Lecture 19

Template Haskell based implementation of `printf`

In the last lecture we saw a type class based solution to create functions that take variable number of arguments. Here we give a template haskell based solution.

19.1 What is Template Haskell ?

Template haskell is the Haskell way of doing Meta programming. At the very least one can use it like a macro substitution but it can be used to do much more. The idea is to process the Haskell code at compile time using Haskell itself. A programmer can write Haskell functions to manipulate Haskell code and using special syntax arrange the compiler to manipulate code at compile time. In this lecture we will see how to define a version of printf using template haskell.

Template Haskell consists of two important steps.

1. Quoting: To allow user defined function to manipulate the Haskell code one needs to represent the program as a value in some suitable data type. The data types defined in the module `Language.Haskell.TH.Syntax` is used for this purpose. For example the type `Exp` defined in the above module is used to represent a valid Haskell expression. Have a look into the documentation of that module.

2. Splicing. Once the Haskell language fragment is processed using various function defined by the user, it needs to be compiled by the compiler. This processing is called splicing.

One point to be noted though is that template haskell does not splice the code directly but only those that are expressions that are inside the quoting monad Q. This monad is required because while generate code various side effects are created. For example a variable `x` used in a fragment of the code has a different meaning if there is a local binding defined on it. Besides one would want to read in data from files (think of config files) to perform compile time operations.

There are two syantactic extensions to Haskell that makes template Haskell possible. If a haskell expression is written between `[|` and `|]`, the compiler will replace it with the corresponding representation in `Language.Haskell.TH.Syntax`. For example, the expression `[| "Hello" |]` is of type `Q Exp`. The corresponding `Exp` value is `LitE (StringL "Hello")`.

The following are the extra syntactic conventions used.

1. `[e| ... |]` or just `[| ... '|]'` for quoting expressions. This has type `Q Exp`.
2. `[d| ... |]` for quoting declarations. This has type `Q [Decl]`
3. `[t| ... |]` for quoting types. This has type `Q Type`.
4. `[p| ... |]` for quoting patterns. This has type `Q Pat`.

The splicing is done using the syntax `$(...)` (no space between `$` and `(`)

19.2 Some convenience types and combinators

The types `Q Exp` and `Q Pat` etc occur so often that there are aliases for them `ExpQ` and `PatQ` respectively. As an exercise guess the types aliases for `Q Type` and `Q Decl`.

Whenever possible it is better to make use of the `[| ... |]` notation to build quoted expressions. However sometimes it is better to use the constructors of `Exp` directly. Recall that we can splice only quoted expressions, i.e values of type Q Expr (or equivalently ExpQ). Say you have a quoted expressions `qe` and `qf` which correspondes to the haskell expression `e` and `f` respectively. If one wants to obtaine the quoted expression which correspondes to the application of the function `f` on the expression `e`, we would have to do something like the following

```
qfe = do e <- qe
         f <- qf
     return $ AppE f e.
```

To make this efficient there is a combinator called `appE` which does essentially what the constructor `AppE` does but works on `Q Exp` rather than `Exp`.

```
appE :: ExpQ -> ExpQ -> ExpQ
```

The above code will then look like `appE qe qf`. There are such monadic version of all the constructors of `Exp` available. Make use of it.

19.3 Printf

First note that we have enabled template Haskell using the compiler pragma given above. It should be the very first line of your source code. A unrecognised pragma is ignored by the compiler but sometimes a warning is issued.

```
> {-# LANGUAGE TemplateHaskell    #-}
> module Printf where
> import Data.List
> import Language.Haskell.TH
```

Few things about enabling the template haskell. Strictly speaking this module does not need TemplateHaskell, rather it can be written without using template Haskell. This is because all it does is define functions that process objects of type `Expr` or `ExprQ`. I have enabled it so as to write `appE [|show|]` instead of the more complicated. `appE (varE 'show)`

First let us capture the formating via a data type

```
> data Format = L String    -- ^ literal string
>             | S           -- ^ %s
>             | G           -- ^ %g generic type
>             deriving Show
```

We need a function that will parse a string and the give the corresponding list for format. The exact details are not really of interest as far as template haskell is concerned. See the end of the file for an implementation.

```
> format :: String -> [Format]
```

The printf function can then be defined as.

```
> printf  :: String -> ExpQ
> printfP :: [Format] -> ExpQ
> printf = printfP . format
```

We would rather implement the prinfP function. Let the list of formatting instructions be [f_1,..,f_n], then we want `prinfP [f_1,...,f_n]` when spliced to return the code.

```
\ x0  ... xm -> concat [e1,...,e_n]
```

Here e_i depends on the ith format instruction f_i. If f_i is a literal then it will just be a literal string. Otherwise it would be an appropriate variable. In our case it should be `xj` where `j` is the number of *non-literal*, i.e. `S` or `G`, formating instructions to the left of e_i. The number m is the total number of *non-literal* formatting instructions in the list [f_1,...,f_n] and should be less than or equal to n.

Suppose we know the number of variables to the left of a format instruction f_i, how do we generate the expression e_i ? The following function does that

```
> toExpQ :: Int         -- ^ formatting instructions to the left
>        -> Format      -- ^ The current spec.
>        -> (Int,ExpQ)  -- ^ The total number of non-literal instruction
>               -- to the left and the resulting expression.
> toExpQ i   (L s) = (i,string s)
> toExpQ i   S     = (i+1,varExp i)
> toExpQ i   G     = (i+1,showE $ varExp i)
```

Here we make use of the following helper Template haskell functions which we have defined subsequently.

```
> string  :: String -> ExpQ  -- ^ quote the string
> showE   :: ExpQ   -> ExpQ  -- ^ quoted showing
> varExp  :: Int    -> ExpQ  -- ^ quoted variable xi
> varPat  :: Int    -> PatQ  -- ^ quoted pattern xi
```

The printfP function then is simple. Recall that when spliced it should generate the expression

```
\ x0 ... xm -> concat [e1,...,e_n]‘
```

The complete definition is given below.

```
> printfP fmts = lamE args . appE conc $ listE es
>         where (nvars,es) = mapAccumL toExpQ 0 fmts
>               args       = map varPat [0 .. (nvars-1)]
>               conc       = [|concat|]
```

Here are the definition of the helper functions

```
> string   = litE . StringL
> varExp i = varE $ mkName ("x" ++ show i)
> varPat i = varP $ mkName ("x" ++ show i)
> showE    = appE [|show|]
```

We now come to the parsing of the format string. For simplicity of implementation if % occurs before an unknow format string it is treated as literal occurance.

```
> format ""             = []
> format ['%']          = [L "%"]
> format ('%':x:xs)
>        | x == 's'  = S : format xs
>        | x == 'g'  = G : format xs
>        | x == '%'  = L "%" : format xs
>        | otherwise = L ['%',x] : format xs
> format zs          = L x : format xs
>        where (x,xs) = span (/='%') zs
```

To try it out load it with ghci using the option -XTemplateHaskell. You need this option to tell ghci that it better expect template haskell stuff like Oxford bracket `[| |]` and splicing `$(...)`

```
$ ghci -XTemplateHaskell src/lectures/Template-Haskell-based-printf.lhs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 1] Compiling Printf           ( src/lectures/Template-Haskell-based-printf.lhs, interpreted
Ok, modules loaded: Printf.
```

```
*Printf> $(printf "%s is a string") "Hello"
"Hello is a string"
*Printf> $(printf "%s is a string %g is an int") "Hello" 10
"Hello is a string 10 is an int"
*Printf>
```

19.4 Exercise

1. Write a function that will optimise the format instructions by merging consecutive literal strings. Is there any point in optimising the format instructions?

2. Rewrite the printf module to not use any template haskell itself.