Chapter 24. Metaprogramming: Macros and Reflection

Metaprogramming is programming that manipulates programs, rather than data. In some languages, the difference between *programming* and *metaprogramming* isn't all that significant. Lisp dialects, for example, use the same *S-expression* representation for code and data, a property called *homoiconicity*. So, manipulating code is straightforward and not uncommon. In statically typed languages like Java and Scala, metaprogramming is less common, but it's still useful for solving many design problems.

The word *reflection* is also sometimes used to mean metaprogramming in general. That is the sense of the term for the Scala reflection library. However, sometimes the term has the narrower meaning of runtime "introspection" of code with limited or no modifications.

In languages like Scala where code is compiled and then run, versus being interpreted "on the fly" like many dynamically typed languages, there is a distinction between compile-time and runtime metaprogramming. In compile-time metaprogramming, any invocations occur just before or during compilation. The classic C-language preprocessor is an example of processing that transforms the *source code* before it's compiled.

Scala's metaprogramming support happens at compile time using a *macro* facility. Macros work more like constrained compiler plug-ins, because they manipulate the *abstract syntax tree* (AST) produced from the parsed source code. Macros are invoked to manipulate the AST before the final compilation phases leading to byte-code generation.

The Java reflection library and Scala's expanded library offer runtime reflection.

Scala's reflection API, which includes the macro support, is the most rapidly evolving part of Scala. Because it's a fast-moving target, we'll focus on the most stable parts: runtime reflection and a macro tool called *quasiquotes*. However, we'll end with a full macro example using the current macro API.

A next-generation macro facility is being developed. The project is called *Scala Meta*. At the time of this writing, a preview release is forthcoming. You should look there for the latest information about macros as they will appear in a subsequent release of Scala. For the current macro implementation for Scala 2.10 and 2.11, see http://scalamacros.org and Macro Paradise, the incubator project for the current macro system.

We'll begin with some useful REPL tools for understanding the types of expressions, then explore runtime reflection, followed by quasiquotes with a final macro example.

Tools for Understanding Types

The REPL has a `:type` command for printing type information:

```
scala> if (true) false else 11.1
res0: AnyVal = false

scala> :type if (true) false else 11.1
AnyVal

scala> :type -v if (true) false else 11.1
// Type signature
AnyVal

// Internal Type
structure
TypeRef(TypeSymbol(abstract class AnyVal extends Any))
```

The `:type` command just shows the type. Usually the REPL echoes the type anyway. However, the `-v` (verbose) option also shows the "internal type structure." The `scala.reflect.api.Types.TypeRef` and `scala.reflect.api.Symbols.TypeSymbol` types are defined in the reflection API, which is now a separate library from the core standard library. The *Scaladocs* can be found at http://bit.ly/1wQkYDN.

Runtime Reflection

Whereas compile-time reflection is used for manipulating code, runtime reflection is used primarily to "tweak" language semantics (within limits) and to load code that isn't known at compile time, so-called *extreme late binding*.

For example, which instance to use for a particular feature might be specified dynamically through properties or command-line arguments. The reflection API is used to locate the corresponding types in the available byte code found on the `CLASSPATH`, and if found, construct instances. Tools like IDEs can use reflection to discover and load plug-ins. IDEs often use reflection to learn about code in projects and libraries, to support code completion, type checking, etc. Byte-code tools might use reflection to look for security vulnerabilities and other problems.

Reflecting on Types

You can use Java's reflection API, such as methods in `java.lang.Class`:

```
//
src/main/scala/progscala2/metaprogramming/reflect.sc

scala> import scala.language.existentials
import scala.language.existentials

scala> trait T[A] {
     |   val vT: A
     |   def mT = vT
     | }
defined trait T

scala> class C(foo: Int) extends T[String] {
     |   val vT = "T"
     |   val vC = "C"
     |   def mC = vC
     |
     |   class C2
     | }
defined class C

scala> val c = new C(3)
c: C = $anon$1@5a58e6a4

scala> val clazz = classOf[C]              // Scala method: classOf[C]
clazz: Class[C] = class C

scala> val clazz2 = c.getClass             // Method from java.lang.Object
clazz2: Class[_ <: C] = class $anon$1

       val name  = clazz.
scala> getName
name: String = C

scala> val methods = clazz.getMethods
methods: Array[java.lang.reflect.Method] =
  Array(public java.lang.String C.mC(), public java.lang.Object C.vT(), ...)

scala> val ctors = clazz.getConstructors
ctors: Array[java.lang.reflect.Constructor[_]] = Array(public C(int))

scala> val fields = clazz.getFields
fields: Array[java.lang.reflect.Field] = Array()

scala> val annos = clazz.getAnnotations
annos: Array[java.lang.annotation.Annotation] = Array()

scala> val parentInterfaces = clazz.getInterfaces
parentInterfaces: Array[Class[_]] = Array(interface T)

scala> val superClass = clazz.getSuperclass
superClass: Class[_ >: C] = class java.lang.Object

scala> val typeParams = clazz.getTypeParameters
typeParams: Array[java.lang.reflect.TypeVariable[Class[C]]] = Array()
```

These methods are only available on subtypes of `AnyRef`. Note that `getFields` does *not* appear to recognize the fields in `C` for Scala types!

`Predef` defines methods for testing whether an object matches a type and for casting an object to a type:

```
scala> c.isInstanceOf[String]
<console>:13: warning: fruitless type test: a value of type C cannot
  also be a String (the underlying of String)
              c.isInstanceOf[String]
                             ^
res0: Boolean = false

scala> c.isInstanceOf[C]
res1: Boolean = true

scala> c.asInstanceOf[T[AnyRef]]
res2: T[AnyRef] = C@499a497b
```

Java uses operators that are language keywords for these tasks. The Scala method names are deliberately verbose to discourage their use! Other language features, especially pattern matching, are better alternatives.

Class Tags, Type Tags, and Manifests

The core Scala 2.11 library has a small reflection API, while the more advanced reflection features are in the separate library. Let's investigate `ClassTag` in the core library, which is a tool for retaining some information that is otherwise lost to *type erasure*, the "feature" of the JVM where it doesn't retain the values used for type parameters when instantiating parameterized types.

We saw in More on Type Matching that erasure prevents us from pattern matching on the types used as type parameters in parameterized types. We used an ugly workaround then, where we first matched on the collection and then on the types within it. We also can't overload methods where the only difference between them is the type parameter for a parameterized type used in the signatures.

`ClassTag` provides a better workaround that we'll now examine:

```
// src/main/scala/progscala2/metaprogramming/match-types.sc
import scala.reflect.ClassTag

def useClassTag[T : ClassTag](seq: Seq[T]): String = seq match { // ❶
  case Nil => "Nothing"
  case head +: _ => implicitly(seq.head).getClass.toString           //
❷
}

def check(seq: Seq[_]): String =
// ❸
  s"Seq: ${useClassTag(seq)}"

Seq(Seq(5.5,5.6,5.7), Seq("a", "b"),
// ❹
    Seq(1, "two", 3.14), Nil) foreach {
                             "%20s:
  case seq: Seq[_] => println(%s"        .format(seq, check(seq)))
                             "%20s:
  case x           => println(%s"        .format(x, "unknown!"))
}
```

❶

Use a context bound with `ClassTag`.

❷

If we have a nonempty list, use `implicitly` to get the implicit `ClassTag` instance and call its `apply` method on `seq.head` to determine its type. The flaw of this method is that it returns the first element's type, but if a sequence of mixed types is passed in, a *least upper bound* supertype is actually the correct type to return. We'll fix this bug shortly.

❸

A helper method to try both functions.

❹

Test the implementation with some examples.

The output is the following, where now we know the type of each sequence's elements:

```
List(5.5, 5.6, 5.7):  Seq: class java.lang.Double
         List(a, b):  Seq: class
java.lang.String
 List(1, two, 3.14):  Seq: class
java.lang.Integer
             List():  Seq: Nothing
```

However, as mentioned, it's not accurate for `Seq[Any]` with mixed elements, the second to last example.

The compiler exploits the type information it knows to construct the implicit `ClassTag`. However, when given previously constructed lists, the crucial type information is already lost. This is an issue if you're passing collections around and somewhere deep in the stack some method wants to use a ClassTag for introspection. You'll need to construct the collection *and* a corresponding ClassTag in the same scope, then pass them together somehow, perhaps through an implicit argument for the ClassTag in subsequent method calls. We'll come back to this issue a little later.

Hence, ClassTags can't "resurrect" type information from byte code, but they can be used to capture and exploit type information before it is erased.

ClassTag is actually a weaker version of scala.reflect.api.TypeTags#TypeTag, found in the separate API. The latter retains the full compile-time information (we'll use it shortly), whereas ClassTag only returns the runtime information. Finally, there is a scala.reflect.api.TypeTags#WeakTypeTag for abstract types. See the detailed descriptions in the Scala docs.

Note that there are older types in the reflect package called Manifests that were used for the same purpose before Scala 2.10 introduced TypeTag, ClassTag, etc. These types are being deprecated. You'll see them in older source code. However, you should use the newer features.

Another important usage for ClassTag is to construct Java Arrays of the correct AnyRef subtype. Here is an example adapted from the Scaladoc page for ClassTag:

```
//
src/main/scala/progscala2/metaprogramming/mkArray.sc
scala> import scala.reflect.ClassTag
import scala.reflect.ClassTag

scala> def mkArray[T : ClassTag](elems: T*) = Array[T](elems: _*)
mkArray: [T](elems: T*)(implicit evidence$1: scala.reflect.ClassTag[T])Array[T]

scala> mkArray(1, 2, 3)
res0: Array[Int] = Array(1, 2, 3)

scala> mkArray("one", "two", "three")
res1: Array[String] = Array(one, two, three)

scala> mkArray(1, "two", 3.14)
<console>:10: warning: a type was inferred to be `Any`;
  this may indicate a programming error.
              mkArray(1, "two", 3.14)
                      ^
res2: Array[Any] = Array(1, two, 3.14)
```

It uses the Array.apply method for AnyRefs, which has a second argument list with a single implicit ClassTag argument.

Scala's Advanced Runtime Reflection API

The rest of the reflection API supports richer runtime reflection as well as the compile-time macros. It includes types that represent abstract syntax trees and other contexts. It is distributed as a separate JAR file, for which we included a dependency in the sbt build. The full details of this API are described at in the Scala docs. We'll discuss the core ideas of this very large API and a few examples for a common task, runtime introspection of types:

```
// src/main/scala/progscala2/metaprogramming/match-type-tags.sc

import scala.reflect.runtime.universe._
// ❶

def toType2[T](t: T)(implicit tag: TypeTag[T]): Type = tag.tpe       //
❷
def toType[T : TypeTag](t: T): Type = typeOf[T]                      //
❸
```

❶

Import the definitions defined in the runtime “universe,” which is of type `scala.reflect.api.JavaUniverse`. It exposes the types reflecting language elements and convenience methods for the target platform.

❷

Use an implicit argument for a `TypeTag[T]`, then ask it for its type.

❸

More convenient alternative using a context bound. The `typeOf[T]` method is a shortcut for `implicitly[TypeTag[T]].tpe`.

Recall that `TypeTag` retains the full compile-time type information while `ClassTag` only retains the runtime type information.

Let’s try these methods with a few types:

```
scala> toType(1)
res1: reflect.runtime.universe.Type = Int

scala> toType(true)
res2: reflect.runtime.universe.Type = Boolean

scala> toType(Seq(1, true, 3.14))
<console>:12: warning: a type was inferred to be `AnyVal`;
  this may indicate a programming error.
              toType(Seq(1, true, 3.14))
                         ^
res3: reflect.runtime.universe.Type = Seq[AnyVal]

scala> toType((i: Int) => i.toString)
res4: reflect.runtime.universe.Type = Int => java.lang.String
```

Note that the types for the parameterized type parameters are correctly determined, fixing the bug we had in `useClassTag`. We’ll omit the `AnyVal` warnings from now on.

We can compare types for equality or parent-child relationships:

```
toType(1) =:= typeOf[AnyVal]
// false
toType(1) =:= toType(1)
// true
toType(1) =:= toType(true)
// false

toType(1) <:< typeOf[AnyVal]
// true
toType(1) <:< toType(1)
// true
toType(1) <:< toType(true)
// false

typeOf[Seq[Int]] =:= typeOf[Seq[Any]]
// false
typeOf[Seq[Int]] <:< typeOf[Seq[Any]]
// true
```

We've been calling the `tpe` method to get the `Type` from the `TypeTag`. You can also get the latter directory with the helper function `typeTag`:

```
typeTag[Int]
// reflect.runtime.universe.TypeTag[Int] =
TypeTag[Int]
                      // ...TypeTag[Seq[Int]] =
typeTag[Seq[Int]]     TypeTag[scala.Seq[Int]]
```

Recall in Functions Under the Hood, we discussed covariance and contravariance of functions. Let's revisit those details using these new tools:

```
//
src/main/scala/progscala2/metaprogramming/func.sc

class CSuper                { def msuper() = println("CSuper")
}
class C      extends CSuper { def m()      = println("C") }
class CSub   extends C      { def msub()   = println("CSub") }

typeOf[C      => C     ] =:= typeOf[C => C]      // true    ❶
typeOf[CSuper => CSub  ] =:= typeOf[C => C]      // false
typeOf[CSub   => CSuper] =:= typeOf[C => C]      // false

typeOf[C      => C     ] <:< typeOf[C => C]      // true    ❷
typeOf[CSuper => CSub  ] <:< typeOf[C => C]      // true    ❸
typeOf[CSub   => CSuper] <:< typeOf[C => C]      // false   ❹
```

❶

None of the pairs is equal except for an exact match.

❷

A type is its own subtype, so this should be true.

❸

True because the argument is a supertype of `C`, satisfying contravariance for arguments, and the return type is a subtype of `C`, satisfying covariance of return types.

❹

Breaks both rules for argument types and return types.

Tip

Can't remember the rules for when one type is a subtype of another? Use `typeOf` or our `toType` method and `<:<` to figure it out.

Now consider some of the information we can learn from the types. First, the `Type` returned is an instance of `TypeRef`, so we use an extractor to determine the "prefix," the symbol for the type (its name), and any type parameters it takes:

```
def toTypeRefInfo[T : TypeTag](x: T): (Type, Symbol, Seq[Type]) = {
  val TypeRef(pre, typName, parems) = toType(x)
  (pre, typName, parems)
}
```

The `Type` and `Symbol` types in the tuple are both defined in `reflect.runtime.universe`, not to be confused with `scala.Symbol`.

```
                                      // (scala.type, class Int,
toTypeRefInfo(1)                      List())
toTypeRefInfo(true)
// (scala.type, class Boolean,
List())
toTypeRefInfo(Seq(1, true, 3.14))
// (scala.collection.type, trait
Seq,
                                      //
                                      List(AnyVal))
                                      // (scala.type, trait
toTypeRefInfo((i: Int) => i.toString) Function1,
                                      //    List(Int,
                                      java.lang.String))
```

Note the different `scala.collection.type` "prefix" for `Seq` versus `scala.type` for the other examples. Both `Seq` and `Function1` have nonempty type parameter lists, as we would expect.

We get even more information with `TypeApi`. Let's try it with `Seq` in the REPL, to see the types returned. We'll elide long output:

```
scala> val ts = toType(Seq(1, true, 3.14))
ts: reflect.runtime.universe.Type = Seq[AnyVal]

scala> ts.typeSymbol
res0: reflect.runtime.universe.Symbol = trait Seq

scala> ts.erasure
res1: reflect.runtime.universe.Type = Seq[Any]

scala> ts.typeArgs
res2: List[reflect.runtime.universe.Type] = List(AnyVal)

scala> ts.baseClasses
res4: List[reflect.runtime.universe.Symbol] =
  List(trait Seq, trait SeqLike, trait GenSeq, trait GenSeqLike, ...)

scala> ts.companion
res5: reflect.runtime.universe.Type = scala.collection.Seq.type

scala> ts.decls
res6: reflect.runtime.universe.MemberScope = SynchronizedOps(
  method $init$, method companion, method seq)

scala> ts.members
res7: reflect.runtime.universe.MemberScope = Scopes(
  method seq, method companion, method $init$, method toString, ...)
```

Most of these are self-explanatory. The `companion` method returns the type of the companion type, `decls` returns the declarations in `Seq` itself, while `members` returns all declarations that are inherited, too.

You'll find more examples in the overview and the reflection Scaladocs.

Macros

Scala's current macro system has been used to implement clever solutions to difficult design problems in many advanced toolkits. However, to use it requires understanding compiler internals, such as the abstract syntax tree (AST) representation used by the compiler. So, a principle goal of the *Scala Meta* project is to implement a new macro system that avoids this coupling to compiler details and the learning burden for users. It will also apply various lessons learned from work on the first system

Because *Scala Meta* is not yet available and the current system will eventually go away, we won't discuss it in detail, but we will end our discussion with an example. However, one feature is expected to remain relatively unchanged in *Scala Meta*, namely *quasiquotes*, which is a tool for manipulating ASTs much more easily, using interpolated strings. It eliminates much of the boilerplate and detailed knowledge required to write macros in the old API. The quasiquote documentation can be found in the Scala docs.

Note that the remaining examples for this chapter only work with Scala 2.11, although they use just a few changes in the API since 2.10. Specifically, a `showCode` helper method we'll see shortly is new and an API change was made that we'll mention in the macro example later.

Let's work through some of the features:

```
// src/main/scala/progscala2/metaprogramming/quasiquotes.sc

import reflect.runtime.universe._
// ❶

import reflect.runtime.currentMirror
// ❷
import tools.reflect.ToolBox
val toolbox = currentMirror.mkToolBox()
```

❶

Import the `universe` features needed for quasiquotes.

❷

Bring in the convenient “toolbox.”

There are several ways to construct quasiquotes, depending on the type of AST tree you’re building. We’ll use the general form `q"…"` and `tq"…"` for type expressions. The full list of options with examples can be found at in the Scala documentation.

```
                "case class C(s:
scala> val C = qString)"
C: reflect.runtime.universe.ClassDef =
case class C extends scala.Product with scala.Serializable {
  <caseaccessor> <paramaccessor> val s: String = _;
  def <init>(s: String) = {
    super.<init>();
    ()
  }
}

scala> showCode(C)
res0: String = case class C(s: String)

scala> showRaw(C)
res1: String = ClassDef(Modifiers(CASE), TypeName("C"), List(), ...)
```

The `showCode` method prints a string similar to the original Scala syntax of the declaration (exactly the same in this simple example), and `showRaw` prints the types corresponding to the actual AST tree.

Whereas `q` is used for general quasiquotes, `tq` is used to construct trees for types, specifically:

```
       val  q =  q
scala> "List[String]"
q: reflect.runtime.universe.Tree = List[String]

scala> val tq = tq"List[String]"
tq: reflect.runtime.universe.Tree = List[String]

scala> showRaw(q)
res2: String = TypeApply(Ident(TermName("List")),
  List(Ident(TypeName("String"))))

scala> showRaw(tq)
res2: String = AppliedTypeTree(Ident(TypeName("List")),
  List(Ident(TypeName("String"))))

scala> q equalsStructure tq
res4: Boolean = false
```

We need to use `showRaw` to see that they are actually different. The Scaladoc page for `scala.reflect.api.Trees#TypeApplyExtractor` explains the difference. `TypeApply` corresponds to a type specification that appears in a *term*, such as `foo[T]` in `def foo[T](t: T) = ...`, and `AppliedTypeTree` is used for type declarations, like `T` in `val t: T`.

To test equality, use `equalsStructure`.

You can expand other quasiquotes into a quasiquote using string interpolation `${…}`, called “unquoting”:

```
scala> Seq(tq"Int", tq"String") map { param =>
          q
        "case class C(s:
     |  $param)"
     |  } foreach { q =>
          println(showCode(q
     |  ))
     |  }
case class C(s: Int)
case class C(s: String)
```

Hence we can parameterize code generation! Note that we used type quasiquotes (`tq"…"`) because the “param” function argument is used for a type declaration. Try replacing `showCode` with `showRaw`, then compare the output when you replace `tq` quasiquotes with `q` or just a string, e.g., `"Int"`.

In some cases, normal values are “lifted” to quasiquotes when interpolated:

```
scala> val list = Seq(1,2,3,4)
                 "%d, %d, %d,
scala> val fmt = %d"
                     "println($fmt,
scala> val printq = q..$list)"
```

The `..$list` syntax expands the list into comma-separated values. (There is also a `...$list` for sequences of sequences.) Here we use it to generate a call to a variable-argument function, `println`. The reverse process is "unlifting," commonly used with pattern matching on quasiquoted strings.

```
            "${i: Int} + ${d:              "1 +
scala> val qDouble}"                    = q3.14"
i: Int = 1
d: Double = 3.14
```

There are a few other kinds of quasiquotes: `cq` generates trees for case clauses, `fq` generates trees for `for` comprehensions, and `pq` generates trees for pattern-match expressions. See the Scala docs for detailed examples.

A Macro Example: Enforcing Invariants

When we discussed *Design by Contract* in Better Design with Design by Contract, we mentioned one aspect of a contract is the *invariants* that should be true before *and* after every method invocation and state change. Let's implement a macro that enforces invariants.

Recall that we said that macros are a limited form of compiler plug-in, invoked in an intermediate phase of the compilation process. This leads to a requirement for macros that they must be compiled separately and ahead of time from the code that uses them. We'll implement the macro in a source file and use it in a *ScalaTest* test file to meet this requirement. This works because `sbt` compiles tests separately from the main code. Also, macro implementations follow certain idioms, which we'll also see. Here is the source for the macro `invariant`:

```
//
src/main/scala/progscala2/metaprogramming/invariant.scala
package metaprogramming
import reflect.runtime.universe._                                    // ❶
import scala.language.experimental.macros
import scala.reflect.macros.blackbox.Context                         // ❷

/**
 * A Macro written using the current macro syntax along with
quasiquotes.
 * Requires a predicate for an invariant to be true before each
expression
 * is
evaluated.
 */
object invariant {                                                   // ❸
  case class InvariantFailure(msg: String) extends RuntimeException(msg)

  def apply[T](predicate: => Boolean)(block: => T): T = macro impl   // ❹

  def impl(c: Context)(predicate: c.Tree)(block: c.Tree) = {         // ❺
    import c.universe._                                              // ❻
    val predStr = showCode(predicate)                                // ❼
    val q"..$stmts" = block                                          // ❽
    val invariantStmts = stmts.flatMap { stmt =>                     // ❾
                 "FAILURE! $predStr == false, for statement:
      val msg = s"                                        + showCode(stmt)
                 "throw new
      val tif = qmetaprogramming.invariant.InvariantFailure($msg)"
                    "if (false == $predicate)
      val predq2 = q$tif"
            "{ val tmp = $stmt; $predq2; tmp
      List(q};"                                    )
    }
    val tif = q
"throw new
metaprogramming.invariant.InvariantFailure($predStr)"
                 "if (false == $predicate)
    val predq = q$tif"
     "$predq;
    q..$invariantStmts"                                              // ❿
  }
}
```

❶

Import the reflection and macro features required. We are building a “blackbox” macro; it won’t change the type signature of the expression it encloses (see the docs for details).

❷

For 2.10, use `scala.reflect.macros.Context` instead.

❸

The `invariant.apply` method is used to wrap the expressions for which we want to enforce invariants. If a failure occurs, `InvariantFailure` is thrown.

❹

Macros always start with a public method that is invoked in client code, with a body that contains `macro impl`. In this case, `invariant` is passed two arguments: a predicate to test around each statement in the second argument, and a `block` of code to evaluate.

❺

The implementation method `impl` takes arguments corresponding to those for `apply`, where each one is the corresponding abstract syntax tree generated from the expression. The type `c.Tree` is a path-dependent type in the `Context` object, the first argument to `impl`.

❻

We must use the universe corresponding to the context, so we import its members.

❼

Create a string for the `predicate` that will be used in failure messages.

❽

Convert the `block` into a sequence of statements, using pattern matching.

❾

Flat map over the statements, modifying each one to capture its return value, then check the predicate (throwing an `InvariantFailure` on failure), and then finish with the return value.

❿

Rejoin the statements, prefixed with an initial test of the predicate, and return the modified AST.

Without quasiquotes, this implementation would be much harder to write, because we would have to know details of the AST implementation and how to manipulate AST trees.

Let’s see an example and try it in the following ScalaTest, where we’ll use a class `Variable` with two mutable fields and we’ll impose the invariant that the `s` field, a string, is never changed:

```
// src/test/scala/progscala2/metaprogramming/InvariantSpec.scala
package metaprogramming
import reflect.runtime.universe._
import org.scalatest.FunSpec

class InvariantSpec extends FunSpec {
  case class Variable(var i: Int, var s: String)

  describe ("invariant.apply") {
    def succeed() = {                                               //
❶
      val v = Variable(0, "Hello!")
      val i1 = invariant(v.s == "Hello!") {                         //
❷
        v.i += 1
        v.i += 1
        v.i
      }
      assert (i1 === 2)
    }

        "should not fail if the invariant
    it (holds"                                 ) { succeed() }

        "should return the value returned by the
    it (expressions"                                         ) { succeed()
}

        "should fail if the invariant is
    it (broken"                                ) {
      intercept[invariant.InvariantFailure] {                       //
❸
        val v = Variable(0, "Hello!")
        invariant(v.s == "Hello!") {
          v.i += 1
          v.s = "Goodbye!"
          v.i += 1
        }
      }
    }
  }
}
```

❶

Helper method to check cases where the invariant holds.

❷

Require the string field to remain the same while executing the statements in the block.

❸

Expect `InvariantFailure` because the string is modified.

It’s instructive to comment out the `intercept[…]` line and the corresponding closing brace. The test fails with the following error message:

```
[info] - should fail if the invariant is broken *** FAILED ***
[info]   metaprogramming.invariant$InvariantFailure:
  FAILURE! v.s.==("Hello!") == false, for statement:
v.`s_=`("Goodbye!")
```

A powerful feature is our ability to show a readable message for the predicate that failed and the statement that triggered the failure. The whole implementation is just a few dozen lines of code, demonstrating the power of macros.

If you wrote all the code by hand, the first test in `InvariantSpec` might look schematically like the following, where I converted a few statements in the loop to a helper method `fail`, to reduce boilerplate:

```
def fail[T](predStr: String, stmtStr: String): Nothing = {
             "FAILURE! $predStr == false, for statement:
  val msg = s$stmtStr"
  throw new metaprogramming.invariant.InvariantFailure(msg)
}
val v = Variable(0, "Hello!")
val i1 = {
                            "v.s ==
  if (v.s != "Hello!") fail(\"Hello!\""        , "")
  val tmp1 = v.i += 1
                            "v.s ==              "v.i +=
  if (v.s != "Hello!") fail(\"Hello!\""        , 1"        )
  val tmp2 = v.i += 1
                            "v.s ==              "v.i +=
  if (v.s != "Hello!") fail(\"Hello!\""        , 1"        )
  val tmp3 = v.i
                            "v.s ==
  if (v.s != "Hello!") fail(\"Hello!\""        , "v.i")
  tmp3
}
```

The quasiquotes documentation has other useful examples, such as printing debug statements before each statement in a block is executed.

Final Thoughts on Macros

The power of macros is quite seductive, but developing, debugging, and maintaining them is challenging. You can find many examples of their use in third-party libraries. Also keep in mind that the whole reflection API, especially the macros package, is considered experimental and it will continue to evolve rapidly.

Wrapping Up and Looking Ahead

If you've read this far in *Programming Scala, Second Edition*, you have learned about all the major features of the language and how best to use them. I hope you'll find the code examples useful as templates for your own projects. For a more extensive collection of examples for different kinds of applications and toolkits, see the *Activator* project on the Typesafe website. Typesafe also offers developer and production subscriptions for Scala, Akka, Play, and a growing list of other tools. Typesafe provides training and consulting, too.

How will Scala change in the next few years? It’s been five years since the first edition of this book. The changes have been enormous, both in the maturity of the language and in industry adoption. I expect brisk adoption to continue, especially in the *Big Data* space. Evolution of Scala itself is stabilizing. Even macros will stabilize in a year or two. Much of the work on Scala and the larger ecosystem now aims to improve performance, reduce bugs, deprecate language “warts,” and improve tooling around Scala, like IDE support.

Martin Odersky is also working on a new Scala-like language based on a new type system called *DOT*, for *dependent object typing*, which may become Scala 3.0 (see the DOT slideshow and PDF for more).

DOT is based on *dependent typing*, the state of the art in type theory that will allow you to express concepts like “an array of three elements” as a type. Currently, the type systems of most languages can’t express the size constraint as part of a type. Why does this matter? It pushes us closer to the ultimate goal of provably correct programs, where the types are theorems and the programs are proofs (see the Wikipedia page).

The new language will also simplify the type system in other ways and remove other language warts. It is several years away, at least.

In the meantime, you can use Scala now to improve how you create applications, while leveraging the richness of the mature Java ecosystem and going forward, the vibrant JavaScript ecosystem with the new port of Scala to JavaScript, scala.js. I hope that *Programming Scala, Second Edition* will help you be successful as you go.