Designing Embedded Domain-Specific Languages in Scala

A Case Study with Action Systems

Philipp Gabler

2015-06-17

Overview of the work

Exploring Scala's capabilities for embedded DSLs

- What helpful features and techniques are there?
- How (far) can they be used, and what are their limits?
- Some examples of their usage: explanatory and “in the wild”

Overview of the work

Construct a DSL for the Action Systems used in model-based mutation testing

- How does the process of development with this goal look like?
- Where does the language help with the embedding, and where does is stand in our way?
- What features are missing, and what could have been done better (in other languages, or with different means)?

“Regular” language features

- Completely object oriented (no strange “primitive types”)
- Functional: immutability preferred, proper closures and lambdas
- Powerful type system: bounded polymorphism done right, with higher kinds; structural types
- Traits: better interfaces, mixin inheritance
- Sophisticated modules, ADTs + pattern matching, monad comprehensions,. . .

Not so regular features (a selection)

- Operators: Inline calls and proper expressions
- Empowered function calling: blocks and by-name args
- Implicits: Extension wrappers and less boilerplate
- Extractors (pattern synonyms/active patterns), interpolators, macros…

Expressions that look like expressions

Java

```
// not so logical...
Expr e1 = new Or(new Var("a"),
                 new And(new Var("b"),
                         new Not(new Var("c"))));
```

Scala

```
// as simple as that!
val e1: Expr = "a" || "b" && !"c"
```

Expressions that look like expressions

The magic behind it:

```
def ||(other: Expr) = Or(this, other)
def &&(other: Expr) = And(this, other)
def unary_! = Not(this)
implicit def stringToExpr(s: String): Var = Var(s)
```

Also not much more code than Java variant.

Combinators!

```scala
def identifier: Parser[String] = "[^()' ]+".r
def readMacroIdentifier: Parser[String] = "'"

def atom = (identifier ^^ Atom) <~ whiteSpace.?

def cons =
  (parenthesized(sexpr.*) ^^ ConsList) <~ whiteSpace.?

def readMacro = (readMacroIdentifier ~ sexpr) <~
  whiteSpace.? ^^ { case s~e => ReadMacro(s, e) }

def sexpr = whiteSpace.? ~> (readMacro | cons | atom)
```

"Natural language interface"[1]

```
class ExampleSpec extends FlatSpec with Matchers {
  "A Stack" should "behave right" in {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    stack.pop() should be (2)
    stack.pop() should be (1)
  }

  it should "throw NoSuchElementException" in {
    val emptyStack = new Stack[Int]
    a [NoSuchElementException] should be thrownBy {
      emptyStack.pop()
    }
  }
}
```

[1] http://www.scalatest.org/

Blocks & by-name args I

Defining this…

```
def _while(condition: => Boolean)(body: => Unit): Unit = {
  if (condition) { body; _while(condition)(body) }
}
```

…we get this!

```
var x = 10
_while(x > 0) {
  print(x)
  x -= 1
}
// 10987654321
```

Blocks & by-name args II

This has also very practical semantics:

Java

```
Socket socket = new Socket("example.com", 80);
try {
  socket.getOutputStream().write("GET".getBytes());
} catch (IOException e) { ... }
// what should we return? null?
```

Scala

```
val socket = Try(new Socket("example.com", 80))
socket map { s =>
  s.getOutputStream write "GET".getBytes
} recover {
  case e: IOException => ???
}
// can simply return socket, or better: socket.toOption
```

Blocks & by-name args III

We even could implement this:

```
on error in {
  socket.getOutputStream write "GET".getBytes
} resume next
// error-free code!
```

Implicits

Have you noticed them?
No, because you shouldn't!

Implicits

Patching strings:

```
trait Read[T] { def read(s: String): T }

implicit object boolIsRead extends Read[Boolean] {
  def read(s: String) = s match {
    case "true" => true
    case "false" => false
  }
}

implicit class StringReadOps(val self: String) {
  def readAs[T](implicit readT: Read[T]): T =
    readT.read(self)
}
```

No more helpers:

```
"true".readAs[Boolean]
// becomes: StringReadOps("true").readAs[Int](boolIsRead)
// which actually is: boolIsRead.read("true")
```

Implicits

We can even nest this:

```
implicit def numericIsRead[N](implicit numN: Numeric[N])
  : Read[N] = new Read[N] {
  def read(s: String) = numN.fromInt(s.toInt)
}

"42".readAs[Int]
// is actually: numericIsRead(intIsNumeric).read("42")
```

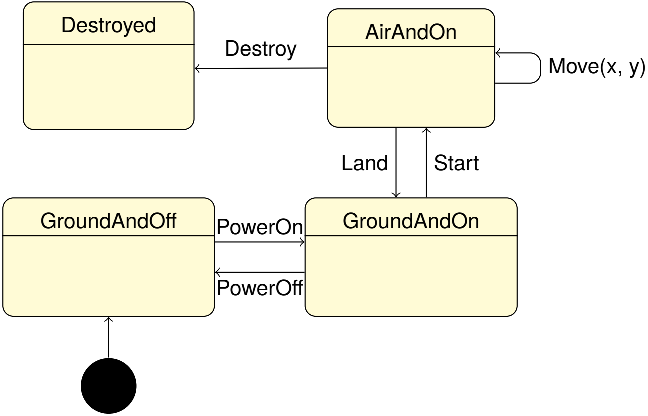
Implicits are present everywhere

```scala
val duration = 1.second + 42.millis

Seq((1,3), (2,1), (1,2)).sorted

42 + " is the answer!"

sender ! Received(answer)

val f: Future[String] = Future {
  s + " future!"
}
```

Context: Action Systems & testing

- Back, 1983: description of distributed sytems, alternative to CSP formalism
- *Processes* can participate in one *action* at a time, if it is enabled (its *guard* is true)
- Now: Action Systems are useful *test models* for *model based testing*
- Specifically: Model based mutation-testing of nondeterministic systems

Example system



Reference (existing) syntax

```
destroy() if mode == Air then
{
  mode := Destroyed;
  engine := 0;
};

move(x:MyNat, y:MyNat) if mode == Air && engine == 1 then
{
  pos_x := pos_x + x;
  pos_y := pos_y + y;
};
```

Implemented syntax

```
when('Destroy) given mode === Air then_do (
  mode := Destroyed,
  engine := F
)

when('Move('dx, 'dy)) given (
  mode === Air && engine === T) then_do (
  pos_x := pos_x + 'dx,
  pos_y := pos_y + 'dy
)
```

Gherkin example[2]

```
Given some precondition
  And some other precondition
 When some action by the actor
  And some other action
  And yet another action
 Then some testable outcome is achieved
  And something else we can check happens too
```

[2] https://github.com/cucumber/cucumber/wiki/Gherkin

Overview of the ActionSystem trait

```
trait ActionSystem {
  // concrete:
  def run: Stream[Choice]
  def initialize(assignments: Assignment[State]*): Unit
  def addAction(action: Action): Unit

  // abstract:
  type State
  def chooseAction(actions: Seq[Action]): Option[Action]
  def chooseParameters(label: Label,
                       params: Seq[Variable[State]])
    : Map[Variable[State], State]
}
```

Look of the plain implementation

Showing code in IDE

Properties of the implementation

- The system contains a mutable environment to represent its state
- Actions are represented symbolically by expression ADTs, which are executed at evaluation (deep embedding)
- Running happens by lazily evaluating a `Stream[Choice]`, which is defined recursively
- As much semantics as possible is left abstract and can be mixed in; actions only define structure and behaviour, not way of execution

Extra features

- Multiple actions with same name automatically supported (internally kept separate)
- External statements (additionally to assignments):

```
externally {
  println(s"E: x = ${'x.value}, y = ${'y.value}")
  if ('x.value < 0 || 'y.value < 0) {
    println("Aborting: x < 0 || y < 0.")
    abort
  }
}
```

Possible improvements

- Better Parametrization of choice methods
- Using macros to allow using blocks instead of parameter lists
- Better support for state types (currently mainly ints), and actually use Scala's type system (or Shapeless generics)
- Wildcard parameters & pattern matching, nested actions
- Make evaluation immutable (currently: stateful updates)

Thank you!

Questions?