Scala for Real-Time Systems?

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
masca@dtu.dk

ABSTRACT

Java served well as a general-purpose language. However, during its two decades of constant change it has gotten some weight and legacy in the language syntax and the libraries. Furthermore, Java's success for real-time systems is mediocre.

Scala is a modern object-oriented and functional language with interesting new features. Although a new language, it executes on a Java virtual machine, reusing that technology. This paper explores Scala as language for future real-time systems.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Language Classifications —*Multiparadigm languages*

Keywords

Real-time systems, Scala, real-time Java

1. INTRODUCTION

Java has served now for two decades as a successful general purpose programming language. Like many "modern" programming languages it evolves and changes over time. However, these changes have to be introduced gradually to not disrupt current use. Furthermore, two decades of changes result in quite some legacy in the language and the library.

For more radical changes a clean cut is needed with a new language definition. Scala is such a new language [7]. Scala is a new language, but keeps the successful runtime system of Java, the Java virtual machine (JVM).

Java has been in widespread use and many domains and niches have been explored with Java. For example, Java was the main programming language for mobile phones, before the arrival of smartphones. Java has been, and still is, considered as language for real-time systems [4] and safety-critical systems [6]. However, we are still waiting on the success stories of Java for real-time systems. Maybe it is now time to reconsider new languages for future real-time systems.

This paper explores possibilities of use of Scala for real-time systems. It looks at new features from a real-time systems programmer's point of view.

Scala's aim is to be a scalable language. It can be used for scripting backed up by a very rich library. But Scala is also used for vary large concurrent systems such as Twitter.

Scala supports a smooth migration from Java to Scala. It executes on the Java virtual machine and Scala code can access Java methods, fields and can even inherit from Java classes. Scala reuses types from Java (e.g., `String`), but also dresses those types up with additional functionality.

Modern scripting languages with dynamic typing, e.g., Python, are becoming quite popular. The argument is that dynamic typing and the type inference at runtime results in more concise code. However, static typing, as in Java and Scala, provides type checks at the compile time and not during unit testing. To circumvent the boilerplate of Java, Scala provides a smart compiler that very often can infer the type automatically, as shown in the following example:

```
  val x = sqrt(10)
```

The above code avoids any type information as the compiler can infer the correct type of `x` from the return value of `sqrt()`. Note also that there are no semicolons needed.

Simple classes with private members and a single constructor can be defined in a single line of code:

```
class Simple(a: Int, s: String)
```

Adding functional programming to an object-oriented language rises the level of abstraction and therefore Scala is a high-level language. The notion of higher-level sequences, as a Scala `String` is, can provide simple control abstractions that take a function as parameter to be applied to this sequence.

Scala is statically typed. The decision between using a dynamic typed language or a static typed one for general purpose programming might be a matter of taste and preferences. However, in real-time and safety-critical applications we want to have as much support from the compiler for checks as possible. With a static type

JTRES '15, October 07-08, 2015, Paris, France
Copyright 2015 ACM 978-978-1-4503-3644-4/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2822304.2822313.

system in the language the compiler can verify the absence of type related runtime errors. Static types also improve the documentation of the code—no guessing what type a function might return, depending on types of parameters.

This paper is organized in 7 sections: Section 2 discusses the object oriented nature of Scala and explores with a tiny example how to build safe unit objects. Section 3 discusses the functional side of Scala. Section 4 presents actors, the Scala notion for future concurrent applications. Section 5 sketches the usage of Scala on top of the RTSJ [4]. Section 6 discusses the notion of domain specific languages that are supported by Scala. Section 7 concludes.

2. SCALA IS OBJECT ORIENTED

Scala builds on the success of object-oriented languages such as Java (and to some extent C++) and is object oriented. Compared to Java it is a "clean" object-oriented language. That means there are no non-objects such as primitive types. For example, integer values are of type `Int` and are full types. However, for performance reasons the Scala compiler maps those integer types to primitive `int` types of the JVM.

Furthermore, Scala can use methods with two parameters in infix notation and allows to use common operators as method names. Therefore, the following addition of two integer values

```
val a = 1
val b = 3
val c = a + b
```

is in reality a method invocation (`+()`) on an `Int` object:

```
val c = a.+(b)
```

This allows for definition of new types that feel like being part of the language. A classic problem in embedded systems, mixing different unit types, has led in 1999 to the loss of NASAs $ 125 million Mars orbiter. Lockheed Martin used English units while NASA itself used the metric system. Therefore, JPL looked into Java (and real-time Java) to provide a safe abstraction of *units* [3].

In Scala a new type to represent length values supporting different units can be defined and used like "normal" numbers. Figure 1 shows a sketch of how such a unit-safe class can be constructed that would have saved NASA $ 125 million.

```
object Main extends App {

  val a = new Length(1, "m")
  val b = new Length(2, "ft")
  val c = a + b
  println(c)
}


class Length(length: Double, unit: String) {

  val m = if (unit.equals("m"))
    length else length/3.28084

  def +(v: Length): Length =
    new Length(m + v.m, "m")

  override def toString(): String =
    m.toString + " " + unit
}
```

Figure 1: A simple `Unit` class to represent length with a unit.

This example class uses immutable values instead of variables, a style preferred for a functional style of programming. Immutable values further improve safety of the types. An object that extends `App` is an executable program where the code of the constructor is executed. We define two values of our new `Length` type using meter and feet. Adding those two numbers is safe and the result of 1.6096 m is printed to the console. The class `Length` is sketched with a constructor that converts all units to metric units and an addition method named '+'.

Note how concise the code is. Just by having parameters in the class definition a default constructor is defined. Code within the class definition, here the definition of `val m`, is already part of this constructor. A short function, such as `+()` and `toString()` in this example, can be defined in a single line without any clutter. Types can often be inferred by the compiler and need not be declared (in this example for value `m`). The type of the add function could be omitted as well, as the compiler can infer it, but it is recommended programming style to declare the return type of a function as part of the documentation.

3. SCALA IS A FUNCTIONAL LANGUAGE

Scala is also a functional language. Functional programming has gained popularity with the rise of multiprocessor and multithreaded programming. Functional programming does not mutate state and is therefore easier for parallel code.

Real-time garbage collectors are now standard in every real-time Java runtime [2, 8, 12]. When the allocation rate and the live-time of objects is known, a real-time garbage collector can be scheduled to cleanup the heap before the application runs out of memory [10].

Calculating maximum allocation for a function is similar to worst-case execution time analysis [9], but considerably easier as no pipeline analysis, global cache analysis, etc. needs to be considered. However, the hard part to derive correct scheduling of the collector is the knowledge of how long objects are live. If objects are used for communication between threads, i.e., a shared state, the live-time of objects would need a intra-thread analysis.

Here comes functional style programming as a rescue. Function code has no side effects and therefore no created objects are shared between threads. This greatly simplifies the calculation of the maximum live time of objects. The temporary created objects are only live during function execution. This property also fits nicely with the scoped memory model of the RTSJ. A function is executed within a temporary scope and after function return exiting the scope collects all garbage.

However, it has to be noted that it might not be a trivial problem to statically analyze how much memory is allocated within a function. As with loop bounds, bounds on the recursion depth need to be known statically to bound not only the execution time, but also the memory allocation.

```
case object Fire
case object Ok
case object NotOk
case object Stop

class Trigger(handler: Actor) extends Actor {
  def act() {
    handler ! Fire
    while (true) {
      receive {
        case Ok =>
          handler ! Fire
        case NotOk =>
          handler ! Stop
          exit()
      }
    }
  }
}

class Handler extends Actor {
  def act() {
    var cnt = 0
    while (true) {
      receive {
        case Fire =>
          if (cnt % 1000 == 0)
            Console.println("Handler: fire count " + cnt)
          if (cnt < 10000)
            sender ! Ok
          else
            sender ! NotOk
          cnt = cnt + 1
        case Stop =>
          Console.println("Handler: stop")
          exit()
      }
    }
  }
}
```

Figure 2: Two simple actors communicating via messages.

4. ACTORS FOR CONCURRENCY

Scala contains the notion of actors to describe concurrency. An actor provides an easier to use concurrency model than threads and shared state. Scala's actors do not share state, but use message-passing for communication. Using message passing helps to avoid race conditions.

Furthermore, access to shared state that resides in main memory (and then in some shared cache) becomes quickly the performance bottleneck on multicore processors. Therefore, we need to move towards message passing that can be easier supported with a better scalable network-on-chip [11]. Therefore, the actor abstraction fits very well for future multicore processors.

Figure 2 shows two actors interacting by exchanging simple messages. Actor `Trigger` sends `Fire` messages to the `Handler` actor, which in turn sends `Ok` or `NotOk` messages back. This example exchanges 10000 messages as fast as possible, but for a real-time application actor `Trigger` would be a periodic thread with calls to `waitForNextPeriod`.

The actor library is basically supporting a Kahn process network [5]. A Kahn process network is a network of processes that use first-in first-out channels between processes with unbounded capacity. Therefore, for real-time systems we need to bound the maximum number of outstanding messages to have a bounded memory consumption for the message buffers.

5. SCALA AND THE RTSJ

Scala can coexist with Java code. As Scala is the richer language using Java classes from Scala is easy, but it is not always straightforward to use Scala classes from Java. In fact Scala makes heavy use of the Java standard library. For example, Scala's string type is Java's `java.lang.String` with enhancements through a `StringLike` trait.

Therefore, it should be possible to use RTSJ classes and threads from a Scala program. Whether the Scala library behaves well with the scoped memory model of RTSJ is a similar issue as using Java standard library. Actually the functional part of Scala should be more scope friendly as no state is shared and the allocated garbage short lived.

We plan to explore Scala on top of the RTSJ implementation of aicas [13]. As Scala needs a full Java runtime and library we see no option on how Scala could be used on top of safety-critical Java [6]. Maybe the other way around is an option: have a restricted Scala (library) for safety-critical systems.

```
periodic task (10) {
  // here is the periodic work
}
```

Figure 3: A simple and lightweight DSL for real-time programming.

6. DOMAIN SPECIFIC LANGUAGES

Domain specific languages (DSL), e.g., a language for real-time systems, are usually hard to establish as a lot of infrastructure, i.e., compiler and runtime, needs to be developed and maintained. Scala might be a door opener for a real-time domain language. Scala has the power to build a DSL within the Scala language. The DSL can simply be implemented as a Scala library. One example is Scala's own actor framework. It feels like a language extension, but only uses Scala's powerful language possibilities. Another example shows Scala in a very different domain: Chisel is a hardware description language [1].

These two DSL examples show that the spectrum of possible DSLs is large. A future direction of research is to define a DSL, or even a family of DSLs, for future real-time and safety-critical systems.

The RTSJ has become, similar to Java, a heavyweight specification (e.g., Draft 19 of RTSJ is about 800 pages[1]). The current version of the safety-critical Java specification [6] with currently 830 pages is probably even worse.[2] This is clearly not what constitutes a brief and concise specification for small safety-critical systems that need certification.

Maybe the definition of a DSL for real-time Java or for safety-critical Java can change the direction of this trend and provide a lightweight language and API. Figure 3 shows a very short example how a real-time DSL could be used to define a periodic task. The task construct has a single parameter, the period in milliseconds (we can add versions for different time requirements with a more elaborated syntax). The code for the periodic task follows immediately between curly braces.

As future work we want to investigate the definition and implementation of a real-time DSL. We will explore this DSL on top of a RTSJ runtime, e.g., the JamaicaVM [13].

7. CONCLUSION

Java has over the last 20 years of existence collected some garbage and legacy. However, the Java virtual machine is a very successful platform for code execution. Scala offers a clean cut from the language perspective to include modern programming idioms, but using the Java virtual machine as execution platform. In this paper we have explored some features of Scala that might be interesting from the real-time programming perspective. The pure object-oriented approach of Scala can help to write safer programs. The functional aspect of Scala can help to write parallel code and combined with the actor model for concurrency this may scale well for future multicore processors.

We assume Scala can be used as an application language to execute on top of a real-time virtual machine with the RTSJ library. However, as Scala supports the definition of domain specific languages we envision such as domain specific language for real-time systems that might delegate to the RTSJ for the implementation.

This paper is a starting point for discussion and further research. There are many topics around Scala that can be explored by the real-time, object-oriented real-time, and real-time Java research community. We hope that Scala will appear as a topic in future JTRES events.

[1]Available at `https://www.aicas.com/cms/en/rtsj`

[2]Available at `https://github.com/scj-devel/doc`

8. REFERENCES

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: constructing hardware in a scala embedded language. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In R. Meersman and Z. Tari, editors, *OTM Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 466–478. Springer, 2003.

[3] E. G. Benowitz and A. F. Niessner. Experiences in adopting real-time java for flight-like software. In R. Meersman and Z. Tari, editors, *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops, OTM Confederated International Workshops, HCI-SWWA, IPW, JTRES, WORM, WMS, and WRSM 2003, Catania, Sicily, Italy, November 3-7, 2003, Proceedings*, volume 2889 of *Lecture Notes in Computer Science*, pages 490–496. Springer, 2003.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[5] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., Aug. 1974.

[6] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Vitek, and A. Wellings. Safety-critical Java technology specification, draft, 2014.

[7] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Inc, 2008.

[8] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 159–172, New York, NY, USA, 2007. ACM.

[9] W. Puffitsch, B. Huber, and M. Schoeberl. Worst-case analysis of heap allocations. In *Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2010)*, 2010.

[10] M. Schoeberl. Scheduling of hard real-time garbage collection. *Real-Time Systems*, 45(3):176–213, 2010.

[11] M. Schoeberl, R. B. Sørensen, and J. Sparsø. Models of communication for multicore processors. In *Proceedings of the 11th Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2015)*, pages 44–51, Aukland, New Zealand, April 2015. IEEE.

[12] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the 2000 international conference on Compilers, architecture,*

and synthesis for embedded systems (CASES 2000), pages 9–17, New York, NY, USA, 2000. ACM.

[13] F. Siebert and A. Walter. Deterministic execution of Java's primitive bytecode operations. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01): April 23–24, 2001, Monterey, California, USA. Berkeley, CA*, pages 141–152. USENIX, 2001.