Quasiquotation

From HaskellWiki

Contents

1 Introduction

This is a tutorial for the quasiquoting facility described in Why It's Nice to be Quoted: Quasiquoting for Haskell (https://www.cs.drexel.edu/~mainland/publications/mainland07quasiquoting.pdf) .

Quasiquoting allows programmers to use custom, domain-specific syntax to construct fragments of their program. Along with Haskell's existing support for domain specific languages, you are now free to use new syntactic forms for your EDSLs.

More information on GHC's support for quasiquoting may be found:

- Using QQ with Template Haskell (http://www.haskell.org/ghc/docs/latest/html/users_guide/template-haskell.html#th-quasiquotation)
- Why It's Nice to be Quoted: Quasiquoting for Haskell (https://www.cs.drexel.edu/~mainland/publications/mainland07quasiquoting.pdf) :: PDF

A number of production examples where the result is a string:

- Quasiquoter for Ruby-style interpolated strings (http://hackage.haskell.org/package/interpolatedstring-qq)
- interpolatedstring-perl6 library: QuasiQuoter for Perl6-style multi-line interpolated strings with "q" (http://hackage.haskell.org/package/interpolatedstring-perl6)
- here (http://hackage.haskell.org/package/here)
- heredoc (http://hackage.haskell.org/package/heredoc)

- raw-strings-qq (http://hackage.haskell.org/package/raw-strings-qq)
- neat-interpolation (http://hackage.haskell.org/package/neat-interpolation)
- Interpolation (http://hackage.haskell.org/package/Interpolation)
- A QuasiQuoter for interpolated Text literals, with support for embedding Haskell expressions (http://hackage.haskell.org/package/QuasiText)
- shakespeare-text (http://hackage.haskell.org/package/shakespeare-text)
- dump (http://hackage.haskell.org/package/dump) : Shows values for expressions like
  ```
  let a = 1 in [d| a, map (+a) [1..3] |]
  ```
  by turning them into
  ```
  "(a) = 1   (map (+a) [1..3]) = [2,3,4]"
  ```
  to ease debugging and development.

Additional examples which producing other things:

- The jmacro JavaScript generation library.
- Parsing: regexqq (http://hackage.haskell.org/package/regexqq) , rex (http://hackage.haskell.org/package/rex) or Peggy: A Quasiquoter for a packrat parser (http://tanakh.github.com/Peggy/)
- A QuasiQuoter for lighttpd configuration files (http://hackage.haskell.org/package/lighttpd-conf-qq)
- XML: text-xml-qq (http://hackage.haskell.org/package/text-xml-qq) , xml-hamlet (http://hackage.haskell.org/package/xml-hamlet)
- haskell-src-meta contains quite a few QuasiQuoters (http://hackage.haskell.org/package/haskell-src-meta) , though the toy examples are no longer exported and others have been moved to applicative-quoters (http://hackage.haskell.org/package/applicative-quoters)
- command-qq (http://hackage.haskell.org/package/command-qq)
- process-qq (http://hackage.haskell.org/package/process-qq)
- Rlang-QQ is a quasiquoter for inline R (http://hackage.haskell.org/package/Rlang-QQ)
- a quasiquoter for C syntax trees (http://hackage.haskell.org/package/language-c-quote)
- dbus-qq (http://hackage.haskell.org/package/dbus-qq)

Other QuasiQuotation/TemplateHaskell tutorials include:

- A look at QuasiQuotation (http://quasimal.com/posts/2012-05-25-quasitext-and-quasiquoting.html)

Note that the syntax for quasiquotation has changed since the paper was written: in GHC 7 one writes
`[expr|...|]`
instead of
`[:expr|...|]`
. GHC 6.12 uses
`[$expr|...|]`
. Quasiquotation appeared in GHC 6.9 and is enabled with the `QuasiQuotes` language

option (`-XQuasiQuotes` on the command line or
`{-# LANGUAGE  QuasiQuotes #-}`
in a source file).

We show how to build a quasiquoter for a simple mathematical expression language. Although the example is small, it demonstrates all aspects of building a quasiquoter. We do not mean to suggest that one gains much from a quasiquoter for such a small language relative to using abstract syntax directly except from a pedagogical point of view---this is just a tutorial!

The tutorial is runnable if its contents is placed in files as follows:

Place the contents of the #Syntax and #Parsing sections in the file `Expr.hs` with header

```
{-# LANGUAGE DeriveDataTypeable #-}
module Expr (Expr(..),
             BinOp(..),
             eval,
             parseExpr)
    where

import Data.Generics
import Text.ParserCombinators.Parsec
```

Place the contents of the section #The Quasiquoter in a file `Expr/Quote.hs` with header

```
module Expr.Quote (expr) where

import Data.Generics
import qualified Language.Haskell.TH as TH
import Language.Haskell.TH.Quote

import Expr
```

2 Syntax

Our simple expression language consists of integers, the standard operators +,x,*,/, and parenthesized expressions. We will write a single parser that takes concrete syntax for this language and transforms it to abstract syntax. Using the SYB approach to generic programming, we will then use this parser to produce expression and pattern quasiquoters. Our quasiquoter will allow us to write

```
[expr|1 + 3|]
```
directly in Haskell source code instead of the

corresponding abstract syntax.

An obvious datatype for the abstract syntax of this simple language is:

```
data Expr  =  IntExpr Integer
           |  BinopExpr (Integer -> Integer -> Integer) Expr Expr
    deriving(Show)
```

Unfortunately, this won't do for our quasiquoter. First of all, the SYB technique we use cannot handle function types in a generic way, so the BinopExpr constructor must be modified. SYB also requires that we derive Typeable and Data, a trivial change. Finally, we want to support antiquoting for two syntactic categories, expressions and integers. With antiquoting support, we can write [expr|$x + $int:y|] where x and y are in-scope variables with types Expr and Integer, respectively. The final data types for our abstract syntax are:

```
data Expr  =  IntExpr Integer
           |  AntiIntExpr String
           |  BinopExpr BinOp Expr Expr
           |  AntiExpr String
    deriving(Show, Typeable, Data)

data BinOp  =  AddOp
            |  SubOp
            |  MulOp
            |  DivOp
    deriving(Show, Typeable, Data)
```

An evaluator for our abstract syntax can be written as follows:

```
eval :: Expr -> Integer
eval (IntExpr n)        = n
eval (BinopExpr op x y) = (opToFun op) (eval x) (eval y)
  where
    opToFun AddOp = (+)
    opToFun SubOp = (-)
    opToFun MulOp = (*)
    opToFun DivOp = div
```

3 Parsing

We use Parsec to write a parser for our expression language. Note that we have (somewhat arbitrarily) chosen the syntax for antiquotaton to be as in the above example; a quasiquoter may choose whatever syntax she wishes.

```
small   = lower <|> char '_'
large   = upper
idchar  = small <|> large <|> digit <|> char '\''

lexeme p     = do{ x <- p; spaces; return x  }
symbol name  = lexeme (string name)
parens p     = between (symbol "(") (symbol ")") p

expr    ::  CharParser st Expr
expr    =   term    `chainl1` addop

term    ::  CharParser st Expr
term    =   factor  `chainl1` mulop
```

```
factor  ::  CharParser st Expr
factor  =   parens expr <|> integer <|> try antiIntExpr <|> antiExpr

mulop   =    do{ symbol "*"; return $ BinopExpr MulOp }
        <|>  do{ symbol "/"; return $ BinopExpr DivOp }

addop   =    do{ symbol "+"; return $ BinopExpr AddOp }
        <|>  do{ symbol "-"; return $ BinopExpr SubOp }

integer  ::  CharParser st Expr
integer  =   lexeme $ do{ ds <- many1 digit ; return $ IntExpr (read ds) }

ident  :: CharParser s String
ident  =  do{ c <- small; cs <- many idchar; return (c:cs) }

antiIntExpr  = lexeme $ do{ symbol "$int:"; id <- ident; return $ AntiIntExpr id }
antiExpr     = lexeme $ do{ symbol "$"; id <- ident; return $ AntiExpr id }
```

The helper function parseExpr takes a source code position (consisting of a file name, line and column) and a string and returns a value of type Expr. This helper function also ensures that we can parse the whole string rather than just a prefix.

```
parseExpr :: Monad m => (String, Int, Int) -> String -> m Expr
parseExpr (file, line, col) s =
    case runParser p () "" s of
      Left err  -> fail $ show err
      Right e   -> return e
  where
    p = do  pos <- getPosition
            setPosition $
              (flip setSourceName) file $
              (flip setSourceLine) line $
              (flip setSourceColumn) col $
              pos
            spaces
            e <- expr
            eof
            return e
```

4 The Quasiquoter

Remember, our quasiquoter allows us to write expression in our simple language, such as [expr|2 * 3|], directly in Haskell source code. This requires that the variable expr be in-scope when the quasiquote is encountered, and that it is bound to a value of type Language.Haskell.TH.Quote.QuasiQuoter, which contains an expression quoter and a pattern quoter. Note that expr must obey the same stage restrictions as Template Haskell; in particular, it may not be defined in the same module where it is used as a quasiquoter, but must be imported.

Our expression and pattern quoters are quoteExprExp and quoteExprPat, respectively, so our quasiquoter expr is written as follows:

```
quoteExprExp :: String -> TH.ExpQ
```

```
quoteExprPat :: String -> TH.PatQ

expr  :: QuasiQuoter
expr  =  QuasiQuoter { quoteExp = quoteExprExp,
                       quotePat = quoteExprPat
                      -- with ghc >= 7.4, you could also
                      -- define quoteType and quoteDec for
                      -- quasiquotes in those places too
         }
```

Our quasiquoters re-use the parser we wrote in the previous section, parseExpr, and make use of the generic functions dataToExpQ and dataToPatQ (described in the Haskell Workshop paper). These functions, from the Language.Haskell.TH.Quote package, take a Haskell value and reflect it back into the language as Template Haskell abstract syntax. The catch is that we don't want to handle all values generically: antiquoted values must be handled specially. Consider the AntiExpr constructor; we don't want this constructor to be mapped to Template Haskell abstract syntax for the AntiExpr constructor, but to abstract syntax for the Haskell variable named by the constructor's argument. The extQ combinator allows us to do this nicely by defining a function antiExprExp that handles antiquotations.

```
quoteExprExp s =  do  loc <- TH.location
                      let pos =  (TH.loc_filename loc,
                                  fst (TH.loc_start loc),
                                  snd (TH.loc_start loc))
                      expr <- parseExpr pos s
                      dataToExpQ (const Nothing `extQ` antiExprExp) expr

antiExprExp :: Expr -> Maybe (TH.Q TH.Exp)
antiExprExp  (AntiIntExpr v)  = Just $ TH.appE  (TH.conE (TH.mkName "IntExpr"))
                                                (TH.varE (TH.mkName v))
antiExprExp  (AntiExpr v)     = Just $ TH.varE  (TH.mkName v)
antiExprExp  _                = Nothing
```

The corresponding code for patterns is:

```
quoteExprPat s =  do  loc <- TH.location
                      let pos =  (TH.loc_filename loc,
                                  fst (TH.loc_start loc),
                                  snd (TH.loc_start loc))
                      expr <- parseExpr pos s
                      dataToPatQ (const Nothing `extQ` antiExprPat) expr

antiExprPat :: Expr -> Maybe (TH.Q TH.Pat)
antiExprPat  (AntiIntExpr v)  = Just $ TH.conP  (TH.mkName "IntExpr")
                                                [TH.varP (TH.mkName v)]
antiExprPat  (AntiExpr v)     = Just $ TH.varP  (TH.mkName v)
antiExprPat  _                = Nothing
```

5 Examples

We can now try out a few examples by invoking ghci as follows: `ghci -XQuasiQuotes`

Expr/Quote

```
> [expr|1 + 3 + 5|]
 BinopExpr AddOp (BinopExpr AddOp (IntExpr 1) (IntExpr 3)) (IntExpr 5)
> eval [expr|1 + 3 + 5|]
 9
```

Taking advantage of our quasiquoter, we can re-write our evaluator so it uses concrete syntax:

```
eval' :: Expr -> Integer
eval' [expr|$int:x|]   = x
eval' [expr|$x + $y|]  = eval' x + eval' y
eval' [expr|$x - $y|]  = eval' x - eval' y
eval' [expr|$x * $y|]  = eval' x * eval' y
eval' [expr|$x / $y|]  = eval' x `div` eval' y
```

Let's make sure it works as advertised:

```
> eval [expr|1 + 2 + 3|] == eval' [expr|1 + 2 + 3|]
 True
> eval [expr|1 + 3 * 5|] == eval' [expr|1 + 3 * 5|]
 True
```