Template Haskell

CIS 194 Week 11 13 November 2014

- **Template Haskell** on the Haskell Wiki.
- **Template Haskell** in the GHC user manual
- The original **paper** on Template Haskell (somewhat outdated)

Template Haskell is a *compile-time meta-programming* facility for Haskell. Let's pull all that terminology apart:

- "Compile-time means" that Template Haskell "happens" as you compile. More concretely, Template Haskell code gets *run* when you *compile*. This means that you can run arbitrary programs *while compiling*.

- "Meta-programming" means that Template Haskell code produces Haskell code as output. We use Template Haskell to programmatically write parts of a program.

Template Haskell is a language extension, specific to GHC. Recall that there is a standard for Haskell, called Haskell 2010. (There was another one called Haskell 98.) Any feature in GHC that deviates from this standard is enabled only by a language extension, which must be specified in a file that uses the feature. (Well, almost any feature.)

So, we enable Template Haskell now:

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad ( replicateM )
import Data.Maybe    ( maybeToList )
```

TH (as we'll call it) also has a library associated with it:

```
import Language.Haskell.TH
```

Somewhat annoyingly (but out of necessity), TH enforces its so-called "stage restriction", which says that the only functions you can call at compile time *must* be written in other modules than the one being defined. This does make sense, because otherwise, you could write a recursive function which would be required to write part of itself. Thinking about such things quickly makes a Haskeller go (more) insane, so we forbid outright.

Thus, for these notes, we'll have a companion module `SpliceFunctions` that has the compile-time code.

However, so that we can see the definitions of the functions as you read this file, they are included here, too. We'll import the `SpliceFunctions` module qualified so that we can unambiguously refer to the `SpliceFunctions` functions in splices.

```
import qualified SpliceFunctions as SF
```

Splices

Template Haskell has two modes: splices and quotes. Let's start with *splices*. A splice is a chunk of TH code that produces some Haskell which becomes a part of your program. The type of an (expression) splice is `Q Exp`, where `Q` and `Exp` are types from the TH library.

Just to get an idea of where we're going, let's take a look at that `Exp` type:

```
*Main> :info Exp
data Exp
  = VarE Name
  | ConE Name
  | LitE Lit
  | AppE Exp Exp
  | InfixE (Maybe Exp) Exp (Maybe Exp)
  | UInfixE Exp Exp Exp
  | ParensE Exp
  | LamE [Pat] Exp
  | LamCaseE [Match]
  | TupE [Exp]
  | UnboxedTupE [Exp]
  | CondE Exp Exp Exp
  | MultiIfE [(Guard, Exp)]
  | LetE [Dec] Exp
  | CaseE Exp [Match]
  | DoE [Stmt]
  | CompE [Stmt]
  | ArithSeqE Range
  | ListE [Exp]
  | SigE Exp Type
  | RecConE Name [FieldExp]
  | RecUpdE Exp [FieldExp]
```

Most expression forms that you can write in Haskell can be encoded into an `Exp`, short for "expression", naturally. (Not all expressions fit into `Exp`. TH often lags behind proper Haskell in some of Haskell's more esoteric features.)

Now, what about `Q`?

```
*Main> :info Q
```

```
<snip>
newtype Q a = <snip>
instance Monad Q -- Defined in 'Language.Haskell.TH.Syntax'
<snip>
```

Oh – so `Q` is just a monad. `Q` is the monad that wraps computations that can be run in the GHC compiler, at compile time. We'll see more of the capabilities the `Q` monad grants as we proceed.

Now, let's see a proper example:

```
add5 :: Integer -> Q Exp
add5 n = return (AppE (AppE (VarE (mkName "+")) (LitE (IntegerL n)))
                      (LitE (IntegerL 5)))

eleven :: Integer
eleven = $( SF.add5 6 )
```

The `compileTimeAdd5` function produces an expression that applies a variable named "+" to the number provided and the number 5. (Recall that `a + b` really means `(+) a b`, which really means `(((+) a) b)`.) TH uses the type `Name` instead of `String` to store variable names. This is because there can be multiple variables of the same name (perhaps in different scopes), and TH needs to be able to distinguish among them. Happily, we have a function `mkName :: String -> Name` that we'll use often to build `Name`s from `String`s.

Then, in `eleven`, we see the use of a splice. A splice is written between `$(` and `)`. It must have type `Q Exp`. GHC takes the `Exp` that is produced and uses that as the code to put in place of the splice. You can see this in action by turning on the `-ddump-splices` option:

```
*Main> :set -ddump-splices
*Main> :reload
...
Splicing expression
    SF.add5 6 ======> (+) 6 5
```

What that output is saying is that the code `SF.add5 6` is run, producing the expression `(+) 6 5`, which is then what is used as the definition of `eleven`.

We can actually do even better: let's do the actual addition at compile time, which means we don't have to bother when the application is running.

```
compileTimeAdd5 :: Integer -> Q Exp
compileTimeAdd5 n = return (LitE (IntegerL (n + 5)))

eleven' :: Integer
eleven' = $( SF.compileTimeAdd5 7 )
```

```
*Main> :reload
...
```

```
Splicing expression
    SF.compileTimeAdd5 7 ======> 12
```

Here, note that the result of `SF.compileTimeAdd5 7` is just the number `12`. No addition needs to happen when our program is running! Cool!

However, adding 5 is a terrible example of what TH is good for. TH shines when you must write a chunk of code, but that code is very repetitive. Say, for example, that you have variables named `a1` through `a100`, and you want to put all of these into a list. It's easy with TH:

```
listOfAs :: Q Exp
listOfAs = return (ListE (map VarE [ mkName ('a' : show n) | n <- [1..100] ]))

{- Why do you think this is commented out??
bigList :: [Int]
bigList = $( SF.listOfAs )
-}
```

As a more practical example, let's consider the `liftM` family of functions, the first few I'll write out.

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f ma = do a <- ma
                return (f a)

liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f ma mb = do a <- ma
                    b <- mb
                    return (f a b)

liftM3 :: Monad m => (a -> b -> c -> d) -> m a -> m b -> m c -> m d
liftM3 f ma mb mc = do a <- ma
                       b <- mb
                       c <- mc
                       return (f a b c)
```

That's boring! We should produce those expressions programmatically.

Let's look at the `DoE` constructor for `Exp`, which we'll need to work with:

```
*Main> :info DoE
data Exp = ... | DoE [Stmt] | ...
```

Hm. Works with `Stmt`s.

```
*Main> :info Stmt
data Stmt
  = BindS Pat Exp | LetS [Dec] | NoBindS Exp | ParS [[Stmt]]
```

The part of `do` notation with the `<-` is a binding statement, or `BindS`. On the left of the arrow is a pattern, naturally:

```
*Main> :info Pat
data Pat
  = LitP Lit
  | VarP Name
  | TupP [Pat]
  | UnboxedTupP [Pat]
  | ConP Name [Pat]
  | InfixP Pat Name Pat
  | UInfixP Pat Name Pat
  | ParensP Pat
  | TildeP Pat
  | BangP Pat
  | AsP Name Pat
  | WildP
  | RecP Name [FieldPat]
  | ListP [Pat]
  | SigP Pat Type
  | ViewP Exp Pat
```

A variable pattern (what we'll need) is a `VarP`. We may have enough to go on now:

```haskell
-- | Produce the body of a @liftM@ implementation. The parameter is the
-- number of the @liftM@.
liftMBody :: Int -> Q Exp
liftMBody n = let m_names = take n [ mkName ('m' : [x]) | x <- ['a'..] ]
                  names   = take n [ mkName [x]         | x <- ['a'..] ]
                  binds   = zipWith mk_bind m_names names
                  ret     = NoBindS (AppE (VarE (mkName "return"))
                                          (mk_apps (VarE (mkName "f"))
                                                   (map VarE names)))

                  mk_bind :: Name -> Name -> Stmt
                  mk_bind m_name name = BindS (VarP name) (VarE m_name)

                  -- apply one expression to a list
                  mk_apps :: Exp -> [Exp] -> Exp
                  mk_apps f []     = f
                  mk_apps f (x:xs) = mk_apps (AppE f x) xs

                  -- or
                  -- mk_apps = foldl AppE
              in
              return $ LamE (map VarP m_names) (DoE (binds ++ [ret]))
```

```haskell
liftM4 :: Monad m => (a -> b -> c -> d -> e)
                  -> m a -> m b -> m c -> m d -> m e
liftM4 f = $( SF.liftMBody 4 )
```

If it type-checks, it must be right! :)

We can do even better, though. TH works not only for expressions, but also for types. If we use a splice in a context where a type is expected, it should have the type `Q Type`, and GHC will splice in a type instead of an expression.

```
*Main> :i Type
data Type
  = ForallT [TyVarBndr] Cxt Type
  | AppT Type Type
  | SigT Type Kind
  | VarT Name
  | ConT Name
  | PromotedT Name
  | TupleT Int
  | UnboxedTupleT Int
  | ArrowT
  | ListT
  | PromotedTupleT Int
  | PromotedNilT
  | PromotedConsT
  | StarT
  | ConstraintT
  | LitT TyLit
```

Using TH at the type level, we can reduce even more boilerplate.

Before we go on, we'll start to use one more little bit of TH syntax: we can quote known names. For example, the `mkName "return"` above is a little silly. Instead, we can write `'return`:

```haskell
returnName :: Name
returnName = 'return
```

To quote a type in the same way, use two quote marks.

```haskell
liftMType :: Int -> Q Type
liftMType n = let names = take n [ mkName [x] | x <- ['a'..] ]
                  types = map VarT names
                  m     = mkName "m"
                  res   = mkName "r"
                  resty = VarT res
```

```haskell
              -- make nested arrows
              mk_arrs :: [Type] -> Type -> Type
              mk_arrs []     result = result
              mk_arrs (x:xs) result = AppT (AppT ArrowT x)
                                           (mk_arrs xs result)

              -- apply "m" to a type
              app_m :: Type -> Type
              app_m = AppT (VarT m)
          in
          return $ ForallT (PlainTV m : PlainTV res :
                             map PlainTV names)
                            [ClassP ''Monad [VarT m]]
                            (mk_arrs (mk_arrs types resty : map app_m types)
                                     (app_m resty))

liftM5 :: $( SF.liftMType 5 )
liftM5 f = $( SF.liftMBody 5 )
```

But disaster strikes at `liftM6`!

```haskell
{-
liftM6 :: $( SF.liftMType 6 )
liftM6 f = $( SF.liftMBody 6 )
-}
```

The problem is that we have not been *hygienic*. (That's a technical term!) When creating names in `liftMBody` and `liftMType`, we've just used single-letter names, without worrying if these names are already in use. Well, `f` is in use, and so we fall over when we try to use it.

We need a way of creating a name guaranteed to be *fresh*. Happily, the `Q` monad has just such a facility – the `newName` function:

```haskell
newName :: String -> Q Name
```

This function is guaranteed always to return a fresh name. The string passed in is just a suggestion for the name. You could always pass the same string in but get different names each time.

```haskell
-- | Produce the body of a @liftM@ implementation. The parameter is the
-- number of the @liftM@.
liftMBody' :: Int -> Q Exp
liftMBody' n = do
  m_names <- replicateM n (newName "m")
  names   <- replicateM n (newName "a")
  let binds   = zipWith mk_bind m_names names
      ret     = NoBindS (AppE (VarE 'return)
                              (mk_apps (VarE (mkName "f"))
```

```
                                 (map VarE names)))

      mk_bind :: Name -> Name -> Stmt
      mk_bind m_name name = BindS (VarP name) (VarE m_name)

      mk_apps :: Exp -> [Exp] -> Exp
      mk_apps = foldl AppE
  return $ LamE (map VarP m_names) (DoE (binds ++ [ret]))

liftM6 :: $( SF.liftMType 6 )
liftM6 f = $( SF.liftMBody' 6 )
```

Note that we still use `mkName` to grab `f`. That's because `liftMBody` is expecting that `f` is bound outside of the TH code. We really *do* want to be unhygienic here, so `mkName` is the right choice.

Quotations

TH is more than just a facility for optimizing compile-time calculations. TH also provides a syntax for *quoting*, where GHC will take an expression you write and convert it into data of the TH `Exp` type. You write a quotation between `[|` and `|]`. A quotation conveniently has the type `Q Exp`. You can even embed splices within quotations!

To continue our `liftM` example, we'll actually use a type quotation, as expression quotations don't really help us here. A type quotation is in brackets written `[t|` and `|]` and has type `Q Type`. Let's see one in action:

```
liftMType' :: Int -> Q Type
liftMType' n = do
  names <- replicateM n (newName "a")
  m     <- newName "m"
  res   <- newName "r"

  let types = map varT names
      mty   = varT m
      resty = varT res

      -- make nested arrows
      mk_arrs :: [Q Type] -> Q Type -> Q Type
      mk_arrs []     result = result
      mk_arrs (x:xs) result = [t| $x -> $(mk_arrs xs result) |]

      -- apply "m" to a type
      app_m :: Q Type -> Q Type
      app_m ty = [t| $mty $ty |]

  forallT (map PlainTV (m : res : names)) (cxt [])
         [t| Monad $mty => $(mk_arrs types resty) ->
```

```
                                           $(mk_arrs (map app_m types) (app_m resty)) |]

liftM7 :: $( SF.liftMType' 7 )
liftM7 f = $( SF.liftMBody' 7 )
```

Using quotations made that definition a little less hairy. One thing to notice here is that we're using lower-case functions like `varT` instead of uppercase data constructors like `VarT`. `varT` has type `Name -> Q Type` as opposed to `VarT` with type `Name -> Type`. Having things wrapped up in the `Q` monad helps us out when we're using quotations. To see why, just follow the types! Remember that anything following a `$` must be in the `Q` monad.

Reification

The `Q` monad provides another very important facility: the ability to inspect predefined types and functions and see their definitions, not unlike the `:info` command in GHCi. The key function here is `reify :: Name -> Q Info`, where `Info` is like this:

```
*Main> :info Info
data Info
  = ClassI Dec [InstanceDec]
  | ClassOpI Name Type ParentName Fixity
  | TyConI Dec
  | FamilyI Dec [InstanceDec]
  | PrimTyConI Name Arity Unlifted
  | DataConI Name Type ParentName Fixity
  | VarI Name Type (Maybe Dec) Fixity
  | TyVarI Name Type
```

To inspect the output from `reify`, it's helpful to use yet another capability of the `Q` monad: to use arbitrary I/O. TH exports the function `runIO :: IO a -> Q a`, which allows any `IO` operation in the `Q` monad. This is indeed very powerful, and we'll discuss some ramifications later. For now, it just means that we can print in the middle of a splice.

```
$( do info <- reify ''Bool
      runIO $ print info
      return [] )
```

Compiling the splice above gives us

```
TyConI (DataD [] GHC.Types.Bool [] [NormalC GHC.Types.False [],NormalC GHC.Types.True []] [])
```

What's going on here? First, we have a top-level declaration splice. We've seen expression splices and type splices, but this is our first declaration splice. Like the others, a declaration splice allows you to run arbitrary code (in this case, of type `Q [Dec]`) and splices the result into your program. Because `Q` is a monad, we can use `do` notation inside of a splice. Our first action is to call `reify` on the type `Bool`, storing the result in the variable `info`. We then embed a I/O action in the `Q` monad with `runIO`. (Recall that `print :: Show a =>`

a -> IO ().) Finally, we must result in a [Dec] – a list of Haskell declarations – but in this case we don't have any declarations to splice, so we just return the empty list.

Looking at the output, we see a TyConI wrapping a DataD. DataD is one of the constructors of the Dec type.

```
*Main> :info Dec
data Dec
  = FunD Name [Clause]
  | ValD Pat Body [Dec]
  | DataD Cxt Name [TyVarBndr] [Con] [Name]
  | NewtypeD Cxt Name [TyVarBndr] Con [Name]
  | TySynD Name [TyVarBndr] Type
  | ClassD Cxt Name [TyVarBndr] [FunDep] [Dec]
  | InstanceD Cxt Type [Dec]
  | SigD Name Type
  | ForeignD Foreign
  | InfixD Fixity Name
  | PragmaD Pragma
  | FamilyD FamFlavour Name [TyVarBndr] (Maybe Kind)
  | DataInstD Cxt Name [Type] [Con] [Name]
  | NewtypeInstD Cxt Name [Type] Con [Name]
  | TySynInstD Name TySynEqn
  | ClosedTypeFamilyD Name [TyVarBndr] (Maybe Kind) [TySynEqn]
  | RoleAnnotD Name [Role]
```

A DataD encodes a datatype declaration. The info just tells us that the definition of Bool is like data Bool = False | True.

Larger example

Template Haskell can be very useful for defining boiler-plate instances. For example, consider the following class:

```haskell
class Sizable a where
  size :: a -> Int
  size _ = 1
```

Any member of this class has a known size, with a default of 1. We can define instances for some basic types, all of size 1:

```haskell
instance Sizable Int
instance Sizable Integer
instance Sizable Bool
instance Sizable Char
```

We could go further and start writing Sizable instances for other types, but the code would be *very* boring.

Instead, let's write TH functions to do the work for us.

```
mapMaybeM :: Monad m => (a -> m (Maybe b)) -> [a] -> m [b]
mapMaybeM _ [] = return []
mapMaybeM f (x:xs) = do
  maybe_b <- f x
  bs      <- mapMaybeM f xs
  return $ maybeToList maybe_b ++ bs

deriveSizable :: [Name] -> Q [Dec]  -- type is suitable for declaration splice
deriveSizable = mapMaybeM deriveSizable1

deriveSizable1 :: Name -> Q (Maybe Dec)
deriveSizable1 name = do
  info <- reify name
  case info of
    TyConI (DataD _ _name tvbs cons _)   ->
      Just `liftM` deriveSizableData name tvbs cons
    TyConI (NewtypeD _ _name tvbs con _) ->
      Just `liftM` deriveSizableData name tvbs [con]
    _                                    -> do
      reportError $ show name ++ " is not a datatype"
      return Nothing

deriveSizableData :: Name -> [TyVarBndr] -> [Con] -> Q Dec
deriveSizableData name tvbs cons = do
  clauses <- mapM deriveSizableClause cons
  return $ InstanceD context
                     (AppT (ConT ''Sizable)
                      (foldl AppT (ConT name) tvb_tys))
                     [FunD 'size clauses]
  where
    tvb_names = map getTvbName tvbs
    tvb_tys   = map VarT tvb_names
    context   = map (ClassP ''Sizable . (:[]) . VarT) tvb_names

    getTvbName (PlainTV n)    = n
    getTvbName (KindedTV n _) = n

deriveSizableClause :: Con -> Q Clause
deriveSizableClause (NormalC con_name stys) = do
  field_names <- mapM (const $ newName "x") stys
  clause [conP con_name (map varP field_names)]
         (normalB [| 1 + sum $(
           listE (map (\x -> [| size $(varE x) |]) field_names)
         ) |])
```

```
          []
deriveSizableClause (RecC con_name vstys)
  = let stys = map strip_fst vstys in
    deriveSizableClause (NormalC con_name stys)
  where
    strip_fst (_,b,c) = (b,c)
deriveSizableClause (InfixC sty1 con_name sty2)
  = deriveSizableClause (NormalC con_name [sty1, sty2])
deriveSizableClause (ForallC _ _ con) = deriveSizableClause con
```

Now, having done all of that, it is easy to generate new `Sizable` instances:

```
$( SF.deriveSizable [''Maybe, ''[], ''Either] )

trySize :: Int
trySize = SF.size (Just (Just "abc"))
```

This “deriving” use-case of TH is rather prevalent, so much so that GHC allows you to drop the `$`:

```
SF.deriveSizable [''Ordering]
```

`Generated 2014-12-04 13:37:45.160867`

Powered by **shake**, **hakyll**, **pandoc**, **diagrams**, and **lhs2TeX**.