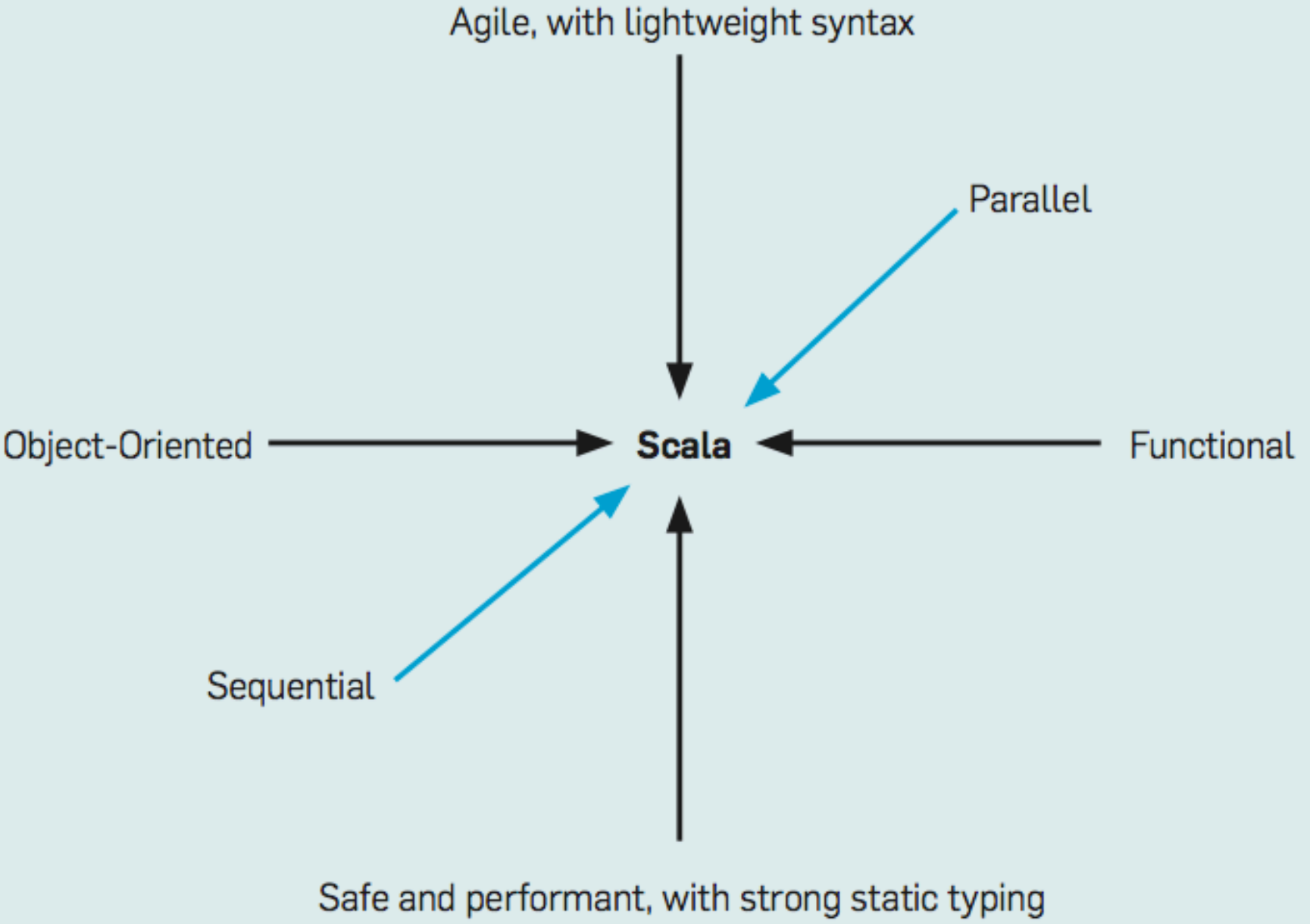
Scala's approach

- Every piece of data in Scala is conceptually an object and every operation a method call
  - Exception: build-in data types
- An economy of features, reasonably small, though multi-paradigm nature
- Functional object system enables construction of high-level, flexible libraries
  - Collection classes – a uniform framework for sequences, sets, maps ； (objects + higher-order functions)
    - immutable, mutable
    - Sequential, parallel
    - Strict, lazy evaluation

Scala's approach, cont.

- Object model absorbs common concepts from module systems;
- Achieves modularity and abstraction




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

Combining Features: Functional Style

- combinators
  - Succinct

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

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

Functional Style

- Algebraic data types (by trait, class, case class)
  - pattern matching for decomposition

```
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
}
```

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

Combining Features

- Trait
  - A generalization of Java's interface
  - Both abstract and concrete methods
- Case subclasses
  - Enables pattern matching
  - Adds convenience methods to the classes

```
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")
}
```

Combining Features

- Pattern matching
  - Standard for a functional language
  - New is that it applies to object types, not algebraic data types
  - Matching on object hierarchies
  - Fixed number of cases
    - Sealed classes
    - A sealed trait with some case subclasses, behaves like an algebraic data type.

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

Combining Features

- Object
  - Replaces static members
  - Together with a class or trait with the same name in a single source
  - Treated like static members of a Java class
- => T : "by-name" parameters
  - As functions without an argument list
  - Evaluated each time the parameter is dereferenced in the called function

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

```
object Try {
def apply[T](expr: => T) =
   try Success(expr)
   catch {
     case ex: Throwable =>
 Failure(ex)
   }
 }
```

Combining Features

- Every object with an apply method can be used as a function value
  - Scala's way of making functions first class
  - Interpreted as objects with apply methods

```
Try.apply(x / y)

Try(x / y)
```

- Function types(double-arrow notation)
  - Int => String, Syntactic abbreviation for object type Function1[Int, String]

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

Define combinator libraries

- A good functional programming style
- Take a data type and compose it in some way
- A combinator for Try values
  - onSuccess, a method on trait Try by pattern matching

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

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

Operators

- Binary infix operator
  - A method call with left operand as receiver
  - Exception: operators ending in :
    - List cons operator, ::
- No special operators in Scala syntax
  - Even + and – are conceptually method calls

Scaling up

- Larger program structures, high-level models for specifications, and lower-level implementations

- Exampel : graph model
  - One abstract structure captures the common, augmented with models and algorithms in a modular way

Graph signature

In Scala, Types can be members, besides fields and methods

Graph type upper bound

Traits can have abstract and concrete members

```
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
}
```

Lazy val

++: concatenates two collections

Require, at any stage of recursion

```
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)
}
```

myGraphModel

```
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 }
}
```

Multi-way Mixin composition

Order of traits matters for initialization order(resolving super calls, overriding definitions).

```
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 }
}
```

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

Faster implementation

- GraphsModel is concise, but not efficient
- GraphsImpl , a faster version; mutable state internally
  - State and mutation are acceptable when they are local

```
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)
}
```


Class Initialization can be written directly in class body; no separate constructor necessary.

Topsort as an imperative algorithm using while loop.

Going parallel

- GraphsImpl implementation efficient on a single processor core.
- Input data size large; to parallelize to run on multiple CPU cores.
- Parallel programming difficult and error-prone.
- Scala's high-level abstractions can help.
  - Parallel collection classes (ParSeq, ParSet, ParMap, etc.)
  - Operations on these may be executed in parallel
  - Transformer operations (map, filter) will return a parallel collection.

Going parallel

- ParSeq object: myseq
  - myseq.map(f).map(g)
  - Synchronize after each step
  - f and g on their own may be operated in parallel
  - .par and .seq, conversion between sequential and parallel collection

```
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
}
```


- Uses locks on a fine-grain level
- AtomicInteger objects
  - Atomic decrementAndGet operation changes each node without interfering with other nodes
- A functional style of topsort

Scalability issue

- The previous is better than coarse-grain locking, but not good performing for real-world inputs
  - Most graphs have a low diameter(longest distance between two nodes)
  - Most nodes have a few connections, but a few nodes large number of connections
  - parallel operations would not balance; individual hubs become bottlenecks and impede scalability

```
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
}
```

- key is to restructure the access patterns, so
  - No two threads write to same memory location concurrently
  - So that, can remove the synchronization
- Parallel topSort using groupBy
- Key innovation: all parallel operations iterate over subsets of nodes
- Counters allow concurrent wirtes to disjoint elements

Counters class

- Counters stored in array elems
  - Different index corresponds to different object x, and different slot in elems
  - Counters for different elements can be written to concurrently without interference

```
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)
 }
}
```

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.

- 8-core Intel X5550 CPU at 2.67GHz; Acyclic graphs
- Parallelization actually results in up to 10x slower for small graphs, but 1.9x speedup for 200,000 nodes and two million edges
- Depends on input data structure
  - A sparser graph(two million nodes, 20,000 edges), up to 3.5x speedup

Conclusion

- Pragmatic choices that unify traditionally disparate programming-language philosophies(object-oriented and functional programming).
- Key lesson is that they need not be contradictory in practice.
- Choice about where to define functionality
  - Functional: Pred, succ on Graphs level, from edges to nodes
  - Object-oriented: put them in edge type, parameterless
    - Type Edge=(Node, Node) would not work, tuples not have those methods
- Ultmately 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

Conclusion, cont.

- Focus on objects and modularity
  - A library-centric language
  - Everything is an object, everything a library module
- Popular choice for embedding domain-specific languages(DSLs)
  - Syntactic flexibility
  - Expressive type system
- Main constructs for component composition are based on traits
  - Can contain other types as members
  - Mutual dependencies between traits are allowed(symmetric composition)
  - Stackable modifications, resolved through a linearization scheme

Conclusion, cont.

- Another important abstraction mechanism
  - Implicit parameters
- Performance scalability
  - Graph client not need to change when its implementation replaced by a parallel one.
  - Lightweight modular staging(LMS) and Delite
    - Enable embedded DSLs; generate code from high-level Scala expressions at runtime
    - Can generate heterogeneous low-level target languages(C, CUDA, OpenCL)
    - Perform competitively with hand-optimized C
    - Many Scala features crucial for LMS and Delite to implement compiler optimizations in a modular and extensible way.