# Template Haskell

**From HaskellWiki**

**Template Haskell (http://hackage.haskell.org/package/template-haskell)** is a GHC extension to Haskell that adds compile-time metaprogramming facilities. The original design can be found here: http://research.microsoft.com/en-us/um/people/simonpj/papers/meta-haskell/. It is included (http://www.haskell.org/ghc/docs/latest/html/users_guide/template-haskell.html) in GHC since version 6.

This page hopes to be a more central and organized repository of TH related things.

## Contents

## 1 What is Template Haskell?

Template Haskell is an extension to Haskell 98 that allows you to do type-safe compile-time meta-programming, with Haskell both as the manipulating language and the language being manipulated.

Intuitively Template Haskell provides new language features that allow us to convert back and forth between concrete syntax, i.e. what you would type when you write normal Haskell code, and abstract syntax trees. These abstract syntax trees are represented using Haskell datatypes and, at compile time, they can be manipulated by Haskell code. This allows you to reify (convert from concrete syntax to an abstract syntax tree) some code, transform it and splice it back in (convert back again), or even to produce completely new code and splice that in, while the compiler is compiling your module.

For email about Template Haskell, use the GHC users mailing list (http://www.haskell.org/mailman/listinfo/glasgow-haskell-users) . It's worth joining if you start to use TH.

## 2 Template Haskell specification

Template Haskell is only documented rather informally at the moment. Here are the main resources:

- The user manual section on Template Haskell (http://www.haskell.org/ghc/docs/latest/html/users_guide/template-haskell.html)
- The user manual section on quasi-quotation (http://www.haskell.org/ghc/docs/latest/html/users_guide/template-haskell.html#th-quasiquotation) , which is closely related to Template Haskell.
- The original Template Haskell paper (http://research.microsoft.com/~simonpj/papers/meta-haskell/)
- Notes on Template Haskell version 2 (http://research.microsoft.com/en-us/um/people/simonpj/tmp/notes2.ps) , which describes changes since the original paper. Section 8 describes the difficulty with pattern splices, which are therefore not implemented.
- The Template Haskell API (http://hackage.haskell.org/package/template-haskell)

## 3 Template Haskell tutorials and papers

- Bulat's tutorials:
  1. Wayback Machine (http://web.archive.org/web/20100703060856/http://www.haskell.org/bz/thdoc.htm) , Google Docs (http://docs.google.com/uc?id=0B4BgTwf_ng_TM2MxZjJjZjctMTQ0OS00YzcwLWE5N2QtMDI0YzE4NGUwZDM3)
  2. Wayback Machine (http://web.archive.org/web/20100703060841/http://www.haskell.org/bz/th3.htm) , Google Docs (http://docs.google.com/uc?id=0B4BgTwf_ng_TOGJkZjM4ZTUtNGY5My00ZThhLTllNDQtYzJjMWJiMzJhZjNj)

  One reader said "These docs are *brilliant* ! Exactly what I need to get an understanding of TH."

(Note: These documents are from the Wayback machine (http://www.archive.org) because the originals disappeared. They're public documents on Google docs, which shouldn't require logging in. However, if you're asked to sign in to view them, you're running into a known Google bug. You can fix it by browsing to Google (http://www.google.com) , presumably gaining a cookie in the process.)

- Dominik's example-driven, practical introduction to Template Haskell.

- A very short tutorial to understand the basics in 10 Minutes.

    - http://www.hyperedsoftware.com/blog/entries/first-stab-th.html

- GHC Template Haskell documentation
    - http://www.haskell.org/ghc/docs/latest/html/users_guide/template-haskell.html

- Papers about Template Haskell

-
    - Template metaprogramming for Haskell, by Tim Sheard and Simon Peyton Jones, Oct 2002. [ps (http://www.haskell.org/wikiupload/c/ca/Meta-haskell.ps) ]
    - Template Haskell: A Report From The Field, by Ian Lynagh, May 2003. [ps (http://www.haskell.org/wikiupload/2/24/Template_Haskell-A_Report_From_The_Field.ps) ]
    - Unrolling and Simplifying Expressions with Template Haskell, by Ian Lynagh, December 2002. [ps (http://www.haskell.org/wikiupload/e/ed/Template-Haskell-Utils.ps) ]
    - Automatic skeletons in Template Haskell, by Kevin Hammond, Jost Berthold and Rita Loogen, June 2003. [pdf (http://www.haskell.org/wikiupload/6/69/AutoSkelPPL03.pdf) ]
    - Optimising Embedded DSLs using Template Haskell, by Sean Seefried, Manuel Chakravarty, Gabriele Keller, March 2004. [pdf (http://www.haskell.org/wikiupload/b/b5/Seefried04th-pan.pdf) ]
    - Typing Template Haskell: Soft Types, by Ian Lynagh, August 2004. [ps (http://www.haskell.org/wikiupload/7/72/Typing_Template_Haskell_Soft_Types.ps) ]

## 4 Other useful resources

- (2011) Basic Tutorial of Template Haskell (https://github.com/leonidas/codeblog/blob/master/2011/2011-12-27-template-haskell.md)

- (2011) Greg Weber's blog post on Template Haskell and quasi-quoting (http://www.yesodweb.com/blog/2011/10/code-generation-conversation) in the context of Yesod.

- (2012) Mike Ledger's tutorial on TemplateHaskell and QuasiQuotation (http://quasimal.com/posts/2012-05-25-quasitext-and-quasiquoting.html) for making an interpolated text QuasiQuoter. Here's another great 2014 blog post on quasiquotation (http://www.well-typed.com/blog/2014/10/quasi-quoting-dsls/) .

- Fraskell documentation (http://www.cs.ox.ac.uk/people/ian.lynagh/Fraskell/) & explanation of how Template Haskell is used to vastly speed it up.

- Quasiquotation

Feel free to update our Wikipedia entry http://en.wikipedia.org/wiki/Template_Haskell

## 5 Projects

What are you doing/planning to do/have done with Template Haskell?

- The ForSyDe methodology (http://www.ict.kth.se/org/ict/ecs/sam/projects/forsyde/www) is currently implemented as a Haskell-based DSL which makes extensive use of Template Haskell.

- I have written a primitive (untyped) binding to the Objective-C runtime system on Mac OS X. It needs just TH, no "stub files" are created, no separate utilities are required. Initial snapshot is at http://www.kfunigraz.ac.at/imawww/thaller/wolfgang/HOC020103.tar.bz2 -- WolfgangThaller

- I am writing Template Greencard - a reimplementation of GreenCard using TH. Many bits work out really nicely. A few bits didn't work so nicely - once I get some time to think, I'll try to persuade the TH folk to make some changes to fix some of these. -- AlastairReid
- I'm writing Hacanon - a framework for automatic generation of C++ bindings. Read "automated Template Greencard for C++" (-: Darcs repo: http://www.ScannedInAvian.org /repos/hacanon - You'll need gccxml (http://www.gccxml.org/) to compile the examples. - 27 Dec Lemmih.
- Following other FFI tools developers, I see some future for Template HSFFIG, especially when it comes to autogenerate peek and poke methods for structures defined in C; may be useful for implementation of certain network protocols such as X11 where layout of messages is provided as C structure/union declaration. - 16 Dec 2005 DimitryGolubovsky
- I am using Template Haskell as a mechanism to get parsed, typechecked code into an Ajax based Haskell Equational Reasoning tool Haskell Equational Reasoning Assistant, as well as simplify the specification of equational relationships between pieces of code. There was a quicktime movie of the tool being used on http://www.gill-warbington.com/home/andy/share /hera1.html - AndyGill
- I am working on functional metaprogramming techniques to enhance programming reliability and productivity, by reusing much of the existing compiler technology. Template Haskell is especially interesting for me because it permits to check size information of structures by the compiler, provided this information is available at compile time. This approach is especially appropriate for hardware designs, where the structures are fixed before the circuit starts operating. See our metaprogramming web page at http://www.infosun.fmi.uni-passau.de /cl/metaprog/ -- ChristophHerrmann(http://www.cs.st-and.ac.uk/~ch)
- I am using Template Haskell to do type safe database access. I initially proposed this on haskell-cafe (http://www.nabble.com/Using-Template-Haskell-to-make-type-safe-database-access-td17027286.html) . I connect to the database at compile-time and let the database do SQL parsing and type inference. The result from parsing and type inference is used to build a type safe database query which can executed at run-time. You can find the project page here -- Mads Lindstrøm (mailto:mads_lindstroem@yahoo.dk)

## 6 Utilities

Helper functions, debugging functions, or more involved code e.g. a monadic fold algebra for TH.Syntax.

- http://www.haskell.org/pipermail/template-haskell/2003-September/000176.html

## 7 Known Bugs

Take a look at the open bugs against Template Haskell (http://hackage.haskell.org/trac/ghc /query?status=new&status=assigned&status=reopened&component=Template+Haskell& order=priority) on the GHC bug tracker.

## 8 Wish list

Things that Ian Lynagh (Igloo) mentioned in his paper *Template Haskell: A Report From The Field* in May 2003 (available here (http://www.haskell.org/wikiupload/2/24/Template_Haskell-A_Report_From_The_Field.ps) ), by section:

- Section 2 (curses)
  - The ability to splice names (into "foreign import" declarations, in particular)
  - The ability to add things to the export list from a splice(?)
  - The ability to use things defined at the toplevel of a module from splices in that same module (would require multi-stage compilation, as opposed to the current approach of expanding splices during typechecking)

- Section 3 (deriving instances of classes)
  - ~~First-class reification~~ (the `reify` function)
  - A way to discover whether a data constructor was defined infix or prefix (which is necessary to derive instances for `Read` and `Show` as outlined in The Haskell 98 Report: Specification of Derived Instances (http://www.haskell.org/onlinereport/derived.html) ) (if there is a way, Derive (http://community.haskell.org/~ndm/derive/) seems ignorant of it)
  - Type/context splicing (in `instance` headers in particular)

- Section 4 (printf)
  - He says something to the effect that a pattern-matching form of the quotation brackets would be cool if it was expressive enough to be useful, but that this would be hard. (Don't expect this anytime soon.)

- Section 5 (fraskell)
  - Type information for quoted code (so that simplification can be done safely even with overloaded operations, like, oh, `(+)` )

- Section 6 (pan)
  - Type info again, and strictness info too (this one seems a bit pie-in-the-sky...)

(Please leave the implemented ones here, but crossed off.)

Any other features that may be nice, and TH projects you'd want to see.

- A TH tutorial (mainly a distillation and update of *Template Meta-programming in Haskell* at this point)
- ~~Write Haddock documentation for the Template Haskell library (http://hackage.haskell.org/trac/ghc/ticket/1576).~~
- Make `reify` on a class return a list of the instances of that class (http://www.haskell.org/pipermail/template-haskell/2005-December/000503.html). (See also GHC ticket #1577 (http://hackage.haskell.org/trac/ghc/ticket/1577) .)
- A set of simple examples on this wiki page
- A TH T-shirt with new logo to wear at conferences
- (Long-term) Unify Language.Haskell.Syntax with Language.Haskell.TH.Syntax so there's just one way to do things (http://hackage.haskell.org/package/haskell-src-meta does a one-way translation, for haskell-src-exts)

# 9 Tips and Tricks

## 9.1 What to do when you can't splice that there

When you try to splice something into the middle of a template and find that you just can't, instead of getting frustrated about it, why not use the template to see what it would look like in longhand?

First, an excerpt from a module of my own. I, by the way, am SamB.

```
{-# OPTIONS_GHC -fglasgow-exts -fth #-}

module MMixMemory where

import Data.Int
import Data.Word

class (Integral int, Integral word)
    => SignConversion int word | int -> word, word -> int where

    toSigned   :: word -> int
    toSigned   = fromIntegral
    toUnsigned :: int -> word
    toUnsigned = fromIntegral
```

Say I want to find out what I need to do to splice in the types for an instance declaration for the SignConversion class, so that I can declare instances for Int8 with Word8 through Int64 with Word64. So, I start up good-ol' GHCi and do the following:

```
$ ghci -fth -fglasgow-exts
Prelude> :l MMixMemory
*MMixMemory> :m +Language.Haskell.TH.Syntax
*MMixMemory Language.Haskell.TH.Syntax> runQ [d| instance SignConversion Int Word where |] >>= print
[InstanceD [] (AppT (AppT (ConT MMixMemory.SignConversion) (ConT GHC.Base.Int)) (ConT GHC.Word.Word)) []]
```

## 9.2 What can `reify` see?

When you use `reify` to give you information about a `Name`, GHC will tell you what it knows. But sometimes it doesn't know stuff. In particular

- **Imported things**. When you reify an imported function, type constructor, class, etc, from (say) module M, GHC runs off to the interface file `M.hi` in which it deposited all the info it learned when compiling M. However, if you compiled M without optimisation (ie `-O0`, the default), and without `-XTemplateHaskell`, GHC tries to put as little info in the interface file as possible. (This is a possibly-misguided attempt to keep interface files small.) In particular, it may dump only the name and kind of a data type into `M.hi`, but not its constructors.

  Under these circumstances you may reify a data type but get back no information about its data constructors or fields. Solution: compile M with
  - `-O`, or
  - `-fno-omit-interface-pragmas` (implied by -O), or
  - `-XTemplateHaskell`.

- **Function definitions**. The `VarI` constructor of the `Info` type advertises that you might get back the source code for a function definition. In fact, GHC currently (7.4) *always* returns `Nothing` in this field. It's a bit awkward and no one has really needed it.

## 9.3 Why does `runQ` crash if I try to reify something?

This program will fail with an error message when you run it:

```
  main = do info <- runQ (reify (mkName "Bool")) -- more hygenic is: (reify ''Bool)
            putStrLn (pprint info)
```

Reason: `reify` consults the type environment, and that is not available at run-time. The type of `reify` is

```
  reify :: Quasi m => Q a -> m a
```

The IO monad is a poor-man's instance of `Quasi`; it can allocate unique names and gather error messages, but it can't do `reify`. This error should really be caught statically.

Instead, you can run the splice directly (ex. in ghci -XTemplateHaskell), as the following shows:

```
GHCi> let tup = $(tupE $ take 4 $ cycle [ [| "hi" |] , [| 5 |] ])
GHCi> :type tup
tup :: ([Char], Integer, [Char], Integer)

GHCi> tup
("hi",5,"hi",5)

GHCi> $(stringE . show =<< reify ''Int)
"TyConI (DataD [] GHC.Types.Int [] [NormalC GHC.Types.I# [(NotStrict,ConT GHC.Prim.Int#)]] [])"
```

Here's an email thread with more details (http://www.haskell.org/pipermail/glasgow-haskell-users/2006-August/010844.html) .

# 10 Examples

## 10.1 Tuples

### 10.1.1 Select from a tuple

An example to select an element from a tuple of arbitrary size. Taken from this paper (http://research.microsoft.com/en-us/um/people/simonpj/papers/meta-haskell/meta-haskell.pdf) .

Use like so:

```
> $(sel 2 3) ('a','b','c')
'b'
> $(sel 3 4) ('a','b','c','d')
'c'
```

```
sel :: Int -> Int -> ExpQ
sel i n = [| \x -> $(caseE [| x |] [alt]) |]
    where alt :: MatchQ
          alt = match pat (normalB rhs) []

          pat :: Pat
          pat = tupP (map varP as)

          rhs :: ExpQ
          rhs = varE(as !! (i -1)) -- !! is 0 based

          as :: [String]
          as = ["a" ++ show i | i <- [1..n] ]
```

Alternately:

```
sel' i n = lamE [pat] rhs
    where pat = tupP (map varP as)
          rhs = varE (as !! (i - 1))
          as  = [ "a" ++ show j | j <- [1..n] ]
```

### 10.1.2 Apply a function to the n'th element

```
tmap i n = do
    f <- newName "f"
    as <- replicateM n (newName "a")
    lamE [varP f, tupP (map varP as)] $
        tupE [  if i == i'
                    then [| $(varE f) $a |]
                    else a
               | (a,i') <- map varE as `zip` [1..] ]
```

Then tmap can be called as:

```
> $(tmap 3 4) (+ 1) (1,2,3,4)
(1,2,4,4)
```

### 10.1.3 Convert the first n elements of a list to a tuple

This example creates a tuple by extracting elements from a list. Taken from www.xoltar.org (http://www.xoltar.org/2003/aug/13/templateHaskellTupleSample.html)

Use like so:

```
> $(tuple 3) [1,2,3,4,5]
(1,2,3)
> $(tuple 2) [1,2]
(1,2)
```

```
tuple :: Int -> ExpQ
tuple n = [|\list -> $(tupE (exprs [|list|])) |]
  where
    exprs list = [infixE (Just (list))
                         (varE "!!")
                         (Just (litE $ integerL (toInteger num)))
                     | num <- [0..(n - 1)]]
```

An alternative that has more informative errors (a failing pattern match failures give an exact location):

```
tuple :: Int -> ExpQ
tuple n = do
    ns <- replicateM n (newName "x")
    lamE [foldr (\x y -> conP '(:) [varP x,y]) wildP ns] (tupE $ map varE ns)
```

### 10.1.4 Un-nest tuples

Convert nested tuples like (a,(b,(c,()))) into (a,b,c) given the length to generate.

```
unNest n = do
    vs <- replicateM n (newName "x")
    lamE [foldr (\a b -> tupP [varP a , b])
                (conP '() [])
                vs]
         (tupE (map varE vs))
```

## 10.2 Marshall a datatype to and from Dynamic

This approach is an example of using template haskell to delay typechecking to be able to abstract out the repeated calls to fromDynamic:

```haskell
data T = T Int String Double

toT :: [Dynamic] -> Maybe T
toT [a,b,c] = do
    a' <- fromDynamic a
    b' <- fromDynamic b
    c' <- fromDynamic c
    return (T a' b' c')
toT _ = Nothing
```

## 10.3 Printf

Build it using a command similar to:

```
ghc --make Main.hs -o main
```

Main.hs:

```haskell
{-# LANGUAGE TemplateHaskell #-}

-- Import our template "printf"
import PrintF (printf)

-- The splice operator $ takes the Haskell source code
-- generated at compile time by "printf" and splices it into
-- the argument of "putStrLn".
main = do
    putStrLn $ $(printf "Hello %s %%x%% %d %%x%%") "World" 12
```

PrintF.hs:

```haskell
{-# LANGUAGE TemplateHaskell #-}
module PrintF where

-- NB: printf needs to be in a separate module to the one where
-- you intend to use it.

-- Import some Template Haskell syntax
import Language.Haskell.TH

-- Possible string tokens: %d %s and literal strings
data Format = D | S | L String
    deriving Show

-- a poor man's tokenizer
tokenize :: String -> [Format]
tokenize [] = []
tokenize ('%':c:rest) | c == 'd' = D : tokenize rest
                      | c == 's' = S : tokenize rest
tokenize (s:str) = L (s:p) : tokenize rest -- so we don't get stuck on weird '%'
    where (p,rest) = span (/= '%') str

-- generate argument list for the function
args :: [Format] -> [PatQ]
args fmt = concatMap (\(f,n) -> case f of
                                  L _ -> []
                                  _   -> [varP n]) $ zip fmt names
    where names = [ mkName $ 'x' : show i | i <- [0..] ]

-- generate body of the function
body :: [Format] -> ExpQ
body fmt = foldr (\ e e' -> infixApp e [| (++) |] e') (last exps) (init exps)
    where exps = [ case f of
                    L s -> stringE s
                    D   -> appE [| show |] (varE n)
                    S   -> varE n
                 | (f,n) <- zip fmt names ]
          names = [ mkName $ 'x' : show i | i <- [0..] ]
```

```haskell
-- glue the argument list and body together into a lambda
-- this is what gets spliced into the haskell code at the call
-- site of "printf"
printf :: String -> Q Exp
printf format = lamE (args fmt) (body fmt)
    where fmt = tokenize format
```

## 10.4 Handling Options with Templates

A common idiom for treating a set of options, e.g. from GetOpt, is to define a datatype with all the flags and using a list over this datatype:

```haskell
data Options = B1 | B2 | V Integer

options = [B1, V 3]
```

While it's simple testing if a Boolean flag is set (simply use "elem"), it's harder to check if an option with an argument is set. It's even more tedious writing helper-functions to obtain the value from such an option since you have to explicitely "un-V" each. Here, Template Haskell can be (ab)used to reduce this a bit. The following example provides the module "OptionsTH" which can be reused regardless of the constructors in "Options". Let's start with showing how we'd like to be able to program. Notice that the resulting lists need some more treatment e.g. through "foldl".

Options.hs:

```haskell
module Main where

import OptionsTH
import Language.Haskell.TH.Syntax

data Options = B1 | B2 | V Int | S String deriving (Eq, Read, Show)

options = [B1, V 3]

main = do
  print foo -- test if B1 set:               [True,False]
  print bar -- test if V present, w/o value: [False,True]
  print baz -- get value of V if available:  [Nothing,Just 3]

foo :: [Bool]
-- Query constructor B1 which takes no arguments
foo = map $(getopt (THNoArg (mkArg "B1" 0))) options

bar :: [Bool]
-- V is a unary constructor. Let mkArg generate the required
-- wildcard-pattern "V _".
bar = map $(getopt (THNoArg (mkArg "V" 1))) options

-- Can't use a wildcard here!
baz :: [(Maybe Int)]
baz = map $(getopt (THArg (conP "V" [varP "x"]))) options
```

OptionsTH.hs

```haskell
module OptionsTH where

import Language.Haskell.TH.Syntax

-- datatype for querying options:
-- NoArg: Not interested in value (also applies to Boolean flags)
-- Arg:   Grep value of unary(!) constructor
data Args = THNoArg Pat | THArg Pat

getopt :: Args -> ExpQ
getopt (THNoArg pat) = lamE [varP "y"] (caseE (varE "y") [cons0, cons1])
 where
  cons0 = match pat   (normalB [| True  |]) []
  cons1 = match wildP (normalB [| False |]) []

-- bind "var" for later use!
getopt (THArg pat@(ConP _ [VarP var])) = lamE [varP "y"] (caseE (varE "y") [cons0, cons1])
 where
  cons0 = match pat   (normalB (appE [|Just|] (varE var))) []
  cons1 = match wildP (normalB [|Nothing|]) []

mkArg :: String -> Int -> Pat
mkArg k c = conP k (replicate c wildP)
```

While the example might look contrived for the Boolean options which could have been tested much easier, it shows how both types of arguments can be treated in a similar way.

### 10.4.1 Limitations

`getopt (THArg pat)` is only able to treat unary constructors. See the pattern-binding: It matches exactly a single VarP.

### 10.4.2 Improvements

The following reduces things even a bit more, though I still don't know if I like it. It only works since `c` is either 0 or 1.

```
mkArg k c = conP k (replicate c (varP "x"))

baz = map $(getopt (THArg (mkArg "V" 1)))
```

-- VolkerStolz

## 10.5 Generic constructor for records

I have a large number of record types like this, of different length:

```
data PGD = PGD {
    pgdXUnitBase           :: !Word8,
    pgdYUnitBase           :: !Word8,
    pgdXLUnitsperUnitBase  :: !Word16
}
```

Currently I use GHC's Binary module to read them from files; it can handle types like `(Word8, (Word8, Word16))`, but there was no easy way to generate the correct amount of "uncurry" calls for automatically grabbing each element.

With Template Haskell, the instance declarations are now written as such:

```
instance Binary PGD where
    get bh = do a <- get bh ; return $ $(constrRecord PGD) a
```

Here the trick lies in constrRecord, which is defined as:

```
constrRecord x = reify exp where
    reify   = \(Just r) -> appE r $ conE $ last args
    exp     = foldl (dot) uncur $ replicate terms uncur
    terms   = ((length args) `div` 2) - 2
    dot x y = (Just $ infixE x (varE ".") y)
    uncur   = (Just [|uncurry|])
    args    = words . show $ typeOf x
```

-- AutrijusTang

## 10.6 zipWithN

Here $(zipn 3) = zipWith3 etc.

```
import Language.Haskell.TH; import Control.Applicative; import Control.Monad

zipn n = do
    vs <- replicateM n (newName "vs")
    [| \f ->
        $(lamE (map varP vs)
            [| getZipList $
                $(foldl
                    (\a b -> [| $a <*> $b |])
                    [| pure f |]
                    (map (\v -> [| ZipList $(varE v) |]) vs))
            |])
     |]
```

## 10.7 'generic' zipWith

A generalization of zipWith to almost any data. Demonstrates the ability to do dynamic binding with TH splices (note 'dyn').

```
zipCons :: Name -> Int -> [String] -> ExpQ
zipCons tyName ways functions = do
    let countFields :: Con -> (Name,Int)
        countFields x = case x of
            NormalC n (length -> fields) -> (n, fields)
            RecC n (length -> fields) -> (n,fields)
            InfixC _ n _ -> (n,2)
            ForallC _ _ ct -> countFields ct

    TyConI (DataD _ _ _ [countFields -> (c,n)] _) <- reify tyName
    when (n /= length functions) $ fail "wrong number of functions named"
    vs <- replicateM ways $ replicateM n $ newName "x"
    lamE (map (conP c . map varP) vs) $
        foldl (\con (vs,f) ->
                  con `appE`
                    foldl appE
                        (dyn f)
                        (map varE vs))
              (conE c)
              (transpose vs `zip` functions)
```

This example uses whichever '+' is in scope when the expression is spliced:

```
:type $(zipCons ''(,,,) 2 (replicate 4 "+"))

  $(zipCons ''(,,,) 2 (replicate 4 "+"))
    :: (Num t, Num t1, Num t2, Num t3) =>
        (t, t1, t2, t3) -> (t, t1, t2, t3) -> (t, t1, t2, t3)
```

## 10.8 Instance deriving example

An example using a 'deriving function' to generate a method instance per constructor of a type. The deriving function provides the body of the method.

Note that this example assumes that the functions of the class take a parameter that is the same type as instance is parameterized with.

The message email message (http://www.haskell.org/pipermail/template-haskell/2006-August /000581.html) contains the full source (extracted file (http://www.iist.unu.edu/~vs/haskell /TH_render.hs) ).

## 10.9 QuasiQuoters

New in ghc-6.10 is -XQuasiQuotes, which allows one to extend GHC's syntax from library code. Quite a few examples were previously part of older haskell-src-meta (http://hackage.haskell.org /package/haskell-src-meta-0.2) . Some of these are now part of applicative-quoters (http://hackage.haskell.org/package/applicative-quoters)

### 10.9.1 Similarity with splices

Quasiquoters used in expression contexts (those using the *quoteExp*) behave to a first approximation like regular TH splices:

```
simpleQQ = QuasiQuoter { quoteExp = stringE } -- in another module

[$simpleQQ| a b c d |] == $(quoteExp simpleQQ " a b c d ")
```

## 10.10 Generating records which are variations of existing records

This example uses syb to address some of the pain of dealing with the rather large data types.

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell #-}
module A where
import Language.Haskell.TH
import Data.Generics

addMaybes modName input = let

    rename :: GenericT
    rename = mkT $ \n -> if nameModule n == modName
        then mkName $ nameBase n ++ "_opt"
        else n

    addMaybe :: GenericM Q
    addMaybe = mkM $ \(n :: Name, s :: Strict, ty :: Type) -> do
                    ty' <- [t| Maybe $(return ty) |]
                    return (n,s,ty')

   in everywhere rename `fmap` everywhereM addMaybe input

mkOptional :: Name -> Q Dec
mkOptional n = do
    TyConI d <- reify n
    addMaybes (nameModule n) d
```

mkOptional then generates a new data type with all Names in that module with an added suffix _opt. Here is an example of its use:

```
{-# LANGUAGE TemplateHaskell #-}
import A
data Foo = Foo { a,b,c,d,e :: Double, f :: Int }

mapM mkOptional [''Foo]
```

Generates something like

```
data Foo_opt = Foo_opt {a_opt :: Maybe Double, .....  f_opt :: Maybe Int}
```

Category:

- Language extensions