Scala - A Powerful and Scalable Function-Objective Programming Language

Troy Hu and Benjamin Killeen

***Abstract*—Insert abstract here.**

I. Introduction

Software components are simply self-contained parts of software used by larger parts or entire applications. Modern software typically relies on many common components like hashable types or iterable data structures to use as building blocks. Component abstraction enables the generalization of implementation for these features, greatly reducing duplication of effort. It is one of the most powerful tools in the programmer's utility belt, a fact which the designers of the Scala programming language aimed to exploit [1]. In the early 2000s, existing languages had little support for type-sound component abstraction, including widely used languages like Java and C#. Odersky *et al.* address this shortcoming with a language that combines elements of object-oriented and functional programming in a statically typed environment. Scala, which gets its name from "scalable," provides a powerful interface for abstraction within a development framework intended for mass adoption.
In pursuit of usability, Scala borrows many syntactic elements from Java and C#, and it integrates smoothly with components from these languages. In fact, the Scala library includes standard Java objects like `java.lang.String`, as shown in **??**. Scala code can take advantage of existing implementations in Java, and in the end it compiles to Java bytecode, making Scala packages available to Java programmers. At the same time, Scala maintains a distinct programming paradigm from either Java or C#. It discards some features from these languages, and it develops completely novel ideas from the νObj calculus [2]. The example in Fig. **??** highlights syntactic similarities between Java and Scala, comparing two implementations of the same program. Note how Java prepends type declarations before terms, whereas Scala affixes type declarations using the : operator. This and other changes effect a terser, more expressive syntax overall.
Of course, Scala's foremost strength comes from its typing system. Abstract class definitions and path-dependent types utilize the νObj Calculus, enabling incredible flexibility through the use of traits and mixins [2]. Somewhat akin to Java's abstract classes, traits allow a programmer to rely on abstract methods for common functionalities. For example, the `Equiv[T]` trait in Fig. **??** represents an equivalence relation on the type `T`, abstracting the definition of `eq` on which a concrete method, `neq` depends. Mixins enable a class to inherit from multiple traits. For instance, one might use `Equiv` in conjunction with a

```java
// Java:
class PrintOptions {
  public static void main(String[] args) {
    System.out.println("Options selected:");
    for (int i = 0; i < args.length; i++)
      if (args[i].startsWith("-"))
        System.out.println(" " + args.substring(1))
  }
}
```

```scala
// Scala:
object PrintOptions {
  def main(args: Array[String]): unit = {
    println("Options selected:")
    for (val arg <- args)
      if (arg.startsWith("-"))
        println(" " + arg.substring(1))
  }
}
```

Listing 1: Notice how Scala's general syntax and structure are similar to Java's. At the same time, there are some visible differences, e.g., unit is returned in the Scala implementation instead of void in the Java implementation.

The strength of Scala's typing system depends on the nuObj calculus.
Large parts of Scala's typing system are unique to the language. Scala's abstract type definitions and path-dependent types utilize νObj Calculus (See Section **??**). Additionally, Scala implements modular mixin composition which combines the advantages of mixins and traits. Traits in Scala are essentially the equivalent of abstract classes in Java while mixins are classes/traits in which other non-child traits can draw methods from.
Scala also has a uniform object model. That is, every value is an object and every operation is a call to a method. For example, the boolean true itself is an object (S singleton object, in fact. See Section II for definition of singleton object). At the same time, Scala includes functional programming aspects such functions being first-class values (i.e. functions can be passed as values) and pattern matching. For pattern matching specifically, Scala allows objects themselves to be decomposed. This language also implements powerful and novel abstraction concepts for types and values. For example, unlike Java abstract classes, traits can include method implementations or fields.

II. Related Work

We did not know what νObj calculus was and therefore read the paper A Nominal Theory of Objects with Dependent Types by Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger to better understand the concept. Because our

main focus is specifically on understanding Scala and not on learning the νObj calculus (which itself is a topic worthy of its own summary paper), we only provide a general description of νObj calculus below.
According to Odersky et al., νObj calculus, broadly, is a calculus and dependent type system for classes and objects can have types as members. Note that a dependent type is a type that is defined by some value. The calculuss type system is well typed and can implement crucial aspects of Javas inner class system, virtual types, and family polymorphism. Moreover, this type system can also model SML-style modules and functors. νObj calculus diverges from standard type systems for objects in three significant ways:

- Instead of only objects being primitive, classes are also primitive. We can view classes as first class as they can be the result of a term evaluation and be associated with labels.
- Object records are not passed around during evaluation and type checking. Rather, every object has a name reference that is passed around.
- Object types can be expressed using (possibly nominal) type components.

We leave Figures 2 (syntax) and 3 (type assignment) below for the reader's interest.

Syntax

x, y, z	Name		
l, m, n	Term label	L, M, N	Type label
s, t, u ::=	Term	S, T, U ::=	Type
x	Variable	$p.\mathbf{type}$	Singleton
$t.l$	Selection	$T\bullet L$	Type selection
$\nu x \leftarrow t\ ;\ u$	New object	$\{x \vert\ \overline{D}\}$	Record type (=:: R)
$[x : S \vert\ \overline{d}]$	Class template	$[x : S \vert\ \overline{D}]$	Class type
$t \mathbin{\&_S} u$	Composition	$T \mathbin{\&} U$	Compound type
d ::=	Definition	D ::=	Declaration
$l = t$	Term definition	$l : T$	Term declaration
$L \preceq T$	Type definition	$L \preceq: T$	Type declaration
p ::=	Path	$\preceq:$ ::=	Type binder
$x \mid p.l$		$=$	Type alias
v ::=	Value	$\prec$	New type
$x \mid [x : S \vert\ \overline{d}]$		$<:$	Abstract type
		$\preceq$ ::=	Concrete type binder
		$= \mid \prec$	

Structural Equivalence α-renaming of bound variables x, plus

(extrude) $e\langle \nu x \leftarrow t\ ;\ u\rangle \equiv \nu x \leftarrow t\ ;\ e\langle u\rangle$
if $x \notin fn(e),\ bn(e) \cap fn(x, t) = \emptyset$

Reduction

(select) $\nu x \leftarrow [x : S \vert\ \overline{d}, l = v]\ ;\ e\langle x.l\rangle \rightarrow \nu x \leftarrow [x : S \vert\ \overline{d}, l = v]\ ;\ e\langle v\rangle$
if $bn(e) \cap fn(x, v) = \emptyset$

(mix) $[x : S_1 \vert\ \overline{d}_1] \mathbin{\&_S} [x : S_2 \vert\ \overline{d}_2] \rightarrow [x : S \vert\ \overline{d}_1 \uplus \overline{d}_2]$

where evaluation context

$e ::= \langle\rangle \mid e.l \mid e \mathbin{\&_S} t \mid t \mathbin{\&_S} e \mid \nu x \leftarrow t\ ;\ e \mid \nu x \leftarrow e\ ;\ t \mid \nu x \leftarrow [x : S \vert\ \overline{d}, l = e]\ ;\ t$

Fig. 1: The syntax of the νObj calculus

(Var) $$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash t : T,\quad T \ni (l : U)}{\Gamma \vdash t.l : U}$$ (Sel)

(VarPath) $$\frac{\Gamma \vdash x : R}{\Gamma \vdash x : x.\mathbf{type}}$$

$$\frac{\Gamma \vdash t : p.\mathbf{type},\quad t.l : R}{\Gamma \vdash t.l : p.l.\mathbf{type}}$$ (SelPath)

(Sub) $$\frac{\Gamma \vdash t : T,\quad T \leq U}{\Gamma \vdash t : U}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : [x : S \vert\ \overline{D}],\quad S \prec \{x \vert\ \overline{D}\} \\ \Gamma, x : S \vdash u : U \qquad x \notin \mathrm{fn}(U)\end{array}}{\Gamma \vdash (\nu x \leftarrow t\ ;\ u) : U}$$ (New)

(Class) $$\frac{\begin{array}{c}\Gamma \vdash S\ \mathrm{wf} \qquad \Gamma, x : S \vdash \overline{D}\ \mathrm{wf},\quad t_i : T_i \\ t_i\ \textit{contractive in}\ x \qquad (i \in 1..n)\end{array}}{\Gamma \vdash [x : S \vert\ \overline{D}, l_i = {t_i}^{i \in 1..n}] : [x : S \vert\ \overline{D}, l_i : {T_i}^{i \in 1..n}]}$$

(&) $$\frac{\Gamma \vdash t_i : [x : S_i \vert\ \overline{D}_i] \qquad \Gamma \vdash S\ \mathrm{wf},\quad S \leq S_i \qquad (i = 1, 2)}{\Gamma \vdash t_1 \mathbin{\&_S} t_2 : [x : S \mid \overline{D}_1 \uplus \overline{D}_2]}$$

Fig. 2: The typing rules of the νObj calculus

```
package nat

import ord._

trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
  def + (that: Nat): Nat = if (that.isZero) this else this.succ + that.
  def - (that: Nat): Nat = if (that.isZero) this else this.pred - that.
  def * (that: Nat): Nat = if (that.isZero) that else this + this * tha
  def ^ (that: Nat): Nat = if (that.isZero) that.succ else this * (this
  def <= (that: Nat): Boolean = NatOrdering.leq(this, that)
  def >= (that: Nat): Boolean = NatOrdering.geq(this, that)
  def <  (that: Nat): Boolean = NatOrdering.lt(this, that)
  def >  (that: Nat): Boolean = NatOrdering.gt(this, that)
  def == (that: Nat): Boolean = NatOrdering.eq(this, that)
  def != (that: Nat): Boolean = NatOrdering.neq(this, that)
}

object NatOrdering extends Ordering[Nat] {
  def leq (x: Nat, y: Nat): Boolean =
    (x.isZero, y.isZero) match {
      case (true, _) => true
      case (false, true) => false
      case (false, false) => leq(x.pred, y.pred)
    }
}

object Zero extends Nat {
  override def toString () = "Z"
  def isZero = true
  def succ = Succ(this)
  def pred = this
}

case class Succ(v: Nat) extends Nat{
  override def toString () = s"S($v)"
  def isZero = false
  def succ = Succ(this)
  def pred = v
}
```

Fig. 3: An example outlining Scala classes.

III. Interesting Technical Details

A. Singleton Objects

Scala is very unique in that it implements Singleton Objects. Singleton Objects are classes that can only have one instance while a Scala program is running. Such classes are defined almost exactly the same as normal Scala classes. The only difference is that the `class` modifier is replaced by the `object` modifier. Some examples of singleton objects include: a class representation of Zero and the PrintOptions example from above. Note that Singleton Objects are used extensively when Scala uses a Java class. Every static member of a Java class is stored in a Singleton Object.

B. *Unified Object Model*

In Scala, every value is an object and every class is a subtype of the `Any` class. Below the `Any` class, Scala classes can generally be divided into two groups: value classes that inherit from the `AnyVal` class and reference classes that inherit from the `AnyRef` class. Note that the value classes that are not `AnyVal` do not subtype each other. This means, for example, that the `Int` class is not a subtype of the `Float` class. Scala prevents such subtyping because the language forces an invariant that when a value is interpreted in a subclass and a super class, the values's representation does not change. For example, both `Int` and `Float` maintain a `MaxValue` member. However, the `MaxValue` of `Float` is different from the `MaxValue` of `Int`. Instead, Scala implements **views** (discussed below) which allow from implicit conversions from one type to another. As expected of an object-oriented language, class `Null` is a subtype of every reference type. However, since Scala is part functional, the language also has a `Nothing` type, which represents an empty type, is a subtype of every single type. For example, `Nil` is defined as `List[Nothing]`. Note that `Nothing` allows for the implementation of options, another functional concept, in Scala.

Another major aspect of Scala's object model is that every operation is the invocation of a method. For example, `x + y` is syntactic sugar for x.+(y); `x` is the receiver object, `+` is a method defined in `x`, and `y` is the method's argument. In fact, Scala desugars every identifier between two expressions as a method call of the first expression. Unlike constructors in Java, Scala constructors just come from the class name. There is no need for a separate class constructor inside the class definition. When a class is instantiated with its constructor, the whole body of the class is executed.

C. *Pattern Matching in an Object-Oriented Setting*

Unlike Java and other object-oriented programming languages, Scala implements pattern matching. That is, Scala provides the programmer with a natural and functional-like mechanism for "creating structured data representations similar to algebraic data types and a decomposition mechanism based on pattern matching."

Since Scala is an object-oriented programming language, it does not have algebraic data types. Instead, Scala creates structured data representations through the **`case`** modifier. If **case** precedes the definition of a class, a factory method with the same arguments as the primary class constructor is automatically defined. For example, in Figure 4, since the **Num** and **`Plus`** classes are defined with the **`case`** modifier, we can define an anonymous **Num** object without using the **new** keyword. As the reader can see, factory methods are very similar in structure to the constructors of algebraic data types. In fact, factory methods serve the same purpose as constructors when pattern matching.

Scala's pattern matching expressions can decompose the factory method constructors as patterns. The syntax for pattern matching expressions is

$$\texttt{x}\ \textbf{case}\ \{\ \textbf{case}\ p_1 => e_1\ \textbf{case}\ p_1 => e_1\ \ldots\ \}$$

This syntax matches the object `x` against the patterns $p_1, p_2, \ldots$ in order. Each p_1 is of the form $FactoryMethod(x_1, x_2, , x_n)$, where $FactoryMethod$ refers to the factory method constructors discussed previously. When a match p_i is found, then e_i is executed. For example, in Figure 5, the `eval` function matches `term` against `Num(x)` and `Plus(left, right)` (the constructors are from Figure 4).

Note: unlike Java, writing a simple language interpreter in Scala is almost as easy as writing an interpreter in SML. Examples of pattern matching being used for an interpreter can be seen in the calculator program we wrote.

D. *Views*

Views are used to implictly convert one type to another. A view is implemented with a method that takes in an arguemnt of one type and returns an object of another type. The only difference between a view method and a normal method is that view methods require the `implicit` modifier, which goes before the method definition. This modifier allows the Scala compiler to know that it is the implcit conversion method when converting from one type to another. Scala implictly applies a view to an expression, e of type T, when one of the following cases occur:

- The expected type of e is not of type T.
- A member selected from e is not a member of T.

For example, in Figure 7, `listToSet` is the view that converts `GenList[T]` to `Set[T]`. The compiler inserts applications of the view onto `xs`.

IV. Engagement: Peano Arithmetic Calculator

V. Discussion

A. *Current Status of Work*

After 15 years, Scala has been regularly updated and is currently at stable release version 2.12.8. Over this period, the core principles of Scala have remained the same. Scala has achieved widespread adoption in the industry with companies such as Twitter and Apple utilizing the language. Moreover, Scala has both a large academic and non-academic user base. The language is often cited or used in computer science research. In addition, Scala's user community is thriving. there exist chatrooms, subreddits, and research conferences devoted entirely to Scala. Numerous libraries, tutorials, and guides are run and maintained by the community. The main community page can be found at: https://www.scala-lang.org/community/. A detailed, updated, and easy to read documentation can be found at: https://docs.scala-lang.org/. Finally, there are dedicated installers/installation guides for all operating systems at: https://www.scala-lang.org/download/.

References

[1] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the Scala programming language," Tech. Rep., 2004.

[2] M. Odersky, V. Cremet, C. Röckl, and M. Zenger, "A Nominal Theory of Objects with Dependent Types," in *ECOOP 2003 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, L. Cardelli, Ed. Springer Berlin Heidelberg, 2003, pp. 201–224.

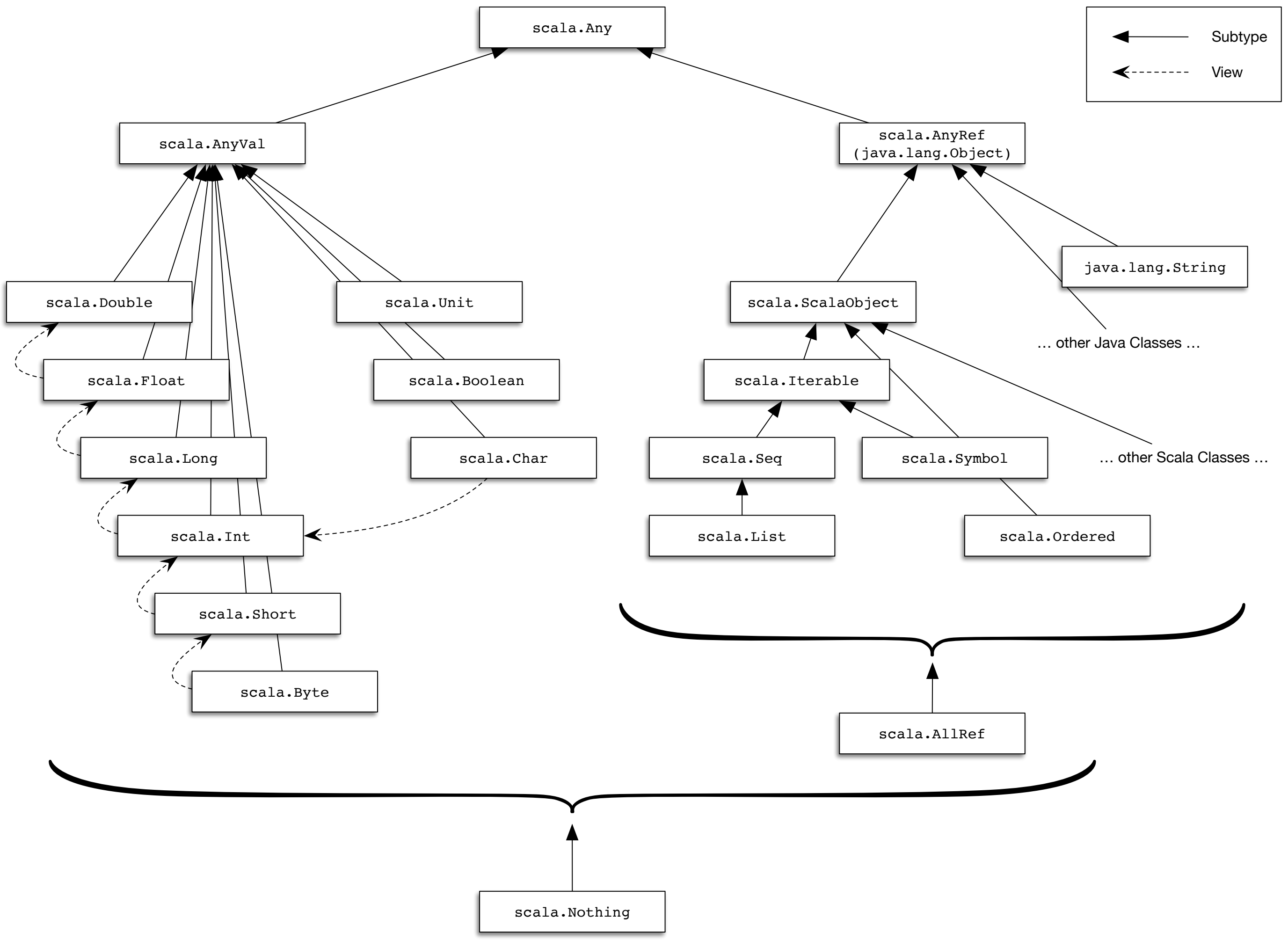

Fig. 4: A visual representation of Scala's class hierarchy

```
abstract class Term
case class Num(x: int) extends Term
case class Plus(left: Term, right: Term) extends Term
```

Fig. 5: example

```
object Interpreter {
  def eval(term: Term): int = term match {
    case Num(x) => x
    case Plus(left, right) => eval(left) + eval(right)
  }
}
```

Fig. 6

```
package Set

trait Set[T]{
  def include(x: T): Set[T]
  def contains(x: T): boolean
}

implicit def listToSet[T](xs: GenList[T]): Set[T] =
  new Set[T] {
    def include(x: T): Set[T] =
      xs prepend x
    def contains (x: T): boolean =
      !isEmpty && (xs.head == x || (xs.tail contains x))
  }

/*******************************************************/
//Assume that xs is a value of type GenList[T]
val s: Set[T] = xs;
xs contains x

//Compiler inserts applications of the view =>

val s: Set[T] = listToSet(xs);
listToSet(xs) contains x;
```

Fig. 7: An example of views and implcit conversions.

```
trait Ordering[T] {
  def leq (x: T, y: T): Boolean; // abstract method
  def geq (x: T, y: T): Boolean =  leq(y, x)
  def gt  (x: T, y: T): Boolean = !leq(x, y)
  def lt  (x: T, y: T): Boolean = !leq(y, x)
  def eq  (x: T, y: T): Boolean =  leq(x, y) &&  leq(y, x)
  def neq (x: T, y: T): Boolean = !leq(x, y) || !leq(y, x)
}
```

Fig. 8: An ordering relation in Scala. It is important to distinguish between an ordered type and an ordering on that type, of which there can be arbitrarily many. `Ordering[T]` represents the latter.