Scala - A Powerful and Scalable Function-Objective Programming Language

Troy Hu and Benjamin Killeen

***Abstract*—Insert abstract here.**

I. Introduction

During the mid 2000s, progress in developing component software was slow, which the researchers believed to be because existing languages had little support for component abstraction and composition. Some examples include statically typed languages such as Java and C#. In response to this lack of progress, Odersky et al. designed and implemented Scala in order to develop better language support for component software. As the researchers put it, components "are simply software parts which are used in some way by larger parts or whole applications. Components can take many forms; they can be modules, classes, libraries, frameworks, processes, or web services." At the same time, they envisioned their language as eventually being widely adopted.

The researchers achieved their goals by focusing on two areas:

- Developing scalable mechanisms for abstraction, composition, and decomposition. Through this focus, they made their language more scalable than existing languages such as Java. I.e., small and large components of component software can be expressed with the same concepts.
- Combining functional programming and object-oriented programming. Odersky et al. decided to combine these two programming paradigms in order to prove better scalable support for software components.

We now describe on a high-level how Scala achieves its goals of providing good support for component software and encouraging mass adoption.

A. High-Level Design

In order to encourage adoption by software developers, Scala's syntax is very similar to Java's and C#'s. In fact, Scala is able to work and interact with components coded in Java and C#. However, remember that Java and C# are suboptimal languages for supporting component software. Thus, in order to improve component support over Java and C#, Scala discards or modifies some their existing conventions. For example, Figure 1 demonstrates a simple program (CITE THIS) that prints out options provided in the command line.

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 III). 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.

```
// 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[i].substring(1))
  }
}

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

Fig. 1. Figure 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. (CITE THIS IMAGE FROM OVERVIEW PAPER.)

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.

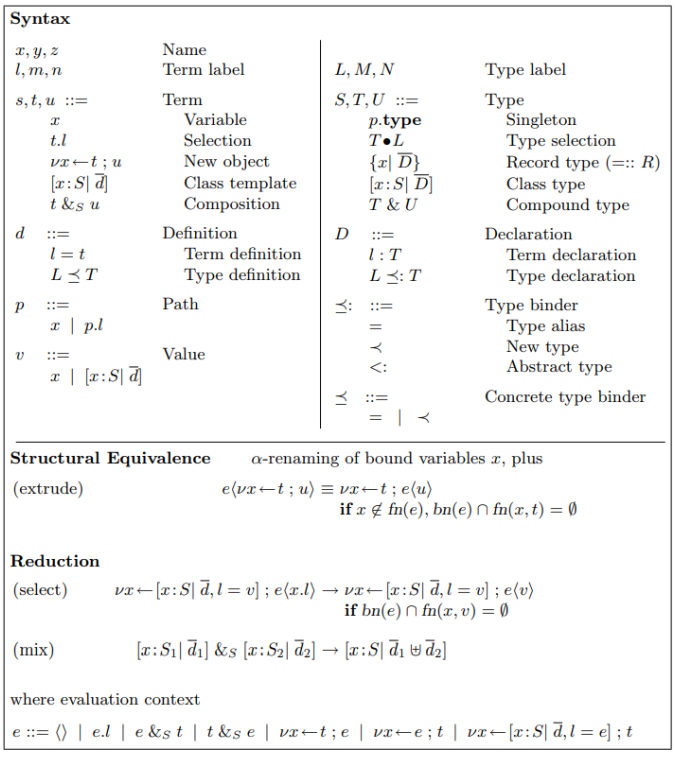

Fig. 2. The syntax of the νObj calculus

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

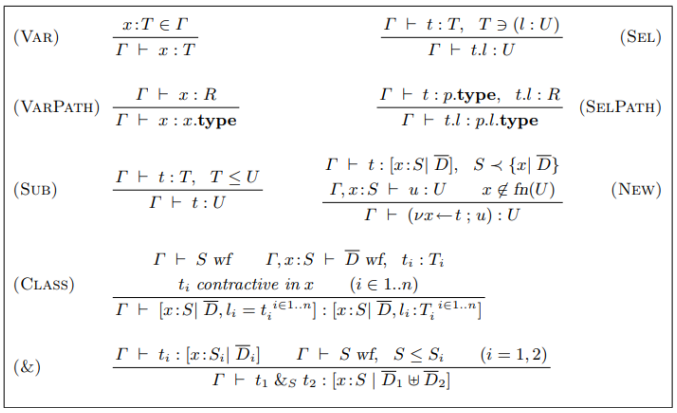

Fig. 3. The typing rules of the νObj calculus

```
trait Nat {

}

object Zero extends Nat {

}
```

Fig. 4. An example outlining Scala classes.

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. 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 $p_1, p_2, ...$ 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).

```
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.

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.

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/.