ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Quasiquotes for Scala, a Technical Report

Denys Shabalin, Eugene Burmako, Martin Odersky

first.last@epfl.ch

March 2013

Contents

1 Introduction

1.1 Background

With the introduction of macros [1] and reflection API [2] in Scala 2.10 users have noticed that tools for code generation quite often are not perfectly suitable for their needs and that sometimes they are forced to construct and deconstruct ASTs manually.

The main weakness of `reify`, the officially advertised way to compose trees, is that it only works with typechecked trees. Whenever one can't have a typechecked tree (e.g. a tree doesn't make sense on it's own but is fine in some context), then one has to resort to fully manual tree constuction for code generation. This is a tedious and error-prone process that requires deep knowledge of how AST nodes are created and how Scala code is represented with this nodes.

Another weakness of existing tools is lack of any support for high-level tree deconstruction. So whenever one wants to split a complex tree apart one needs to work with pattern matching on direct AST nodes which can be even more complex to understand especially when combined with custom extractors and wildcard patterns.

In this report we introduce quasiquotes for the Scala programming language, which solve the problem of manual tree construction and deconstruction. Included case studies show that quasiquotes indeed are easier to work with than `reify` and manual tree construction. When limitations of current implementation are lifted quasiquotes will become the ultimate tool for the tree manipulation.

1.2 Quasiquotes

Quasiquotes are short snippets Scala code written using string interpolation [3] syntax:

```
q"here.comes { some => Scala(code) }"
```

Snippets like that will always return an untyped tree that corresponds to the code written within a quotes as if it was just parsed by Scala parser:

```
Apply(Select(Ident(TermName("here")), TermName("comes"))
    , List(Function(List(ValDef(Modifiers(PARAM),
    TermName("some"), TypeTree(), EmptyTree)), Apply(
    Ident(TermName("Scala")), List(Ident(TermName("code")
    ))))))
```

The quasiquote version is much more compact and much easier to read than corresponding manually constructed tree. Writing a quasiquote also requires less compiler knowledge from the end user.

String interpolation also allows splicing of code blocks right into the quote using `$name` or `${expr}` syntax:

```
q"$a + ${function(call)}"
```

In this case quasiquotes will treat spliced arguments as placeholders that should be filled in with values got from the corresponding expressions, so this is equivalent to:

```
Apply(Select(a, TermName("$plus")), function(call))
```

Quasiquotes can not only be used for trees construction but also for deconstruction:

```
val q"$a + $b" = q"1 + 2"
```

This will extract 1 & 2 from the tree on the right side. Similarly we can use quasiquotes for pattern matching on trees:

```
q"foo(10)" match {
  case q"$f($x)" => (f, x)
}
```

In pattern matching it's also possible to nest custom extractors. For example a following code will only match function call that takes a single argument which should be `Literal`:

```
q"foo(10)" match {
  case q"$f(${x: Literal})" => (f, x)
}
```

Similarly we can omit parts of the tree we match upon with wildcard:

```
q"foo(10)" match {
  case q"${_}($x)" => x
}
```

1.3 Splicees and cardinality

Splicees (values inserted into a quasiquote) can have following types:

- Term and type names. Splicing a name allows to fill class, function, method names, parts of select expression etc. For example:

  ```
  q"def $foo = $a.$b"
  ```

  Here `foo`, `b` can only be term names, `a` identifier can also possibly correspond to some tree. This quasiquote will translate to:

  ```
  DefDef(Modifiers(), foo, List(), List(), TypeTree(),
      Select(Ident(a), b))
  ```

- Trees. This is the most natural type of a splicing argument. Trees can be spliced into the any other tree which contains fields of `Tree`, `List[Tree]` and `List[List[Tree]]` types. For example:

  ```
  q"$foo($bar)"
  ```

  This expression is the same as:

  ```
  Apply(foo, List(bar))
  ```

Where both foo and bar are trees.

- Modifiers and Flags. Splicing and extraction of such values is essential part of work with Scala's definition trees (DefDef, ValDef, ClassDef etc.) The goal of the Modifiers is to represent additional meta-data about corresponding definition: flags and annotations. Flags represent a set of scala-specific modificators to the given valdef: accessibility restrictions (private, protected), case class modifier, laziness of vals, abstractness of members etc.

  For example a following snippet:

  ```
  q"$mods val $name: $tpe = $rhs"
  ```

  Is a dirrect correspondence to the underlying tree:

  ```
  ValDef(mods, name, tpe, rhs)
  ```

  As a shorthand it's also possible to splice particular flags in the same position:

  ```
  q"$IMPLICIT val $name: $tpe = $rhs"
  ```

  Which will set the same flags as if the flags were set by corresponding keywords:

  ```
  q"implicit val $name: $tpe = $rhs"
  ```

  For most of the default flags it's easier to use default Scala keywords. But for some flags (e.g. SYNTHETIC, PARAM) such splicing is essential as there is no dirrect correspondence in regular scala code.

- Symbols. Unlike other values they are not stored directly inside of Scala trees but encoded in a particular way depending on the place you splice the symbol.

  Currently quasiquotes support two places for symbol splicing:

  ```
  q"$sym1.$sym2()"
  ```

  In regular tree position (corresponds to Ident node) and in member selection position (corresponds to Select node).

- Iterable of Trees. This kind of splice requires special cardinality ".." annotation:

  ```
  q"$foo(..$bar)"
  ```

  This annotations clearly states that `bar` should be a list of trees and should be spliced accordingly into final tree:

  ```
  Apply(foo, bar)
  ```

If the `bar` won't have a correct corresponding type (e.g. `Tree` instead of required `List[Tree]`) a compile-time error will be thrown.

It's also possible to combine subtrees of different cardinalities in the same sub expression:

```
q"$foo($bar, ..$baz)"
```

This will behave as if first non-list arguments are prepended to the list:

```
Apply(foo, bar :: baz)
```

- Iterable of Iterable of Trees. This splices are useful for inserting valdefs into function definition, function call with multiple arguments lists and whenever there is `List[List[Tree]]` field in some particular `Tree`. For example:

  ```
  q"def foo(...$args) = ???"
  ```

  This will allow splice a list of lists of valdefs into a function definition. As you can see there is a "`...`" cardinality annotation similar to "..". It's easy to generalize cardinalities to any `Iterable[...Iterable[Tree]...]` type but currently there are no Trees that would benefit from that.

- Liftable types. Quasiquotes provide Liftable trait that allows end users to define a conversion from their specific type into a tree:

  ```
  trait Liftable[T] {
    def apply(universe: api.Universe, value: T):
        universe.Tree
  }
  ```

  There are a few default liftables pre-defined: for standard value types (numbers, booleans and unit), strings, for types, typed expressions and type tags. In order to define a custom liftable a user should just provide an implicit val with a value that implements `Liftable` interface for the required type. This val will be looked up by macro and it will be used to automatically convert instances of corresponding type T into trees.

1.4 Type quasiquotes

Another notable feature of quasiquotes is construction and deconstruction of trees that correspond to types. A different string interpolator `tq` is used for that:

```
tq"Foo[Bar]"
```

Is the same as writing this tree manually:

```
AppliedTypeTree(Ident(TypeName("Foo")), List(Ident(
   TypeName("Bar"))))
```

Using a `q` interpolator for this snippet will yield a different tree:

```
TypeApply(Ident(TermName("Foo")), List(Ident(TypeName("
   Bar"))))
```

Type quasiquotes behave in the same way as regular quasiquotes and support deconstruction too.

2 Implementation

2.1 Overview

Quasiquotes are implemented as a custom string interpolator. So the following code:

```
q"function($arg)"
```

Is desugared into:

```
StringContext("function(", ")").q(arg)
```

In this code `q` interpolator is defined through an implicit class that should be imported from the universe we are currently working in:

```
trait Quasiquotes { self: Universe =>

  implicit class Quasiquote(ctx: StringContext) {
    object q {
      def apply(args: Any*): Any = macro ???
      def unapply(subpatterns: _*): Option[Any] =
        macro ???
    }
    object tq {
      def apply(args: Any*): Any = macro ???
      def unapply(subpatterns: _*): Option[Any] =
        macro ???
    }
  }
}
```

As you can see quasiquotes are implemented as macros. This makes it possible to transform call to `q` member of `StringContext` into a corresponding AST without any runtime overhead.

It's important to highlight that deconstruction quasiquotes use new untyped flavor of macros which supports inline expansion in pattern-matching position. This lets deconstruction quasiquotes to expand into a dirrectly corresponding pattern before typechecking.

The implementation of macros is resolved through `FastTrack` mechanism. This lets us to have implementation inside of the compiler with all the benefits of private APIs.

Even though current approach is integrated into compiler it's also possible to have quasiquotes implemented as a separate library. In such a case current universe can be resolved as an implicit value. Unfortunately this complicates both implementation and end user experience.

FastTrack looks up macro implementation and resolves it to the call of one of the methods of `scala.tools.reflect.Quasiquotes` class. Depending on current type of quasiquotes (apply/unapply, term/type) that is being used it will be one of: `applyQ`, `applyTq`, `unapplyQ`, `unapplyTq`.

This implementation class is split into multiple slices: `Macros`, `Parsers` and `Reifiers`. All of them are mixed in into the final `Quasiquotes` cake.

Let's have a look at how `applyQ` macro expansion will work on our `q"function($arg)"` example. `applyQ` is defined in `Macros` slice of the cake:

```
trait Macros { self: Quasiquotes =>
  ...

  def applyQ = (new AbstractMacro
                    with ApplyReification
                    with TermParsing).apply()
  def applyTq = (new AbstractMacro
                     with ApplyReification
                     with TypeParsing).apply()
  ...
}
```

As you can see both apply macros share the same implementation but with different instances of parser. Corresponding unapply macros are defined in the similar fashion but with UnapplyReification instead of ApplyReification.

The core implementation of general quasiquote is concentrated in Abstract-Macro class):

```
trait AbstractMacro {
  val parser: Parser
  def reifier(universe: Tree,
              placeholders: Placeholders): Reifier
  def extract = ...
  def generate(args: List[Tree],
               parts: List[String]):
                   (String, Placeholders) = ...
  def apply() = {
    val (universe, args, parts) = extract
    val (code, placeholders) = generate(args, parts)
    val tree =
      parser.parse(code, placeholders.keys.toSet)
    val reified =
      reifier(universe, placeholders)
      .quasiquoteReify(tree)
    reified
  }
}
```

This class contains two abstract members: `parser` and `reifier`. This members are filled in by `TermParsing`, `TypeParsing` and `ApplyReification`, `UnapplyReification` correspondigly giving us 4 possible flavors of quasiquotes.

Let's hava a loo at core logic of quasiquote expansion which is located in apply method.

First of all quasiquotes extracts it's own macroApplication (a tree where a macros is being expanded) to recover universe, args and parts data:

```
def extract = c.macroApplication match {
  case q"$universe.Quasiquote($stringContext.apply(..
     $parts0)).${_}.${_}(..$args)" =>
    val parts = parts0.map {
      case Literal(Constant(s: String)) => s
      case part => c.abort(...)
    }
    if (args.length != parts.length - 1)
      c.abort(...)
    (universe, args, parts)
  case _ =>
    c.abort(...)
}
```

The extraction itself is performed with the help of deconstruction quasiquote and additional sanity checks.

Next step is construction of the code which will be fed to the parser:

```
def generate(args: List[Tree],
             parts: List[String]):
                 (String, Placeholders) = {
  val sb = new StringBuilder()
  var pmap = ListMap[String, (Tree, Int)]()

  foreach2(args, parts.init) { (tree, p) =>
    val (part, cardinality) =
      if (p.endsWith("..."))
        (p.stripSuffix("..."), 2)
      else if (p.endsWith(".."))
        (p.stripSuffix(".."), 1)
      else
        (p, 0)
    val freshname = c.fresh(nme.QUASIQUOTE_PREFIX)
    sb.append(part)
    sb.append(freshname)
    pmap += freshname -> (tree, cardinality)
  }
  sb.append(parts.last)

  (sb.toString, Placeholders(pmap))
}
```

Each argument is replaced with a fresh name. A mapping from fresh names to original trees and their corresponding cardinalities is memorized for future usage. In our example variable generated code will be equal to:

```
function($quasiquote$1)
```

Right after that we feed it into the instance of the customized scala parser:

```
def apply() = {
  ...
  val tree = parser.parse(code, placeholders.keys.toSet)
  ...
}
```

The parser has to know not only about the code to parse but also about a set of special unique identitifiers to solve possible ambiguities in customized scala syntax.

Lastly we reify the parsed tree replacing all the unique quasiquote identifiers with corresponding arguments:

```
def apply() = {
  ...
  val reified =
    reifier(universe, placeholders)
    .quasiquoteReify(tree)
  reified
}
```

The reification strategy will be covered in the next section.

2.2 Changes made to the Scala parser

Quasiquotes use a local instance of a customized Scala parser that makes some minor changes to the syntax:

- Allow function and class arguments to have parameters without types. This is crucial for splicing argument lists. E.g. quasiquote like:

  ```
  q"def foo(..$valdefs) = ???"
  ```

  Should be able to parse `"def foo($quasiquote$1) = ???"` code and then replace randomly generated identifier with incoming valdefs during reification.

- Make blocks with a single expresssion not to be eliminated. This allows to generate blocks of code with multiple statements:

  ```
  q"{ ..$statements }"
  ```

- Allow pattern matching body to contain casedefs without body if the pattern is quasiquote identifiers that represented CaseDef tree:

  ```
  q"x match { case $casedef }"
  ```

  So that end users can splice custom casedefs.

- Allow quasiquote identifiers in flags position to support flag and modifier splicing:

  ```
  q"$mods def foo"
  ```

  Such special flags are encoded as custom annotations.

2.3 Customized reifier

Quasiquotes unify reification and filling of the placeholders into the single phase. This lets us to recursively walk over parsed tree only once. Similarly to the Parser customization quasiquotes inherit from default `scala.reflect.reify.Reifier` and customize its behaviour by overriding some of the reification methods.

The core strategy of the reification is too look for identity-like subtrees with randomly generated names (e.g. `$quasiquote$1`) and replace them with corresponding incoming arguments (splicees). If a tree doesn't have any spliced arguments it will be reified without any deviations from default reification strategy. For example a tree like:

```
Ident(TermName("$quasiquote$1"))
```

Will be reified as an argument which correspond to randomly generated identifier `$quasiquote$1`. The mapping between such identifiers and incoming splicees is stored in `placeholders` map. This map also stores corresponding cardinalities.

The cardinality of the incoming spilcee also affects the way it will be spliced. In order to reify ".." and "..." arguments a hook in `reifyList` looks for such subtrees and reifies them depending on the list structure. For example a list:

```
List(Ident(TermName("$quasiquote$1")), Ident(TermName("
   $quasiquote$1")))
```

Where `$qusasiquote$1` corresponds to splicee `$x` and `$quasiquote$2` to `..$y` will be reified into a tree that is equivalent to the following Scala code:

```
List(x) ++ y
```

To obtain such a close correspondent to manually constructed list we use a following reification logic:

```
def reifyListGeneric[T](xs: List[T])
                       (fill: PartialFunction[T, Tree])
                       (fallback: T => Tree): Tree =
  xs match {
    case Nil =>
      mkList(Nil)
    case _ =>
      def reifyGroup(group: List[T]): Tree = ...
      val head :: tail = group(xs) { (a, b) =>
        !fill.isDefinedAt(a) && !fill.isDefinedAt(b)
      }
      tail.foldLeft[Tree](reifyGroup(head)) {
        (tree, lst) =>
          q"$tree ++ ${reifyGroup(lst)}"
      }
  }
```

This peforms a reification of a list with awareness of possibel high-cardinality placeholders.

Non-trivial case consists of three basic steps:

1. At first we split a list into a sequence of groups where each high-cardinality hole is isolated into a separate group. We distinguish such holes as a domain of `fill` partial function. For example a sequence:

   ```
   List(1, 2, $x, ..$xs, ..$ys)
   ```

   Gets splitted into following groups:

   ```
   List(List(1, 2, $x), List(..$xs), List(..$ys))
   ```

   The group function is implemented as:

   ```
   def group[T](lst: List[T])
                (similar: (T, T) => Boolean) =
     lst.foldLeft[List[List[T]]](List()) {
       case (Nil, el) =>
         List(List(el))
       case (ll :+ (last @ (lastinit :+ lastel)), el)
            if similar(lastel, el) =>
         ll :+ (last :+ el)
       case (ll, el) =>
         ll :+ List(el)
     }
   ```

2. Then we reify each group separatly with `reifyGroup` function:

   ```
   def reifyGroup(group: List[T]): Tree =
     group match {
       case List(elem) if fill.isDefinedAt(elem) =>
         fill(elem)
       case elems =>
         mkList(elems.map(fallback))
     }
   ```

   Each hole gets filled with `fill` function and regular sublists are reified element-wise with `fallback` reifier function. After this step our example gets transformed into:

   ```
   List(q"List(1, 2, $x)", xs, ys)
   ```

3. Lastly we fold the list of reified groups with `++` operator to obtain our final reified list:

   ```
   q"List(1, 2, $x) ++ $xs ++ $ys"
   ```

The reason why we need to generalize reification of lists over fill and fallback functions is to reuse code for regular list reification and reification of annotation lists which requires different treatment. For example a list of annotations in a tree:

```
@Example1 @Example2(argument) def foo
```

Will be represented as constructor calls instead of regular Idents and function Applications:

```
List(q"new Example1", q"new Example2(argument)")
```

3 Case studies

3.1 Quasiquotes are implemented using quasiquotes

An interesting aspect of quasiquotes is that they are implemented using themselves. Originally there were only manual tree construction and deconstruction implementation but recently it became possible to refactor the implementation using a bootstrapped compiler that already contained quasiquotes.

Let's have a look at a single tree construction from quasiquotes reifier which was used to reify a function application with "..." splicee. Before the refactoring it looked like:

```
Apply(
  Apply(
    TypeApply(
      Select(argss, nme.foldLeft),
      List(
        Select(Ident(nme.UNIVERSE_SHORT), tpnme.Tree))),
    List(
      reifyTree(f1))),
  List(
    Function(
      List(
        ValDef(Modifiers(PARAM), nme.f, TypeTree(),
           EmptyTree),
        ValDef(Modifiers(PARAM), nme.args, TypeTree(),
           EmptyTree)),
      Apply(
        Select(Ident(nme.UNIVERSE_SHORT), nme.Apply),
        List(Ident(nme.f), Ident(nme.args))))))
```

And after:

```
q"$argss.foldLeft[$u.Tree]($f1) { $u.Apply(_, _) }"
```

As you can see difference between code size and readability is quite dramatic. It's really easy to understand what's a refactored function is doing while previous version required some additional mental effort.

3.2 An experimental version of ScalaMock

After a first release of experimental branch Macro Paradise [4] a few people were quite enthusiatic about all the new perspectives of type macros and quasiquotes. Notably Paul Butcher, the creator of ScalaMock [6], decided to improve his library with these new features.

Lets have a look at one of his refactoring made using quasiquotes. Initially the code looked like:

```
if (m.isStable) {
  ValDef(
    Modifiers(),
    m.name,
    TypeTree(mt),
    TypeApply(
      Select(
        Literal(Constant(null)),
        TermName("asInstanceOf")),
      List(TypeTree(mt))))
} else ...
```

After refactoring with quasiquotes it was transformed into:

```
if (m.isStable) {
  q"val ${m.name} = null.asInstanceOf[$mt]"
} else ...
```

There is a similar reduction in code size and improvement in readability and ease of understanding.

4 Design limitation: lack of referential transparency

Unlike `reify`, current implementation of quasiquotes lacks referential transparency. So for example in a code like this (this is a modified version of an example from Scala Macros, a Technical Report [5]):

```
object Asserts {
  def assertionsEnabled = ...
  def raise(msg: Any) = throw new AssertionError(msg)
  def assert(cond: Boolean, msg: Any): Unit = macro
     Asserts.assertImpl
  def assertImpl(c: Context)(cond: c.Expr[Boolean], msg:
      c.Expr[Any]): c.Tree = {
    import c.universe._
    if(assertionsEnabled)
      q"if (!$cond) raise($msg)"
    else
      q"()"
  }
}
```

It's not safe to assume that `raise` will always refer to `Asserts.raise` method as it can be hijacked in the local scope where the macro is expanded. In order to fix this currently a user has to manually splice symbols instead:

```
object Asserts {
  def assertionsEnabled = ...
  def raise(msg: Any) = throw new AssertionError(msg)
  def assert(cond: Boolean, msg: Any): Unit = macro
     Asserts.assertImpl
  def assertImpl(c: Context)(cond: c.Expr[Boolean], msg:
      c.Expr[Any]): c.Tree = {
    import c.universe._
    val raiseSym = ...
    if(assertionsEnabled)
      q"if (!$cond) $raiseSym($msg)"
    else
      q"()"
  }
}
```

5 Conclusion

We have presented quasiquotes, a simple solution to the tree construction and deconstruction problem. This solution builds upon existing macro system and extensible string interpolation syntax and integrates right into existing reflection APIs.

Lack of referential transparancy is an unpleasant limitation of current implementation but we are going to address this issue in our future research. Nevertheless, case studies show that quasiquotes can be effectively used to simplify manipulaton of Scala trees even in it's current form.

6 References

1. Scala Macros: http://docs.scala-lang.org/overviews/macros/overview.html
2. Scala reflection overview: http://docs.scala-lang.org/overviews/reflection/overview.html
3. Scala string interpolation SIP: http://docs.Scala-lang.org/sips/pending/string-interpolation.html
4. Macro Paradise: http://docs.scala-lang.org/overviews/macros/paradise.html
5. Eugene Burmako, Martin Odersky "Scala Macros, a Technical Report" (2012) http://infoscience.epfl.ch/record/183862/
6. ScalaMock: http://scalamock.org/