Notes on Template Haskell Version 2
November 7, 2003

Tim Sheard
OGI School of Science & Engineering
Oregon Health & Science University
sheard@cse.ogi.edu

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

We discuss some design decisions for improving (or fixing) the current Template Haskell implementation. Most of these decisions have to do with the semantics and legal uses of the quasi-quotation mechanism.

1 Introduction

Template Haskell has past its first birthday, and its become clear that some of the initial design, and some of the initial implementation are in need of some adjustment. In this document we explore several issues.

- **Principles.** Section 2 describes two principles that we have found very helpful, but that have not been properly articulated before now.
- **The Need for a** `Name` **ADT** (Section 4). Currently we use `String` as the type for representing variables. In particular, `Var :: String -> Exp`, and `Pvar :: String -> Pat`. This causes many problems with inadvertent scoping, as an unlucky user could use a name which clashed with some internal (GHC) name. For this reason (and others we enumerate later) we introduce an ADT for representing names. This comes with a new lightweight syntax for quoting names (Section 4.3), and a clearer story about dynamic binding (Section 4.4).
- **The Semantics of Reification** (Section 5). Reification is surely the orphan feature of the Template Haskell paper. Mentioned but never really described in any detail. We elaborate on its uses and tie down the semantics of this important feature. We also propose a new interface for reification.
- **Error recovery.** Section 6 describes the new error-reporting interface.
- **Splicing into binding positions** is very tricky, but rather useful. We sketch some ideas in Section 8.
- **Quasi-Quotes as Patterns.** Quasi-quotes are great for constructing code in an abstract manner. But in order to pattern match against code one needs to use the data constructor functions. It's possible to support quasi-quoted code as patterns by using the following convention. When a quasi-quote is used in a pattern, as in: `f [| (x,3,$y,$(Lit x)) |] = ...` We interpret the variables in unspliced contexts as literals, and the spliced contexts as normal patterns. So the pattern example above matches a 4-tuple, whose first component is a variable whose name is "x", whose second component is a literal 3, whose third component is any `Exp`, and which becomes bound to `y`, and whose fourth component must be a literal, and this literal becomes bound to `x`.

This document is not a paper; it is more a collection of design notes. It describes Version 2 of Template Haskell, which we will call "TH2" to distinguish it from the first iteration, TH1. In some places we simply document the design of TH2, while in others we discuss open design choices.

2 Principles

The design of Template Haskell is guided by two over-arching principles:

Static scope is absolute. if an occurrence of a variable `x` looks as if it is lexically bound to an enclosing binding for `x`, then it is so bound, and no Template-Haskell transformation can threaten that binding. For example:

```
x :: Int
x = 3

g :: Int -> ExpQ
g y = [| x + y |]

t :: a -> Int
t x = $(g [| 4 |])
```

The occurrence of `x` in the RHS of `g` is indissoluably bound to the first definition of `x`. The fact that the call to `g` splices the code into a scope where a different `x` is bound makes no difference.

Dollar/bracket cancellation always works, in

- Expressions: `$[|` *e* `|]` = *e*
- Declarations: `$[d|` *d* `|]` = *d*
- Types: `$[t|` *t* `|]` = *t*

This requirement is stronger than it looks, but it is very important, because it gives the user a way to "hand-execute" a program.

Concerning the second point, consider the example given in the original TH paper:

```
data Format = D | S | L String

gen :: [Format] -> ExpQ -> ExpQ
gen []        x = x
gen (D  : xs) x = [| \n-> $(gen xs [| $x++show n |]) |]
```

```
gen (S   : xs) x = [| \s-> $(gen xs [| $x++s |]) |]
gen (L s : xs) x = gen xs [| $x ++ s |]

printf :: String -> ExpQ
printf s = gen (parse s) [| "" |]
```

Now consider a call `$(printf "%d%s")`. We can understand what it means by using ordinary inlining, plus the dollar/bracket rule:

```
      $(printf "%d%s")
⟹    [inline printf, execute parse]
      $(gen [D,S] [| "" |])
⟹    [inline D case of gen]
      $[| \n-> $(gen [S] [| $[| "" |] ++ show n |]) |]
⟹    [apply dollar/bracket rule, twice]
      \n-> $(gen [S] [| "" ++ show n |])
⟹    [property of ++]
      \n-> $(gen [S] [| show n |])
⟹    [inline S case of gen]
      \n-> $([| \s -> $(gen [] [| $[| show n |] ++ s |]))
⟹    [apply dollar/bracket rule, twice]
      \n-> \s -> $(gen [] [| show n ++ s |])
⟹    [inline [] case of gen]
      \n-> \s -> $[| show n ++ s |]
⟹    [apply dollar/bracket rule, once]
      \n-> \s -> show n ++ s
```

Of course, one needs (as usual) to be careful during substitution to maintain correct binding. For example, if we had started with `$(printf "%d%d")`, we would have to take care with this step:

```
      $(printf "%d%d")
⟹    [...]
      \n-> $(gen [D] [| show n |])
⟹    [inline D case of gen, carefully!]
      \n-> $[|\n' -> $(gen [] [| $[| show n |] ++ show n' |]
```

Here, we substituted the actual parameter `[| show n |]` inside the lambda in the body of `gen` so we had to rename the lambda first, to have bound variable `n'`. There is nothing unusal here; it's just the standard rule.

3 Library structure

Much of Template Haskell is in the form of a library. The library structure has changed a bit in TH2, by being split into three modules:

- `Language.Haskell.TH` is the "root" module, and the only one that should be imported by a TH client. It exposes the public interface of the TH library.
- `Language.Haskell.TH.THSyntax` contains the data type declarations for TH. This module should not be directly imported by clients, because it exposes functions that are needed internally (by the TH implementation), but are unsafe if exposed to clients.
- `Language.Haskell.TH.THLib` contains a growing collection of "helper functions" to make life easier for a TH client. Examples include pretty printing and "smart constructors". All of these functions are re-exported via the root module.

4 The `Name` data type

Template Haskell programs occurring inside quasi-quotations represent the computation of ordinary algebraic data structures. These data structures represent Haskell Programs. The particular data types used for Template Haskell include algebraic datatypes to represent expressions (`Exp`), declarations (`Dec`), patterns (`Pat`), and types (`Type`). Additional data types are used to represent other syntactic elements of Haskell.

The naming of these data types, and type synonyms such as `ExpQ`, has changed since TH1. We do not give the details here – look at the source code!

However, there is one big change: the name of a variable is no longer represented by a `String`, as it was in TH1. In TH2 a name is represented by a value of type `Name`, which is abstract to the client.

4.1 The problem we are trying to solve

One of the key features of Template Haskell is that the quasi-quote notation respects both the scoping and typing discipline of Haskell. This is important because it identifies errors in object-level programs (the programs being built) as early as possible. Consider the declaration

```
z = [| \ x -> $(g [| x |]) |]
```

The scoping and typing discipline tries to ensure that the `x` inside the quotes (`[| x |]`) will always refer to the lambda bound `x` in whose scope the quoted expression appears.

Unfortunately, this desire is in direct competition with the desire to manipulate representations of programs as ordinary algebraic data structures. In particular, the original use of the type `String` to represent variable names is too fragile. In Figure 1 we provide three different meta-functions `g` which, if passed `[| x |]` as an argument, might break the connection between the object variable `x` and its lambda binder.

In example **A**, the use of strings to represent both program variables and pattern variables leads to the unfortunate possibility that the user might bind over an expression, that just happens to conflict with one of its free variables. Template Haskell supplies the function `gensym` to produce unique new strings so that programmers can avoid this problem. But, as long as the user can use arbitrary strings as names for variables, it's possible by bad luck, or nefarious action, to choose a string that was also generated by gensym, causing this unwanted capture.

In example **B**, we illustrate another subtle problem. Suppose the user alpha-renamed his program by changing the declaration

```
z = [| \ x -> $(g [| x |]) |]
```

to the declaration

```
z = [| \ y -> $(g [| y |]) |]
```

One would hope that changing the name of a bound variable in a consistent manner would not affect the semantics of the program. Unfortunately, in the presence of meta-functions like the `g` in example **B** this is not the case. Note how `g` returns different results depending upon the *name* of its parameter.

In example **C**, we illustrate a third problem. The ability of a meta-

The three programs below illustrate problems with using strings to represent variables.

example A	example B	example C
`g e = Lam [Pvar "x21"] e`	`g e = do { a <- e ; case e of Var s | s=="x21" -> ... other -> ... }`	`g e = do { a <- e ; case a of Var s -> Lam [Pvar s] a }`
Inadvertent capture of internal names.	Changing the meaning of a program by simple alpha-renaming.	Breaking the connection between a variable and its binding site.

Figure 1. Problems with using strings to represent variables and lexical scoping.

program g to purposefully rebind an object level expression, thus breaking its connection to its original binding site.

We considered several possible solutions to these problems, and settled on one that fixes the problems inherent in examples **A** and **C**, and which enables first class quasi-quotes and (almost) first class splicing. It also facilitates the desire for unscoped quasi-quotes.

4.2 The `Name` data type

We introduce a new abstract data type `Name`, which represents both the binding and use occurrences of variables, type constructors, type variables, classes, and so on:

```
data Pat
  = Pvar Name      -- { x }
  | Ptup [Pat]     -- { (p1,p2) }
  | ...

data Exp
  = Var Name       -- { x }
  | App Exp Exp    -- { f x }
  | Lam [Pat] Exp  -- { \ p1 p2 -> e }
  | ...
```

The `Name` type has the following operations (exported by Language.Haskell.TH):

```
data Name

mkName   :: String -> Name
newName  :: String -> Q Name
nameBase :: Name -> String

instance Show Name
instance Eq   Name
instance Ord  Name
```

The implementation maintains the following invariants.

- The function `mkName` must be passed a string which is a legal Haskell identifier or else it raises an error.
- The function `newName` returns a computation returning a `Name`, and this computation will never return the same `Name` twice, or a `Name` returnable by `mkName`.
- `baseName (mkName s) = s`

In the context of our original example:

```
z = [| \ x -> $(g [| x |]) |]
```

The quasi-quote mechanism uses the `newName` operation to translate the bound object-variable `x` (see Section **??**). Thus, in example **A** the meta-function `g` cannot inadvertently recapture `x`. The implementation of quasi-quotation is free to use `Names` returned by the `newName` mechanism without fear of those clashing with object variables chosen by the programmer.

The problem inherent in Example **C** is not solved by this mechanism, but that's because there is no totally enforcable solution.

Solving the problem inherent in Example **B** is more subtle. In order to solve this problem we must ensure that `Name` is completely abstract — this means no comparisons for equality, and no way to print or show them. This seemed too Draconian for our tastes, as it makes it impossible to analyze `Exp`, for example writing a substitution function becomes impossible. So problem **B** remains.

4.3 New syntax for quoting names

The quotation `[| map |]` has type `Q Exp`. If embedded in a computation thus:

```
do { ...; Var m <- [| map |] ; ... }
```

then `m` would be a `Name` that is indissoluably bound to the `map` from the Prelude (assuming there is no lexically enclosing binding for `map`.

This is fine, but it is clumsy, because (a) `[| map |]` has a monadic type, and (b) it gives an `Exp` rather than a `Name` and one sometimes wants the `Name`. As an example of (b), consider an optimiser that wants to pattern-match against an application of `map`:

```
f :: Exp -> Exp
f (App (Var m) e) | m == <what> = ...
```

In TH1, the only robust solution is clumsy:

```
f :: Exp -> Q Exp
f app = do { Var map_name <- [| map |]
           ; return (f_helper map_name app) }

f_helper :: String -> Exp -> Exp
f_helper map_name (App (Var m) e)
   | m == map_name = ...
```

So TH2 supports a new form of quotation:

```
'map
```

The expression `'`*v* has type `Name` (not `Q Exp`, nor `Q Name`). So now we can say simply:

```
f :: Exp -> Exp
f (App (Var m) e) | m == 'map = ...
```

This new form is still a quotation construct, just like `[| v |]`, and follows the same rules as regards staging. For example, one cannot quote inside quotes, so this is illegal: `[| 'v |]`

Haskell's name-spaces make things just slightly more complicated. The quotation `[| P |]` would mean the *data constructor* `P`, whereas `[t| P |]` would mean the *type constructor* `P`. So we need the same distinction for lightweight quoting. We use two single-quotes to distinguish the type context:

'*v*	means	The name *v* interpreted in an expression context
''*v*	means	The name *v* interpreted in an type context

So `''a` means the type variable `a`, for example.

4.4 Dynamically Scoped `Names`

A *dynamically-scoped* variable is one that does not have to be in scope where it is used, and the system expects that the programmer will close over this free variable before it is spliced into the program. Dynamically scoped variables can be implemented using the `mkName` function to build a `Name`:

```
dyn :: String -> Q Exp
dyn s = return (Var (mkName s))

tp1 :: Q Exp
tpi = [| 2 * pi |]

tp2 :: Q Exp
tp2 = [| 2 * $(dyn "pi") |]
```

Inside `tp1`, the `pi` is indissoluably bound to the Prelude `pi` (presumably in scope here); but inside `tp2` there is no such indissoluable link for `$(dyn "pi")`. Instead the dynamically scoped variable `"pi"` will be captured by the nearest enclosing binding called "pi". For example:

```
h1 pi = $tp1
h2 pi = $tp2
```

In the case of `h1` we have to rename the local binder before inlining `tp1`, to avoid capture:

```
h1 pi' = 2 * pi
```

But in the case of `h2`, the dynamic name is, by design, captured:

```
h2 pi = 2 * pi
```

We are considering providing special syntax: `%x` means `$(dyn "x")`. But Simon thinks it's not worth it.

5 Reification

Reification in TH1 was a language construct, but in TH2 is has become a plain function:

```
reify :: Name -> Q Info
```

The function `reify` takes a `Name`, perhaps got by using the quote notation (e.g. `'map`), or perhaps got from walking over a piece of code. Then `reify` consults the compiler's symbol table and type information, and returns a chunk of information about that thing.

The exact shape of `Info` is likely to change, but here is the current version:

```
data Info
  = ClassI Dec
  | ClassOpI
        Name         -- The class op itself
        Type          -- Type of the class-op
        Name          -- Name of the parent class
        Fixity

  | TyConI Dec
  | DataConI
        Name         -- The data con itself
        Type          -- Type of the constructor
        Name          -- Name of the parent TyCon
        Fixity

  | VarI
        Name         -- The variable itself
        Type
        (Maybe Dec)
        Fixity

  | TyVarI         -- Scoped type variable
        Name
        Type         -- What it is bound to
```

Starting with a data constructor, say, one can get the `Name` of its type constructor, and then use `reify` again to disoover the other data constructors in that type, and so on.

There are several open questions here:

- The type checker may not know the full type of the thing. For example:
  ```
  f x = $( do { i <- reify 'x; ... } )  .....
  ```
  The type of `x` isn't known yet. It may get refined by the other "..." parts of `f`'s body, or by `f`'s use. So should TH give up and refuse to give you `f`'s type? Or what?

  This can apply to top-level decls too:
  ```
  f = 1+2
  $( ... reify 'f... )
  g = f + (4::Int)
  ```
  Because of the monomorphism restriction, the type of `f` is fixed by `f`'s caller, which may be "after" the reification.

  The current implementation simply tells whatever the type checker knows at that moment. If there are as-yet-un-fixed types, the `Type` retuned by `reify` may contain free type variables.

- What `Dec` should be returned for a pattern-bound thing? For example:
  ```
  (f,g) = ...
  ...$(do { d <- reify 'f ; ... }) ...
  ```
  Current solution: return the declaration that binds `f`, even if it binds other things too. If it is part of a mutually recursive group, only the requested binding is returned; if you want the others, you can make further calls to `reify`.

  [Note 7 Nov] This feature still not implemented.

- GHC doesn't have 'deriving' information conveniently to hand for reifiying type decls, so it always returns a `Dec` with an empty `deriving` list. Does it matter?

- GHC doesn't remember whether a DataCon was actually declared infix, so it currently never returns an `InfixC`.

6 Error reporting and recovery

The `Q` monad makes it possible to report errors, and recover from failure gracefully. Here is the current interface (still very much a draft):

`report :: Bool -> String -> Q ()`
: Report something to the user. If the `Bool` is `True`, the something is treated as an error, otherwise it is simply displayed. In both cases, though, execution continues. The difference between the two is seen by `recover`; if there is no enclosing `recover`, compilation fails.

`giveUp :: Q a`
: Stop execution; find the enclosing `recover`.

`recover :: Q a -> Q a -> Q a`
: The call (`recover h q`) runs `q`. If `q` executes `giveUp`, execution resumes with `h`. If `q` runs to completion, but has made some calls to `report True`, the result is discarded and `h` is run. If `q` runs to completion with no error report, `h` is ignored, and `q`'s result is the result of the call to `recover`.

`currentModule :: Q String`
: Returns the name of the module being compiled.

`currentLoc :: Q (FilePath, Int)`
: Returns the location of the top-level splice being executed.

7 Import and export declarations

[Nov 2003: not yet implemented]

We should be able to generate declarations which add to the list of names imported and exported. Consider:

```
module T( f,g, ...)
f x = ...
g y = ....

splice [d| { h x = x + 1; export h } |]
```

The `...` in the module declaration indicates we expect more things to be exported from the module (some of them to be computed by the `spliced` declarations). Spliced `export` lists take scope only in the block(s) following the splice point.

8 Splicing into binding positions

NOTE: this whole section is un-revised, and certainly unimplemented.

We allow splicing in as many positions as possible. It is important to allow a splice of a `Expr`, `Pat`, `Decl`, `Typ`, or a `Var` whenever a syntactic element of that class is expected. Some examples follow:

```
[d| $f x y = x + y + 1 |]          -- f :: Var

[d| f $x y = $(g x [| y |]) |]     -- x :: Pat

[d| f (x,$y) = $(g y [|x|]) |]     -- y :: Pat

[d| f x = $e |]                    -- e :: Q Expr

[d| let $d in x + %y |]            -- d :: Q Decl

[d| data $f = $c Int | $d String |] -- f,c,d :: Var
```

In the original releast o Template Haskell, it was grotesquely difficult to write a function that, say, takes the name of a data type and produces a simple data type with that name. Now we can write:

```
f :: String -> Q Decl
f s = [d| $(mkBndr s) = MkS Int |]
```

However, these new forms introduce a new issue of static scope: what do they bind? When type checking a spliced-in `Pat`, `Decl` or `Var`, we answer the latent scoping question by applying the following rule of thumb: *inside quasi-quotes, a spliced-in binding occurrence binds nothing during the type-checking phase*.

Of course, at the splicing stage, when the quasi- quoted template builds a term, the spliced in object will introduce names that will be binding. If we want to refer to these things with an explicit name, we need to use the dynamic scoping annotation `%f`. Note when the evaluated template is spliced in to become part of the generated program the term is type checked again, and everything is fully known.

```
p = [p| (x,y) |]

[d| f $p  = %x + %y |]
```

Splices, not enclosed inside quasi-quotes follow the same rule of thumb. They expand at compile-time, but are still not known at scope checking time. So the use of dynamic scoping is still necessary.

```
p = [p| (x,y) |]

f $p  = %x + %y
```

8.1 Latent Scoping

One of the purposes of the quasi-quote mechanism is to perform type checking as early as possible to catch errors as early as possible. Catching errors at the compile-time of a template is better than catching them at the splice time of the template. This early type-checking of quasi-quoted terms is complicated by a phenomenon we call latent scoping. Consider the program below.

```
module M
import N(p,derive,gen)

f x = x + 1

g = f 7
 where $(derive "z")
       $p = $(gen 2)
```

Module `M` imports several meta-values from module `N`. The imported values are typed as:

```
p :: Q Pat
derive :: String -> Q Decl
gen :: Int -> Q Exp
```

Latent scoping concerns itself with answering the question: *In the definition of* `g`*, what does* `f` *refer to?* It might refer to the `f` defined at top level in module `M`, or it might refer to names brought into scope by expanding the splices in the `where` clause of `g`.

We decided (for good reasons elaborated below) that if static scoping is to mean static scoping then `f` must refer to the top level `f`, because that is the `f` visible statically (i.e. before any splices in binding positions are expanded). If the user wants `f` to refer to some `f` introduce by splicing `$(derive "z")` or `$(p)`, then he should use dynamically scoped variable `%f`.

```
f x = x + 1

g = %f 7
 where $(derive "z")
       $p = $(gen 2)
```

Such an `f` will be captured by the top-level `f` only if neither `$(derive "z")` or `$(p)` expand to include `f`.

8.2 Top-Level Splices are Special

Top-level decl splices of declarations using the `splice` annotation increase static scope (remember there are no duplicate bindings at top level) in the portion of the module following the splice. They use the top to bottom ordering rule. We call this the "current hack", and it is the reason we separate code into inter-splice groups, each of which gets broken into a number of SCC, each of which is typed together (to maximize polymorphism).

```
splice [d| f x = x + 1 |]

g = f 5
```

Such splices are fully expanded during type checking, and the things they introduce are known, in subsequent parts of the file, and the use of dynamically scoped variables in these subsequent parts of the file are not required.

Top-level splices may introduce binding sites that capture dynamically scoped variables. A dynamically scoped variable can see everything before the current inter splice group, binders in the current decl, but nothing else in this group, or following groups (since the particular partial ordering chosen by the SCC algorithm will effect what is seen).

Inside brackets, Dynamic scope respects the left-right, top-bottom rule

```
  [d| $d; g =4 ; f = ...(dyn "g") |]
```

8.3 Pattern quasi-quotes

Quasi-quotes for patterns introduce new binders, but because the scope of the binders is not apparent they have no scope. The names in the patterns are literally lifted from the name inside the quasi-quote.

```
[p| (x,y) |]  ===
    return(Ptuple [Pvar (mkVar "x"),Pvar (mkVar "y')])
```

Splicing holes inside pattern level quasi-quotes are in the scope were the pattern quasi-quote appears. Not in the new scope produced by the pattern. Thus inside

```
let x = 3
in [p| (x,$(f x)) |]

===

let x = 3
in return(Ptuple[Pvar "x",f x])
```

Note that the two x's refer to two different things. You can use this pattern to close a scope, by using the dynamic binding mechanism. Thus

```
let p1 = [p| (x,y) |]
[| \ $p1 -> %x - y |]
```

the pattern closes over `%x` but not over `y`.

8.4 Pattern Quasi-Quotes with Explicit Vars

If the user wants to indicate which variables a computed pattern will export, the *fresh* pattern notation may be used.

```
z = True
f x $[x,y](g x y z 2) = x - y * %w
```

This tells the type checker that the expression `(g x y z 2)` will evaluate to a pattern that binds `x` and `y` (this is tested at run time, and if they are not bound it is an error). The vars `x` and `y` are in scope in both `(g x y z 2)` and in `(x - y * %w)`, but with different interpretations. Inside `(g x y z 2)` they are bound to new gensym'd `Vars`, and inside `(x - y * %w)` they are meta variables bound by the pattern computed by `(g x y z 2)`, and their type is determined by the type checker. This applies to any spliced object-level term in a binding position (namely patterns and declarations).

9 Quasi-Quotes as Meta-Level Patterns

NOTE: this whole section is un-revised, and certainly un-implemented.

A quasi-quoted expression (pattern, decl, typ) in a meta-level pattern position stands for a pattern.

```
f [| e1 e2 |] = ...
  ===
f (App (Var x) (Var y)) | x== 'e1 && y=='e2 = ...

f [| $x + $y |] = ...
  ===
f (App x y) = ...


f [| \ $p -> $e |] = ...
  ===
f (Lam [p] e) = ...
```

In fact under a splice in such a LHS quasi-quote (used as a meta-level pattern) you can put any meta-level pattern, not just variables.

```
f [| \ ($(Pvar s),$(Pvar t)) -> $e |] = ...
f (Lam [Ptuple[Pvar s,Pvar t]] e) = ...
```

How do you interpret bound variables in quasi-quote meta-level patterns?

```
f [| \ x -> (x,x) |] = ...
   ===
f (Lam [Pvar v] (Ptuple [Var o1,Var o2]))
    | bndVar o1 == v && bndVar o2 == v = ...
```

Note that if the bound var does not occur in its scope it matches any binding. Also note that the bound var does get bound at the meta-level. So the right-hand-side of the equation can't use the bound variable in any way. The invented pattern variables `v`, `o1`, `o2`, etc. are unknown to the programmer and cannot be accessed.

```
f [| \ x -> (x,x,$w) |] =  [| \ x -> (x,x+1,$w) |]
```

If `x` appears in `w`, it then becomes unbound in the right-hand-side that is built, because the new `x` is alpha renamed, and the old `Var` is lost. If you want some other effect, you must use the constructor functions like `Lam` and `PVar` to construct the pattern.

```
f [| \ $(Pvar x) -> ($(Var y),$w) |] | bv x ==y =
    [| \ $(Pvar x) -> ($(Var y)+1,$w) |]
```

10 Functoring the Q monad

(This is really an implementation matter, but it's neat.) One of the tricky things about TH, from an implementation point of view, is that much of it is a library. The library uses `reify`, `Q` etc, and hence requires a type definition for `Q`. But we really want `Q` to be the typechecker monad itself... but we don't want to import GHC into the TH library.

So here's a neat trick. Define a class, `Quasi`, which provides the fundamental operations necessary to implement `Q`:

```
class Monad m => Quasi m where
  qReify :: Name -> m Info
  qNewName :: m Name
  qRecover :: m a -> m a -> m a
  ...etc..
```

Now define `Q` using `Quasi`, thus:

```
newtype Q a = Q (forall m . Quasi m => m a)
```

That is, a `Q` value is a function that takes a record of operations (`qNewName`, `qReify` etc) and returns a computation `m a`. Now we can implement the `Q` versions of `reify` etc easily:

```
reify :: Name -> Q Info
reify n = Q (qReify n)

recover :: Q a -> Q a -> Q a
recover (Q a) (Q b) = Q (qRecover a b)
```

The beautiful thing is that this makes no mention of GHC's type checker monad `TcM`. But when we import this module into a stage-2 build of GHC, we can make `TcM` an instance of `Quasi`:

```
instance Quasi TcM where
  qReify n = ...consult typechecker monad,
                generate Info...

  qReport b s = ...add error into typechecker monad...
  ...etc...
```

So the computation really does take place directly in the type-checker moand, with no impedence-matching.

Sometimes one may want to run a `Q` computation at run-time:

```
f :: ExpQ
f = [| \x -> x |]

main = do { e <- runQ f; print e }
```

So how does `runQ :: Q a -> IO a` work? Well, we just need to make `IO` an instance of `Quasi` and away we go. Of course, the `IO` monad does not have access to the type environment, so calls to `reify` in the computations are going to fail, but that is fair enough.

What all this amounts to is a kind run-time functorisation of the Q monad. A `Q` computation takes a record (or module) of monadic operations, and uses those.

11 Open issues

- Lifting rule. `f x = [| x |]` is ok, if x has a liftable type, but `f x = 'x` is not.

12 Other things asked for

Wolfgang Thaller, Alastair Reid. State!

Alastair Reid. A related issue is that I generate files corresponding to each TH module I compile. If ghc is producing a file called `foo/bar.o`, I'd like the name of the file to be `foo/bar_stub.o`. For this sort of application, I need access to the name of the input and output filenames.

Alastair Reid. Can you drop the 'tick' from variable names produced using gensym? If you'd used underscore, the variable name would be a legal C identifier too. (Ok, so this is a bit specialized but it's very easy to do.)

Wolfgang Thaller. reifyImportedModules :: Q [String]

Sean Seefried. I think the ability to reify entire modules or reify functions from any module (including standard libraries) would be very useful to any person interested in language extension.

However, I realise that there are philosophical and implementation oriented problems with the idea. Firstly, being able to reify code from another module means that all .hi files would bloat to many times their size as they now have to contain reified versions of the declarations in the modules and all this just in case someone happens to want to reify the declaration somewhere else.

Also, I realise that this partially breaks the abstraction mechanism of modules. One reason that we have modules and interfaces is so that we don't have to (or even *can't*) look inside the implementation of a module.

More Descriptive Literals. At the moment literals are quite general, and often get implicitly wrapped by overloaded injection functions, such as `fromInteger` by the type-checking process. If we could be more precise about literals, then this needn't be the case. We might consider a two tier system like.

```
data Exp = Lit Lit | ...
data Lit = Tlit Descr Typ | Lit Descr
data Descr = I Integer | R Rational | S String
```