How to Grow a Language for "Big" Applications

Philipp Haller
@philippkhaller

Joint work with Heather Miller
Overview

• Motivation

• Short overview of Scala

• Implicits

• Experimental macro extension

• Symbiosis of implicits and macros

• “Big” applications
Motivation

• Context: reactive programming
  • Scalability, efficiency, resiliency
  • Productivity and predictability

• What are good tools for PL research?
  • How to prototype and evaluate language features effectively?
Short Overview of Scala

• A modular language

• Integrates object-oriented and functional language features

• Strong static typing

• Compilation targets: JVM (stable), ECMAScript (experimental)
class DataEvent(
    i: String, t: Int, m: String)
extends Event(i) {

    def id: String = i
    def timestamp: Int = t
    def message: String = m
}
class DataEvent(val id: String, 
    val timestamp: Int, 
    val message: String) 
extends Event(id) {
}

Classes (2)
Case Classes

case class DataEvent(id: String,
  timestamp: Int,
  message: String)
  extends Event(id) {
}

Adds:
  • *Pattern matching*
  • *Structural equality*
  • *Getters* for constructor parameters
  • ...
Singleton Objects

object DataEvent {
    private var currId = 0

    private def nextId() = {
        currId += 1
        s"$currId"
    }

    def apply(message: String): DataEvent = 
        new DataEvent(nextId(), now, message)
}

val evt = DataEvent.apply("a message")
trait Ordering[T] extends Comparator[T] {

    def compare(x: T, y: T): Int // abstract

    def lt(x: T, y: T): Boolean = compare(x, y) < 0

    def equiv(x: T, y: T): Boolean = compare(x, y) == 0

    ...
}

Mix-in composition of multiple traits
trait IntOrdering extends Ordering[Int] {
    def compare(x: Int, y: Int): Int =
        if (x < y) -1
        else if (x == y) 0
        else 1
}
Implicits in Scala

• A language mechanism for *generic programming*

• Related to *type classes* and the *C++0x concepts* proposal

• Main principle: *implicit instantiation of implementations* of parameter types

“a programming style that decouples algorithms from the concrete types on which they operate”
Implicits: Example

/** Quickly sort an array of items
 * with an implicit Ordering.
 */
def quickSort[T](a: Array[T])
  (implicit ord: Ordering[T]): Unit = ..

Calling quickSort requires an implicit of type Ordering[T] in scope.
Defining an Implicit Ordering

object DataEventImplicits {

  /** Order events by their time stamp. */
  implicit val evtOrdering: Ordering[DataEvent] =
    new Ordering[DataEvent] {
      def compare(x: DataEvent, y: DataEvent) =
        if (x.timestamp < y.timestamp) -1
        else if (x.timestamp == y.timestamp) 0
        else 1
    }

}
Sorting Events

Brings implicit of type Ordering[DataEvent] into scope

```scala
import scala.util.Sorting._
import DataEventImplicits._

object Main {
  def main(args: Array[String]): Unit = {
    val events: Array[DataEvent] = // read in events
    quickSort(events)
  }
}
```
import scala.util.Sorting._
import DataEventImplicits._

object Main {
  def main(args: Array[String]): Unit = {
    val events: Array[DataEvent] = // read in events
    
    quickSort(events)(evtOrdering)
  }
}

Sorting Events: Expansion
How are Implicit Arguments Provided?

**Eligible arguments:**

1. Identifiers that are *accessible at the invocation site without a prefix* and that denote an implicit definition

2. Implicit members of some object that belongs to the *implicit scope* of the implicit parameter’s type $T$
Eligible Implicits

Members of object Ordering:
- implicit object Int extends IntOrdering
- implicit object Long extends LongOrdering

Identifiers accessible without prefix:
- implicit val evtOrdering

Expected type:
- Ordering[DataEvent]

```scala
import scala.util.Sorting._
import DataEventImplicits._

val events: Array[DataEvent] = // read in events

quickSort(events)(???)
```
implicit def makeTupleOrdering[A, B](implicit orda: Ordering[A], ordb: Ordering[B]): Ordering[(A, B)] =
new Ordering[(A, B)] {
    def compare(x: (A, B), y: (A, B)): Int = {
        val resa = orda.compare(x._1, y._1)
        if (resa != 0) resa
        else ordb.compare(x._2, y._2)
    }
}
Implicit Scope

From the Scala Language Specification:

- The implicit scope of a type $T$ consists of all companion objects of classes that are associated with the implicit parameter’s type.

- A class $C$ is associated with a type $T$, if it is a base class of some part of $T$. 
Parts of a Type

The parts of a type $T$ are:

- If $T$ is a compound type $T_1$ with ... with $T_n$, the union of the parts of $T_1$, ..., $T_n$, as well as $T$ itself,

- If $T$ is a parameterized type $S[T_1, ..., T_n]$, the union of the parts of $S$ and $T_1, ..., T_n$,

- ..

- in all other cases, just $T$ itself.
Parts: Example

- The parts of type `Ordering[DataEvent]` is the union of the parts of `Ordering` and the parts of `DataEvent`.

- Therefore, class `DataEvent` is associated with type `Ordering[DataEvent]`.

- Therefore, the companion object of class `DataEvent` is in the `implicit scope` of type `Ordering[DataEvent]`.
Avoiding Explicit Imports

• Recall: “implicit members of some object that belongs to the implicit scope of the implicit parameter’s type $T$ are eligible as arguments for implicit parameters”

• Therefore, implicit members of the $DataEvent$ object are eligible as arguments for implicit parameters of type $Ordering[DataEvent]$
Defining an Implicit Ordering

```scala
object DataEvent {
  /** Order events by their time stamp. */
  implicit val evtOrdering: Ordering[DataEvent] = 
    new Ordering[DataEvent] {
      def compare(x: DataEvent, y: DataEvent) =
        if (x.timestamp < y.timestamp) -1
        else if (x.timestamp == y.timestamp) 0
        else 1
    }
}
```
import scala.util.Sorting

object Main {
  def main(args: Array[String]): Unit = {
    val events: Array[DataEvent] = 
      // read in events
      Sorting.quickSort(events)
  }
}
Applications of Implicits

- Flexible default arguments
- Type class pattern
- Type-level programming
Foundations

- Implicits might seem to be complex
- Foundations not well-established
- Promising candidate:

The Implicit Calculus: A New Foundation for Generic Programming

BRUNO C. D. S. OLIVEIRA, The University of Hong Kong
TOM SCHRIJVERS, Ghent University
WONTAE CHOI, Seoul National University
WONCHAN LEE, Seoul National University
KWANGKEUN YI, Seoul National University
PHILIP WADLER, University of Edinburgh
Overview

- Motivation
- Short overview of Scala
- Implicits
- *Experimental macro extension*
  - Symbiosis of implicits and macros
  - “Big” applications
Macros

- Experimental language feature
- Enables new constructs implemented as macros
  - No full-blown compiler plug-in required
- Well-suited for prototypes that inform language evolution
- New macro system in the works (project scala.meta)
Macros: High-Level Summary

- Macros = methods that operate on and return typed expression trees

- Macro invocations are evaluated at compile time, the resulting expression trees are inlined, and then type-checked

```scala
def assert(x: Boolean, msg: String): Unit = macro assert_impl

def assert_impl(c: Context)(x: c.Expr[Boolean], msg: c.Expr[String]): c.Expr[Unit] = ...
```
Implementing Macros

• Main issue: how to consume and produce expression trees?

• Reflection API

• Quasiquotes

```scala
val cond: c.Expr[Boolean] = ...
q"if ($cond) $thenb else $elseb"
```
Overview

- Motivation
- Short overview of Scala
- Implicits
- Experimental macro extension
- *Symbiosis of implicits and macros*
- “Big” applications
Implicit Macros

• What are they? Implicits that are applied at compile time

```scala
implicit def genPickler[T]: Pickler[T] = macro ..
```

• genPickler[T] is invoked whenever an implicit of type Pickler[T] is required

• The invocation is expanded at compile time!
Recursive Implicit Macros

The genPickler[T] macro can generate invocations of itself (that are also evaluated at compile time)

```scala
val v: $fldType = obj.fldName
val fldp = implicitly[Pickler[$fldType]]
fldp.pickle(v)
```

May be expanded to genPickler[$fldType]
Bottom Line

• Combination of implicit and macros enable powerful *generic abstractions*

• New possibilities for datatype-generic programming
Overview

• Motivation

• Short overview of Scala

• Implicits

• Experimental macro extension

• Symbiosis of implicits and macros

• “Big” applications
Big Applications

- Big Data -> Big Application
- Big Concurrency -> Big Application
# Big Data Analytics

- Popular datatypes:

<table>
<thead>
<tr>
<th></th>
<th>primitives/primitive arrays</th>
<th>value-like types</th>
<th>collections</th>
<th>case classes</th>
<th>type descriptor</th>
<th>generics</th>
<th>subtyping polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoTrellis (Akka)</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Evactor (Akka)</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Spark</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Storm</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>N/A</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Twitter Chill</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

**Legend:**  ●: Heavy Use  ○: Light Use  ○: No Use
Implicits + Macros

Implicits and macros enable generic abstractions that are fast:
Summary

- Implicits: a lightweight language mechanism with solid foundations

- **Combining implicits with a simple macro system enables fast generic abstractions**

- Open questions:
  - How to enable writing generic abstractions on a higher level?
  - Foundations for the combination of implicits and macros?