Scala Macros

Eugene Burmako, Denys Shabalin
Martin Odersky

Macros are functions that are called by the compiler during compilation. Within these functions the programmer has access to compiler APIs. For example, it is possible to generate, analyze and typecheck code.

Macros are shipping with the official Scala compiler. Since 2.10.0 Scala includes macros enabled with `import language.experimental.macros`. Numerous commercial and research projects are already using macros.

Macros are good for code generation, static checking and domain-specific languages. Scenarios that traditionally involve writing and maintaining boilerplate can be addressed with macros in a concise and maintainable way.

Term generation

```
def specialized[T: ClassTag](code: => T) = macro ...
def createArray[T: ClassTag](size: Int, el: T) = {
  val a = new Array[T](size)
  specialized[T] {
    for (i <- 0 until size) a(i) = el
  }
  a
}
```

```
def createArray[T: ClassTag](size: Int, el: T) = {
  val a = new Array[T](size)
  {
    def body[@specialized T](el: T) {
      for (i <- 0 until size) a(i) = el
    }
    classTag[T] match {
      case ClassTag.Int => body(el.asInstanceOf[Int])
      ...
    }
  }
  a
}
```

- In Scala, macro calls look exactly like normal method calls, so users might not even notice that they use macros.
- Macro defs are also very similar to normal method defs, which means that they have typed signatures.
- A lot of features in Scala (getters/setters, pattern matching, for loops, interpolation, etc) are implemented with methods, therefore macros can empower them all!

Type generation

```
def h2db(connString: String): Any = macro ...
val db = h2db("jdbc:h2:coffees.h2.db")
```

```
val db = {
  trait Db {
    case class Coffee(name: String, price: Decimal)
    val Coffees: Table[Coffee] = ...
  }
  new Db {}
}
```

```
scala> val db = h2db("jdbc:h2:coffees.h2.db")
db: AnyRef {
  type Coffee { val name: String; val price: Decimal }
  val Coffees: Table[this.Coffee]
} = $anon$1...

scala> db.Coffees.all
res1: List[Db$1.this.Coffee] = List(Coffee(...))
```

- Thanks to Scala's support for first-class modules, the same mechanism that is used to generate terms can be used to generate types.
- There are caveats, but they can mostly be handled with a few more macros. Details can be found in our docs.
- In macro paradise, a compiler plugin for Scala, we're experimenting with other means of generating types and definitions.

Materialization

```
trait Showable[T] { def show(x: T): String }
def show[T](x: T)(implicit ev: Showable[T]) = ev.show(x)

implicit val intShowable = new Showable[Int] {
  def show(x: Int) = x.toString
}
show(42) // you write
show(42)(IntIsShowable) // you get
show("42") // doesn't compile: can't find implicit value

implicit def lstShowable[T](implicit s: Showable[T]) =
  new Showable[List[T]] {
    def show(x: List[T]) = x.map(s.show).mkString
  }

implicit def materialize[T]: Showable[T] = macro ...
show(person) // you write
show(person)(materialize[Person]) // you get
```

```
show(person)(new Showable[Person] { ... })
```

- Scala's implicits marry the worlds of terms and types by providing a type-directed facility of looking up terms in scope. Similar to Haskell's type classes, but more elaborate.
- Implicits can be provided by methods marked with the `implicit` modifier, and methods can be macros, which gives rise to automatic derivation of type class instances.
- But fun doesn't end here. Check out our documentation for information about fundep materializers!

Advanced type signatures

```
trait Request
case class Command(msg: String) extends Request

trait Reply
case object CommandSuccess extends Reply
case class CommandFailure(msg: String) extends Reply

abstract class ActorRef {
  def !(msg: Any): Unit
}

val actor = someActor()
actor ! Command // compiles, but fails at runtime

class ChannelRef[T](actor: ActorRef) {
  def <-!-[M](msg: M): Unit = macro ...
}

type Spec = (Request, Reply) :+: TNil
val actor = new ChannelRef[Spec](someActor())
actor <-!- Command // doesn't compile
```

- Akka actors are dynamically typed, i.e. the ! method takes Any, which loosens guarantees provided by Scala.
- Even in vanilla Scala one can implement session types for actors using type-level computations via implicits, but it's cumbersome and doesn't allow for good error messages.
- With macros, session types in Scala get to be practical, having a straightforward implementation and supporting fine-grained error messages.

Advanced static checks

```
var sender: ActorRef = _

def future[T](body: => T): Future[T] = ...
def handleActorRequest(data) = future {
  val result = transform(data)
  sender ! Response(result) // dangerous capture of a var
}
```

```
def future[T](body: Spore[T]): Future[T] = ...
def spore[T](body: => T): Spore[T] = macro ...
def handleActorRequest(data) = future(spore {
  val result = transform(data)
  sender ! Response(result) // doesn't compile
})
```

```
def future[T](body: Spore[T]): Future[T] = ...
implicit def sporify[T](body: => T): Spore[T] = macro ...
def handleActorRequest(data) = future {
  val result = transform(data)
  sender ! Response(result) // doesn't compile
}
```

- With macros it becomes possible to enforce invariants that cannot be expressed in Scala's type system.
- In this example, the `spore` macro encapsulates function literals and makes sure that they don't contain references to things that shouldn't be captured.
- Since macros are type-driven, it is possible to create an implicit conversion from a function type to a spore type that would apply checks automatically, without any user effort.

Typing the untypeable

```
scala> val x = "42"
x: String = 42

scala> "%d".format(x)
j.u.IllegalFormatConversionException: d != j.l.String
at j.u.Formatter$FormatSpecifier.failConversion...

scala> f"$x%d"
<console>:31: error: type mismatch;
 found   : String
 required: Int

implicit class Formatter(c: StringContext) {
  def f(args: Any*): String = macro ...
}
f"$x%d" // you write
StringContext("", "%d").f(x) // you get
```

```
val arg$1: Int = x // doesn't compile
"%d".format(arg$1)
```

- Even stringly-typed representation for data, which is typically scoffed at, can be given meaning thanks to macros.
- By parsing strings at compile-time programmers can guarantee that those strings are well-formed with respect to expected syntax and conform to domain-specific semantics.
- During expansion, macros can communicate with Scala's typechecker that provides APIs to calculate types, inspect existing definitions and even introduce new ones.

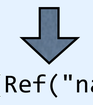
Language virtualization

```
// approach #1: external DSL with strings
val usersMatching = query[String, (Int, String)](
  "select id, name from users where name = ?")
usersMatching("John")

// approach #2: internal DSL with vanilla Scala
case class User(id: Column[Int], name: Column[String])
users.filter(_.name === "John")

// approach #3: language virtualization with macros
class Queryable[T](val query: Query) {
  def filter(p: T => Boolean): Queryable[T] = macro ...
  def toList: List[T] = query.translate.execute[T]
}

case class User(id: Int, name: String)
val ppl: Queryable[User] = ...
ppl.filter(_.name == "John")
```



```
Query(Filter(ppl, Equals(Ref("name"), Literal("John"))))
```

- Language virtualization is the first use case for Scala macros and the direct motivator for adding macros to the language.
- Macros have access to code snippets representing their arguments, so it becomes possible to analyze these snippets and then override the usual semantics of Scala for them.
- In particular, language virtualization with macros enables language-integrated queries, like C#'s LINQ, without the necessity to introduce additional language features.

Internal DSLs

```
val futureDOY: Future[Response] =
  WS.url("http://api.day-of-year/today").get
val futureDaysLeft: Future[Response] =
  WS.url("http://api.days-left/today").get
futureDOY.flatMap { doyResponse =>
  val dayOfYear = doyResponse.body
  futureDaysLeft.map { daysLeftResponse =>
    val daysLeft = daysLeftResponse.body
    Ok(s"$dayOfYear: $daysLeft days left!")
  }
}

// dealing with callback hell is quite painful
// luckily things can be reformulated in a simpler way

def async[T](body: => T): Future[T] = macro ...
def await[T](future: Future[T]): T = macro ...
async {
  val dayOfYear = await(futureDOY).body
  val daysLeft = await(futureDaysLeft).body
  Ok(s"$dayOfYear: $daysLeft days left!")
}
```

- Scala is well-known for its features that simplify writing domain-specific languages, and macros push this even further.
- Access to abstract syntax trees of the DSL and the ability to call into typechecker APIs add valuable tools into a DSL author's toolbox.
- In this example, the `async` macro performs a CPS-like transformation to emulate C#'s async, again without having to add extra language features.

External DSLs

```
scala> Query("""
     [ :find ?e ?age
       :in $ ?name
       :where [ ?e :person/name ?name
     ]""")
<console>:14: error: `]' expected but end of source found
     ]""")
      ^

scala> Query("""
     [ :find ?e ?n
       :in $ ?char
       :where  [ ?e :person/name ?n ]
               [ ?e person/character ?char ]
     ]""")
res0: TypedQueryAuto2[DatomicData, DatomicData,
(DatomicData, DatomicData)] = [ :find ?e ?n :in \$ ?
char :where ... ]
```

- Macros also provide facilities to deeply embed external domain-specific languages, such as HTML, SQL, JSON, etc.
- Without special language or tool support, programs view external DSLs as passive strings, which can be parsed and interpreted, but cannot communicate with the main program.
- Compile-time metaprogramming provides a way to animate such languages, making them able to analyze and possibly influence the enclosing program.

Authors heartily thank members of the Scala reflection group for amazing discussions, early adopters for fearless experiments with new macro flavors, colleagues from Laboratoire de Méthodes de Programmation at Ecole Polytechnique Fédérale de Lausanne for helpful advice, and the entire Scala community for adoption and feedback. This poster uses code snippets and ideas from open-source projects by Jason Zaugg, Miles Sabin, Travis Brown, Dominik Gruntz, Heather Miller, Philipp Haller, Nicolas Stucki, Vlad Ureche, Typesafe, Pellucid Analytics & Zenexity. 21 Nov 2013