Parser macros for Scala

Martin Duhem

September 20, 2015

Plan

1. Overview of parser macros
2. Demonstration
3. Implementation
4. Current limitations & future work

Overview of parser macros

Overview of parser macros

What are parser macros?

- A new macro flavor
- They take streams of tokens as arguments
- Their arguments can have an arbitrary syntax
- They can introduce new definitions

Overview of parser macros

What are tokens?

- Tokens represent atomic fragments of input
- They are a key component of `scala.meta`
- Good abstraction level for parsing

```
scala> "val foo = bar".tokens
res0: meta.Tokens = Tokens(BOF (0..0), val (0..3),   (3..4),
  foo (4..7),   (7..8), = (8..9),   (9..10), bar (10..13),
  EOF (13..13))
```

Overview of parser macros

Using parser macros

- Parser macro applications look like function applications
- Parameters are in different set of parentheses
- The first set must be prefixed by '#'

```
Util.datatypes#{
  * Greeting:
    - msg: String
    * Anon:
    * Named:
      - author: String
      - ...
}
```

```
abstract class Greeting(val msg: String)
class Anon(msg: String) extends Greeting(msg)
object Anon { def apply(msg: String) = ... }
class Named(msg: String, val author: String, ...) { ... }
```

Overview of parser macros

Why are parser macros useful?

- They give us a mean to experiment with new syntaxes

```
newSyntax#{
  factorial of n (Int) is
   - if n < 2 : 1
   - otherwise: n * factorial(n - 1)
}
```

```
def factorial(n: Int) =
  if (n < 2) 1 else n * factorial(n - 1)
```

Overview of parser macros

Why are parser macros useful?

- Could they simplify some transformations made by scalac?

```
For#{
  x <- (1 to 10) if x % 2 == 0
  y <- (20 to 30) if y % 3 == 0
}{yield x * y}
```

```
(1 to 10).
  filter(x => x % 2 == 0).
  flatMap(x => (20 to 30).
    filter(y => y % 3 == 0).
    map(y => x + y))
```

Overview of parser macros

Why not just use string interpolation?

- We could, but there would be a few problems.
- Strings are harder to handle than tokens.
- They are not as precise.
- There are a few problems with string interpolation...

```
scala> s"\""
<console>:1: error: unclosed string literal
s"\""
    ^
```

Overview of parser macros

Writing parser macros

- Parser macros use scala.meta
- We can enjoy all scala.meta APIs!

```
object Case {
  def Class(name: Tokens, fields: Tokens) = macro {
    val cName =
      name(1).parse[Term].asInstanceOf[Term.Name]
    val fields = makeFields(fields)
    q"""class $cName(..$fields) {
      override def toString = ...
    }
    object $cName {
      ...
    }"""
  }
}
```

Demonstration

Implementation of parser macros

Implementation of parser macros

Modifying Scala's parser

- We want to allow parser macro applications:

```
Lib.enum#(WeekDays)(Mon Tue Wed Thu Fri Sun Sat)
```

- They should also be allowed at top-level

Implementation of parser macros

Modifying Scala's parser

- We modified Scala's grammar to accept parser macro applications

$$
\begin{array}{lll}
\langle \mathit{TopStat} \rangle & ::= & \text{Unchanged} \\
& | & \langle \mathit{PMacroRef} \rangle \\
\\
\langle \mathit{PMacroRef} \rangle & ::= & \langle \mathit{QualId} \rangle \ (\text{`.'} \ \langle \mathit{PMacroRef} \rangle) \mid \langle \mathit{PMacroApp} \rangle \\
\\
\langle \mathit{PMacroApp} \rangle & ::= & \text{`\#'} \ \langle \mathit{PMacroArgs} \rangle \\
\\
\langle \mathit{PMacroArgs} \rangle & ::= & (\text{`(`} \ \langle \mathit{Anything} \rangle \ \text{`)'} \mid \text{`\{'} \ \langle \mathit{Anything} \rangle \ \text{`\}'}) \\
& & [\langle \mathit{PMacroArgs} \rangle] \\
\\
\langle \mathit{SimpleExpr1} \rangle & ::= & \text{Unchanged} \\
& | & \langle \mathit{PMacroApp} \rangle \\
\\
\langle \mathit{Anything} \rangle & ::= & \text{Any sequence of characters, except `)' and `\}'}
\end{array}
$$

Implementation of parser macros

Rewriting parser macro applications

- Parser macros can expand into definitions and expressions
- Expanding into definitions is very complicated
- We take advantage of Macro Paradise that already does that
- Different rewriting for applications in stat or expr position

Implementation of parser macros

Rewriting parser macro applications

```
object Calendar {
  Lib.enum#(WeekDays)(Mon Tue Wed Thu Fri Sat Sun)
  ...
}
```

```
object Calendar {
  @ParserMacroExpansion(Lib.enum)
  object TemporaryObject {
    val tokens = List(
      "WeekDays",
      "Mon Tue Wed Thu Fri Sat Sun"
    )
  }
  ...
}
```

Implementation of parser macros

Rewriting parser macro applications

```
object Math {
  def gcd(a: Int, b: Int) = Lib.GCL#{
    do a < b -> b := b - a
    |  b < a -> a := a - b
    od
    b
  }
  ...
}
```

```
object Foo {
  def gcd(a: Int, b: Int) = Lib.GCL
  ...
}
```

- `Lib.GCL` has an attachment that contains the arguments.

Implementation of parser macros

Generating synthetic implementations

- Unlike `scala.reflect` macros, we don't need an explicit impl
- Macros are special methods, not visible to Java reflection
- Macro implementation will be invoked with Java reflection!

```
def enum(n: Tokens, items: Tokens) = macro {
  val name = makeTermName(n(1))
  ...
}
```

```
def enum(n: Tokens, items: Tokens) = macro {
  val name = makeTermName(n(1))
  ...
}
private def enum$impl(n: Tokens, items: Tokens) = {
  val name = makeTermName(n(1))
  ...
}
```

Implementation of parser macros

Expanding parser macros

- There are 2 slightly different implementations:
- Expansion in expression vs. statement position
- Difference: How are arguments extracted?
  - Expression position: attachment
  - Statement position: from `TemporaryObject`

Implementation of parser macros

Expanding parser macros

```
object Foo {
  @ParserMacroExpansion(Lib.enum)
  object TemporaryObject {
    val tokens = List(
      "WeekDays",
      "Mon Tue Wed Thu Fri Sat Sun"
    )
  }
}
```

1. Extract the name of the macro method `Lib.enum`
2. Resolve it
3. Get the name of the synthesized impl from its signature
4. Get this method using Java reflection

Implementation of parser macros

Expanding parser macros

```
object Foo {
  @ParserMacroExpansion(Lib.enum)
  object TemporaryObject {
    val tokens = List(
      "WeekDays",
      "Mon Tue Wed Thu Fri Sat Sun"
    )
  }
}
```

1. Prepare the arguments
2. Invoke the macro implementation

Implementation of parser macros

Expanding parser macros

- The result of the expansion is a `scala.meta.Tree`
- We have to convert it to `scala.reflect.Tree`
- It is then given back to the compiler!

Current limitations and future work

Current limitations and future work

Separate compilation of providers and clients

- Macro providers and clients must be compiled separately
- Comes from the reflective invocation of the macro impl

- We keep the complete tree of the macro impl
- We could interpret it directly!
- Solves 2 problems:
  - Separate compilation problem
  - We wouldn't need to clone the impl

Current limitations and future work

Cannot introduce top level definitions

```
@ParserMacroExpansion(Lib.enum)
object TemporaryObject {
  val tokens = List(
    "WeekDays",
    "Mon Tue Wed Thu Fri Sat Sun"
  )
}
```

- Parser macros cannot expand into top-level definitions
- Macro paradise says:
  top-level object can only expand into an eponymous object
- The name of the temporary object could be configurable
- This would solve the problem in most situations

Summary

Summary

Parser macros

- Parser macros are a new macro flavor
- Take advantage of Macro Paradise and scala.meta
- Allow quick prototyping of new syntaxes
- Can be used to implement language features

Questions?