Template Meta-programming for Haskell

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

Simon Peyton Jones
Microsoft Research Ltd
simonpj@microsoft.com

Abstract

We propose a new extension to the purely functional programming language Haskell that supports *compile-time meta-programming*. The purpose of the system is to support the *algorithmic* construction of programs at compile-time.

The ability to generate code at compile time allows the programmer to implement such features as polytypic programs, macro-like expansion, user directed optimization (such as inlining), and the generation of supporting data structures and functions from existing data structures and functions.

Our design is being implemented in the Glasgow Haskell Compiler, `ghc`.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages

General Terms

Languages, Design

Keywords

Meta programming, templates

1 Introduction

> "Compile-time program optimizations are similar to poetry: more are written than are actually published in commercial compilers. Hard economic reality is that many interesting optimizations have too narrow an audience to justify their cost... An alternative is to allow programmers to define their own compile-time optimizations. This has already happened accidentally for C++, albeit imperfectly... [It is] obvious to functional programmers what the committee did not realize until later: [C++] templates are a functional language evaluated at compile time..." [12].

Haskell Workshop, Oct 2002, Pittsburgh

Robinson's provocative paper identifies C++ templates as a major, albeit accidental, success of the C++ language design. Despite the extremely baroque nature of template meta-programming, templates are used in fascinating ways that extend beyond the wildest dreams of the language designers [1]. Perhaps surprisingly, in view of the fact that templates are functional programs, functional programmers have been slow to capitalize on C++'s success; while there has been a recent flurry of work on *run-time* meta-programming, much less has been done on *compile-time* meta-programming. The Scheme community is a notable exception, as we discuss in Section 10.

In this paper, therefore, we present the design of a compile-time meta-programming extension of Haskell, a strongly-typed, purely-functional language. The purpose of the extension is to allow programmers to *compute* some parts of their program rather than *write* them, and to do so seamlessly and conveniently. The extension can be viewed both as a template system for Haskell (*à la* C++), as well as a type-safe macro system. We make the following new contributions:

- We describe how a quasi-quotation mechanism for a language with binders can be precisely described by a translation into a monadic computation. This allows the use of a gensym-like operator even in a purely functional language like Haskell (Sections 6.1 and 9).
- A staged type-checking algorithm co-routines between type checking and compile-time computations. This staging is useful, because it supports code generators, which if written as ordinary programs, would need to be given dependent types. The language is therefore expressive and simple (no dependent types), but still secure, because all run-time computations (either hand-written or computed) are always type-checked before they are executed (Section 7).
- Reification of programmer-written components is supported, so that computed parts of the program can analyze the structure of user-written parts. This is particularly useful for building "boilerplate" code derived from data type declarations (Sections 5 and 8.1).

In addition to these original contributions, we have synthesized previous work into a coherent system that provides new capabilities. These include

- The representation of code by an ordinary algebraic datatype makes it possible use Haskell's existing mechanisms (`case` analysis) to observe the structure of code, thereby allowing the programmer to write code *manipulation* programs, as well as code *generation* programs (Sections 6.2 and 9.3).

- This is augmented by a quotation monad, that encapsulates meta-programming features such as fresh name generation, program reification, and error reporting. A monadic library of *syntax operators* is built on top of the algebraic datatypes and the quotation monad. It provides an easy-to-use interface to the meta-programming parts of the system (Sections 4, 6, 6.3, and Section 8).
- A quasi-quote mechanism is built on top of the monadic library. Template Haskell extends the meta-level operations of static scoping and static type-checking into the object-level code fragments built using its quasi-quote mechanism (Sections 9 and 7.1). Static scoping and type-checking do not automatically extend to code fragments built using the algebraic datatype representation; they would have to be "programmed" by the user (Sections 9 and 9.3).
- The reification facilities of the quotation monad allows the programmer (at compile-time) to query the compiler's internal data structures, asking questions such as "What is the line number in the source-file of the current position?" (useful for error reporting), or "What is the kind of this type constructor?" (Section 8.2).
- A meta-program can produce *a group of declarations*, including data type, class, or instance declarations, as well as an *expression* (Section 5.1).

2 The basic idea

We begin with an example to illustrate what we mean by meta-programming. Consider writing a C-like `printf` function in Haskell. We would like to write something like:

```
printf "Error: %s on line %d." msg line
```

One cannot define `printf` in Haskell, because `printf`'s type depends, in a complicated way, on the value of its first argument (but see [5] for an ingenious alternative). In Template Haskell, though, we can define `printf` so that it is *type-safe* (i.e. report an error at compile-time if `msg` and `line` do not have type `String` and `Int` respectively), *efficient* (the control string is interpreted at compile time), and *user-definable* (no fixed number of compiler extensions will ever be enough). Here is how we write the call in Template Haskell:

```
$(printf "Error: %s on line %d") msg line
```

The "`$`" says "evaluate at compile time"; the call to `printf` returns a Haskell expression that is spliced in place of the call, after which compilation of the original expression can proceed. We will often use the term "splice" for `$`[1]. The splice `$(printf ...)` returns the following code:

```
(\ s0 -> \ n1 ->
    "Error: " ++ s0 ++ " on line " ++ show n1)
```

This lambda abstraction is then type-checked and applied to `msg` and `line`. Here is an example interactive session to illustrate:

```
prompt> $(printf "Error: %s at line %d") "Bad var" 123
  :: [Char]
  "Error: Bad var at line 123"
```

[1]Note that in Template Haskell that `$` followed by an open parenthesis or an alphabetic character is a special syntactic form. So `x $y` means "`x` applied to splice `y`", whereas `x $ y` means the ordinary infix application of the function `$` just as it does in ordinary Haskell. The situation is very similar to that of "`.`", where `A.b` means something different from `A . b`.

The function `printf`, which is executed at compile time, is a program that produces a program as its result: it is a *meta-program*. In Template Haskell the user can define `printf` thus:

```
printf :: String -> Expr
printf s = gen (parse s)
```

The type of `printf` says that it transforms a format string into a Haskell expression, of type `Expr`. The auxiliary function `parse` breaks up the format string into a more tractable list of format specifiers:

```
data Format = D | S | L String
parse :: String -> [Format]
```

For example,

`parse "%d is %s"`   *returns*   `[D, L " is ", S]`

Even though `parse` is executed at compile time, it is a perfectly ordinary Haskell function; we leave its definition as an exercise. The function `gen` is much more interesting. We first give the code for `gen` assuming exactly one format specifier:

```
gen :: [Format] -> Expr
gen [D]   = [| \n -> show n |]
gen [S]   = [| \s -> s |]
gen [L s] = lift s
```

The results of `gen` are constructed using the *quasi-quote* notation — the "templates" of Template Haskell. Quasi-quotations are the user's interface to representing Haskell programs, and are constructed by placing quasi-quote brackets `[| _ |]` around ordinary Haskell concrete syntax fragments. The function `lift :: String -> Expr` "lifts" a string into the `Expr` type, producing an `Expr` which, if executed, would evaluate to `lifts`'s argument. We have more to say about `lift` in Section 9.1

Matters become more interesting when we want to make `gen` recursive, so that it can deal with an arbitrary list of format specifiers. To do so, we have to give it an auxiliary parameter, namely an expression representing the string to prefix to the result, and adjust the call in `printf` accordingly:

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

gen :: [Format] -> Expr -> Expr
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 ++ $(lift s) |]
```

Inside quotations, the splice annotation (`$`) still means "evaluate when the quasi-quoted code is constructed"; that is, when `gen` is called. The recursive calls to `gen` are therefore run at compile time, and the result is spliced into the enclosing quasi-quoted expression. The argument of `$` should, as before, be of type `Expr`.

The second argument to the recursive call to `gen` (its accumulating parameter) is of type `Expr`, and hence is another quasi-quoted expression. Notice the arguments to the recursive calls to `gen` refer to object-variables (`n`, and `s`), bound in outer quasi-quotes. These occurrences are within the static scope of their binding occurrence: static scoping extends across the template mechanism.

3 Why templates?

We write programs in high-level languages because they make our programs shorter and more concise, easier to maintain, and easier to think about. Many low level details (such as data layout and memory allocation) are abstracted over by the compiler, and the programmer no longer concerns himself with these details. Most of the time this is good, since expert knowledge has been embedded into the compiler, and the compiler does the job in manner superior to what most users could manage. But sometimes the programmer knows more about some particular details than the compiler does. It's not that the compiler couldn't deal with these details, but that for economic reasons it just doesn't [12]. There is a limit to the number of features any compiler writer can put into any one compiler. The solution is to construct the compiler in a manner in which ordinary users can teach it new tricks.

This is the rationale behind Template Haskell: to make it easy for programmers to teach the compiler a certain class of tricks. What do compilers do? They manipulate programs! Making it easy for users to manipulate their own programs, and also easy to interlace their manipulations with the compiler's manipulations, creates a powerful new tool.

We envision that Template Haskell will be used by programmers to do many things.

- *Conditional compilation* is extremely useful for compiling a single program for different platforms, or with different debugging options, or with a different configuration. A crude approach is to use a preprocessor like `cpp` — indeed several compilers for Haskell support this directly — but a mechanism that is part of the programming language would work much better.
- *Program reification* enables programs to inspect their own structure. For example, generate a function to serialise a data structure, based on the data type declaration for that structure.
- *Algorithmic program construction* allows the programmer to construct programs where the algorithm that describes how to construct the program is simpler than the program itself. Generic functions like `map` or `show` are prime examples, as are compile-time specialized programs like `printf`, where the code compiled is specialized to compile-time constants.
- *Abstractions that transcend the abstraction mechanisms accessible in the language*. Examples include: introducing higher-order operators in a first-order language using compile-time macros; or implementing integer indexed functions (like `zip1`, `zip2`, ... `zip`*n*) in a strongly typed language.
- *Optimizations* may teach the compiler about domain-specific optimizations, such as algebraic laws, and in-lining opportunities.

In Template Haskell, functions that execute at compile time are written in the same language as functions that execute at run time, namely Haskell. This choice is in sharp contrast with many existing systems; for example, `cpp` has its own language (`#if`, `#define` etc.), and template meta-programs in C++ are written entirely in the type system. A big advantage of our approach is that existing libraries and programming skills can be used directly; arguably, a disadvantage is that explicit annotations ("`$`" and "`[| |]`") are necessary to specify which bits of code should execute when. Another consequence is that the programmer may erroneously write a non-terminating function that executes at compile time. In that case, the compiler will fail to terminate; we regard that as a programming error that is no more avoidable than divergence at run time.

In the rest of the paper we flesh out the details of our design. As we shall see in the following sections, it turns out that the simple quasi-quote and splice notation we have introduced so far is not enough.

4 More flexible construction

Once one starts to use Template Haskell, it is not long before one discovers that quasi-quote and splice cannot express anything like the full range of meta-programming opportunities that we want.

Haskell has built-in functions for selecting the components from a pair, namely `fst` and `snd`. But if we want to select the first component of a triple, we have to write it by hand:

```
case x of (a,b,c) -> a
```

In Template Haskell we can instead write:

```
$(sel 1 3) x
```

Or at least we would like to. But how can we write `sel`?

```
sel :: Int -> Int -> Expr
sel i n = [| \x -> case x of ... |]
```

Uh oh! We can't write the "`...`" in ordinary Haskell, because the pattern for the case expression depends on `n`. The quasi-quote notation has broken down; instead, we need some way to construct Haskell syntax trees more directly, like this:

```
sel :: Int -> Int -> Expr
sel i n = lam [pvar "x"] (caseE (var "x") [alt])
  where alt :: Match
        alt = simpleM pat rhs

        pat :: Patt
        pat = ptup (map pvar as)

        rhs :: Expr
        rhs = var (as !! (i-1)) -- !! is 0 based

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

In this code we use *syntax-construction functions* which construct expressions and patterns. We list a few of these, their types, and some concrete examples for reference.

```
-- Syntax for Patterns
pvar    :: String -> Patt           -- x
ptup    :: [Patt] -> Patt           -- (x,y,z)
pcon    :: String -> [Patt] -> Patt -- (Fork x y)
pwild   :: Patt                     -- _

-- Syntax for Expressions
var     :: String -> Expr           -- x
tup     :: [Expr] -> Expr           -- (x,3+y)
app     :: Expr -> Expr -> Expr     -- f x
lam     :: [Patt] -> Expr -> Expr   -- \ x y -> 5
caseE   :: Expr -> [Match] -> Expr  -- case x of ...
simpleM :: Patt -> Expr -> Match    -- x:xs -> 2
```

The code for `sel` is more verbose than that for `printf` because it uses explicit constructors for expressions rather than implicit ones. In exchange, code construction is fundamentally more flexible, as `sel` shows. Template Haskell provides a full family of syntax-construction functions, such as `lam` and `pvar` above, which are documented in Appendix A.

The two styles can be mixed freely. For example, we could also write `sel` like this:

```
sel :: Int -> Int -> Expr
sel i n = [| \ x -> $(caseE [| x |] [alt]) |]
        where
          alt = simpleM pat rhs
```

```
            pat = ptup (map pvar as)
            rhs = var (as !! (i-1))
            as  = ["a"++show i | i <- [1..n] ]
```

To illustrate the idea further, suppose we want an n-ary `zip` function, whose call might look like this:

```
$(zipN 3) as bs cs
```

where `as`, `bs`, and `cs` are lists, and `zipN :: Int -> Expr` generates the code for an n-ary `zip`. Let's start to write `zipN`:

```
zipN :: Int -> Expr
zipN n = [| let zp = $(mkZip n [| zp |]) in zp |]

mkZip :: Int -> Expr -> Expr
```

The meta-function `zipN` generates a local let binding like (`let zip3 = ... in zip3`). The body of the binding (the dots ...) is generated by the auxiliary meta-function `mkZip` defined below. The function defined in the let (`zip3` in the example in this paragraph) will be recursive. The name of this function doesn't really matter, since it is used only once in the result of the let, and never escapes the scope of the let. It is the whole let expression that is returned. The name of this function must be passed to `mkZip` so that when `mkZip` generates the body, the let will be properly scoped. The size of the zipping function, `n`, is also a parameter to `mkZip`.

It's useful to see what `mkZip` generates for a particular `n` in understanding how it works. When applied to `3`, and the object variable (`var "ff"`) it generates a value in the `Expr` type. Pretty-printing that value as concrete syntax we get:

```
\ y1 y2 y3 ->
    case (y1,y2,y3) of
      (x1:xs1,x2:xs2,x3:xs3) ->
           (x1,x2,x3) : ff xs1 xs2 xs3
      (_,_,_) -> []
```

Note how the parameter (`var "ff"`) ends up as a function in one of the arms of the case. When the user level function `zipN` (as opposed to the auxiliary function `mkZip`) is applied to `3` we obtain the full let. Note how the name of the bound variable `zp0`, which is passed as a parameter to `mkZip` ends up in a recursive call.

```
let zp0 =
     \ y1 y2 y3 ->
        case (y1,y2,y3) of
         ((x1:xs1),(x2:xs2),(x3:xs3)) ->
               (x1,x2,x3) : zp0 xs1 xs2 xs3
         (_,_,_) -> []
in zp0
```

The function `mkZip` operates by generating a bunch of patterns (e.g. `y1`, `y2`, `y3` and `(x1:xs1,x2:xs2,x3:xs3)`), and a bunch of expressions using the variables bound by those patterns. Generating several patterns (each a pattern-variable), and associated expressions (each an expression-variable) is so common we abstract it into a function

```
genPE :: :: String -> Int -> ([Patt],[Expr])
genPE s n = let ns = [ s++(show i) | i <- [1..n]]
            in (map pvar ns, map var ns)

-- genPe "x" 2 -->
--  ([pvar "x1",pvar "x2"],[var "x1",var "x2"])
```

In `mkZip` we use this function to construct three lists of matching patterns and expressions. Then we assemble these pieces into the lambda abstraction whose body is a case analysis over the lambda abstracted variables.

```
mkZip :: Int -> Expr -> Expr
mkZip n name = lam pYs (caseE (tup eYs) [m1,m2])
  where
    (pXs, eXs)  = genPE "x"  n
    (pYs, eYs)  = genPE "y"  n
    (pXSs,eXSs) = genPE "xs" n
    pcons x xs  = [p| $x : $xs |]
    b  = [| $(tup eXs) : $(apps(name : eXSs)) |]
    m1 = simpleM (ptup (zipWith pcons pXs pXSs)) b
    m2 = simpleM (ptup (copies n pwild)) (con "[]")
```

Here we use the quasi-quotation mechanism for patterns `[p| _ |]` and the function `apps`, another idiom worth abstracting into a function — the application of a function to multiple arguments.

```
apps :: [Expr] -> Expr
apps [x]      = x
apps (x:y:zs) = apps ( [| $x $y |] : zs )
```

The message of this section is this. Where it works, the quasi-quote notation is simple, convenient, and secure (it understands Haskell's static scoping and type rules). However, quasi-quote alone is not enough, usually when we want to generate code with sequences of indeterminate length. Template Haskell's syntax-construction functions (`app`, `lam`, `caseE`, etc.) allow the programmer to drop down to a less convenient but more expressive notation where (and only where) necessary.

5 Declarations and reification

In Haskell one may add a "`deriving`" clause to a `data` type declaration:

```
data T a = Tip a | Fork (T a) (T a)  deriving( Eq )
```

The `deriving( Eq )` clause instructs the compiler to generate "boilerplate" code to allow values of type `T` to be compared for equality. However, this mechanism only works for a handful of built-in type classes (`Eq`, `Ord`, `Ix` and so on); if you want instances for other classes, you have to write them by hand. So tiresome is this that Winstanley wrote DrIFT, a pre-processor for Haskell that allows the programmer to specify the code-generation algorithm once, and then use the algorithm to generate boilerplate code for many data types [17]. Much work has also been done on poly-typic algorithms, whose execution is specified, once and for all, based on the structure of the type [9, 6].

Template Haskell works like a fully-integrated version of DrIFT. Here is an example:

```
data T a = Tip a | Fork (T a) (T a)

splice (genEq (reifyDecl T))
```

This code shows two new features we have not seen before: reification and declaration splicing. Reification involves making the internal representation of `T` available as a data structure to compile-time computations. Reification is covered in more detail in Section 8.1.

5.1 Declaration splicing

The construct `splice (...)` may appear where a *declaration group* is needed, whereas up to now we have only seen `$(...)` where an *expression* is expected. As with `$`, a `splice` instructs the compiler to run the enclosed code at compile-time, and splice in the resulting declaration group in place of the `splice` call[2].

Splicing can generate one or more declarations. In our example, `genEq` generated a single `instance` declaration (which is essential

[2]An aside about syntax: we use "`splice`" rather than "`$`" only because the latter seems rather terse for a declaration context.

for the particular application to `deriving`), but in general it could also generate one or more `class`, `data`, `type`, or value declarations.

Generating declarations, rather than expressions, is useful for purposes other than deriving code from data types. Consider again the n-ary `zip` function we discussed in Section 4. Every time we write `$(zipN 3) as bs cs` a fresh copy of a 3-way zip will be generated. That may be precisely what the programmer wants to say, but he may also want to generate a single top-level zip function, which he can do like this:

```
zip3 = $(zipN 3)
```

But he might want to generate all the zip functions up to 10, or 20, or whatever. For that we can write

```
splice (genZips 20)
```

with the understanding that `zip1`, `zip2`, ... , `zip20` are brought into scope.

6 Quasi-quotes, Scoping, and the Quotation Monad

Ordinary Haskell is statically scoped, and so is Template Haskell. For example consider the meta-function `cross2a` below.

```
cross2a :: Expr -> Expr -> Expr
cross2a f g = [| \ (x,y) -> ($f x, $g y) |]
```

Executing `cross2a (var "x") (var "y")` we expect that the (`var "x"`) and the (`var "y"`) would not be inadvertently captured by the local object-variables `x` and `y` inside the quasi-quotes in `cross2a`'s definition. Indeed, this is the case.

```
prompt> cross2a (var "x") (var "y")
Displaying top-level term of type: Expr
\ (x0,y1) -> (x x0,y y1)
```

The quasi-quote notation renames `x` and `y`, and we get the expected result. This is how static scoping works in ordinary Haskell, and the quasi-quotes lift this behavior to the object-level as well. Unfortunately, the syntax construction functions `lam`, `var`, `tup`, etc. do not behave this way. Consider

```
cross2b f g
  = lam [ptup [pvar "x",       pvar "y"]]
        (tup  [app f (var "x"),app g (var "y")])
```

Applying `cross2b` to `x` and `y` results in inadvertent capture.

```
prompt> cross2b (var "x") (var "y")
Displaying top-level term of type: Expr
\ (x,y) -> (x x,y y)
```

Since some program generators cannot be written using the quasi-quote notation alone, and it appears that the syntax construction functions are inadequate for expressing static scoping, it appears that we are in trouble: we need some way to generate fresh names. That is what we turn to next.

6.1 Secrets Revealed

Here, then, is one correct rendering of `cross` in Template Haskell, without using quasi-quote:

```
cross2c :: Expr -> Expr -> Expr
cross2c f g =
  do { x <- gensym "x"
     ; y <- gensym "y"
     ; ft <- f
     ; gt <- g
     ; return (Lam [Ptup [Pvar x,Pvar y]]
                   (Tup  [App ft (Var x)
                         ,App gt (Var y)]))
     }
```

In this example we reveal three secrets:

- The type `Expr` is a synonym for monadic type, `Q Exp`. Indeed, the same is true of declarations:

  ```
  type Expr = Q Exp
  type Decl = Q Dec
  ```

- The code returned by `cross2c` is represented by ordinary Haskell algebraic datatypes. In fact there are two algebraic data types in this example: `Exp` (expressions) with constructors `Lam`, `Tup`, `App`, etc; and `Pat` (patterns), with constructors `Pvar`, `Ptup`, etc.
- The monad, `Q`, is the quotation monad. It supports the usual monadic operations (`bind`, `return`, `fail`) and the `do`-notation, as well as the `gensym` operation:

  ```
  gensym :: String -> Q String
  ```

We generate the `Expr` returned by `cross2c` using Haskell's monadic `do`-notation. First we generate a fresh name for `x` and `y` using a monadic `gensym`, and then build the expression to return. Notice that (tiresomely) we also have to "perform" `f` and `g` in the monad, giving `ft` and `gt` of type `Exp`, because `f` and `g` have type `Q Exp` and might do some internal `gensym`s. We will see how to avoid this pain in Section 6.3.

To summarize, in Template Haskell there are three "layers" to the representation of object-programs, in order of increasing convenience and decreasing power:

- The bottom layer has two parts. First, *ordinary algebraic data types* represent Haskell program fragments (Section 6.2).

  Second, *the quotation monad*, `Q`, encapsulates the notion of generating fresh names, as well as failure and input/output (Section 8).
- A library of *syntax-construction functions*, such as `tup` and `app`, lift the corresponding algebraic data type constructors, such as `Tup` and `App`, to the quotation-monad level, providing a convenient way to access the bottom layer (Section 6.3).
- *The quasi-quote notation*, introduced in Section 2, is most convenient but, as we have seen, there are important meta-programs that it cannot express. We will revisit the quasi-quote notation in Section 9, where we show how it is built on top of the previous layers.

The programmer can freely mix the three layers, because the latter two are simply convenient interfaces to the first. We now discuss in more detail the first two layers of code representation. We leave a detailed discussion of quasi-quotes to Section 9.

6.2 Datatypes for code

Since object-programs are data, and Haskell represents data structures using algebraic datatypes, it is natural for Template Haskell to represent Haskell object-programs using an algebraic datatype.

The particular data types used for Template Haskell are given in Appendix B. The highlights include algebraic datatypes to represent expressions (`Exp`), declarations (`Dec`), patterns (`Pat`), and types (`Typ`). Additional data types are used to represent other syntactic elements of Haskell, such as guarded definitions (`Body`), `do` expressions and comprehensions (`Statement`), and arithmetic sequences (`DotDot`). We have used comments freely in Appendix B to illustrate the algebraic datatypes with concrete syntax examples.

We have tried to make these data types complete yet simple. They are modelled after Haskell's concrete surface syntax, so if you can write Haskell programs, you should be able to use the algebraic constructor functions to represent them.

An advantage of the algebraic approach is that object-program representations are just ordinary data; in particular, they can be analysed using Haskell's `case` expression and pattern matching.

Disadvantages of this approach are verbosity (to construct the representation of a program requires considerably more effort than that required to construct the program itself), and little or no support for semantic features of the object language such as scoping and typing.

6.3 *The syntax-construction functions*

The syntax-construction functions of Section 4 stand revealed as the monadic variants of the corresponding data type constructor. For example, here are the types of the `data` type constructor `App`, and its monadic counterpart (remember that `Expr = Q Exp`):

```
App :: Exp  -> Exp  -> Exp
app :: Expr -> Expr -> Expr
```

The arguments of `app` are *computations*, whereas the arguments of `App` are *data values*. However, `app` is no more than a convenience function, which simply performs the argument computations before building the result:

```
app :: Expr -> Expr -> Expr
app x y = do { a <- x; b <- y; return (App a b)}
```

This convenience is very worth while. For example, here is yet another version of `cross`:

```
cross2d :: Expr -> Expr -> Expr
cross2d f g
  = do { x <- gensym "x"
       ; y <- gensym "y"
       ; lam [ptup [pvar x, pvar y]]
             (tup [app f (var x)
                  ,app g (var y)])
       }
```

We use the monadic versions of the constructors to build the result, and thereby avoid having to bind `ft` and `gt` "by hand" as we did in `cross2c`. Instead, `lam`, `app`, and `tup`, will do that for us.

In general, we use the following nomenclature:

- A four-character type name (e.g. `Expr`) is the monadic version of its three-character algebraic data type (e.g. `Exp`).
- A lower-cased function (e.g. `app`) is the monadic version of its upper-cased data constructor (e.g. `App`)[3].

While `Expr` and `Decl` are monadic (computational) versions of the underlying concrete type, the corresponding types for patterns (`Patt`) and types (`Type`) are simply synonyms for the underlying data type:

```
type Patt = Pat
type Type = Typ
```

[3]For constructors whose lower-case name would clash with Haskell keywords, like `Let`, `Case`, `Do`, `Data`, `Class`, and `Instance` we use the convention of suffixing those lower-case names with the initial letter of their type: `letE`, `caseE`, `doE`, `dataD`, `classD`, and `instanceD`.

Reason: we do not need to `gensym` when constructing patterns or types. Look again at `cross2d` above. There would be no point in `gensym`'ing `x` or `y` inside the pattern, because these variables must scope over the body of the lambda as well.

Nevertheless, we provide type synonyms `Patt` and `Type`, together with their lower-case constructors (`pvar`, `ptup` etc.) so that programmers can use a consistent set — lower-case when working in the computational setting (even though only the formation of `Exp` and `Dec` are computational), and upper-case when working in the algebraic datatype setting.

The syntax-construction functions are no more than an ordinary Haskell library, and one that is readily extended by the programmer. We have seen one example of that, in the definition of `apps` at the end of Section 4, but many others are possible. For example, consider this very common pattern: we wish to generate some code that will be in the scope of some newly-generated pattern; we don't care what the names of the variables in the pattern are, only that they don't clash with existing names. One approach is to `gensym` some new variables, and then construct both the pattern and the expression by hand, as we did in `cross2d`. But an alternative is to "clone" the whole pattern in one fell swoop, rather than generate each new variable one at a time:

```
cross2e f g =
  do { (vf,p) <- genpat (ptup [pvar "x",pvar "y"])
     ; lam [p] (tup[app f (vf "x"),app g (vf "y")])
     }
```

The function `genpat :: Patt -> Q (String->Expr, Patt)` alpha-renames a whole pattern. It returns a new pattern, and a function which maps the names of the variables in the original pattern to `Expr`s with the names of the variables in the alpha-renamed pattern. It is easy to write by recursion over the pattern. Such a scheme can even be mixed with the quasi-quote notation.

```
cross2e f g =
  do { (vf,p) <- genpat [p| (x,y) |]
     ; lam [p] [| ( $f $(vf "x"), $g $(vf "y") ) |]
     }
```

This usees the quasi-quote notation for patterns: `[p| _ |]` that we mentioned in passing in Section 4. We also supply a quasi-quote notation for declarations `[d| _ |]` and types `[t| _ |]`. Of course all this renaming happens automatically with the quasi-quotation. We explain that in detail in Section 9.

7 Typing Template Haskell

Template Haskell is strongly typed in the Milner sense: a well-typed program cannot "go wrong" at run-time. Traditionally, a strongly typed program is first type-checked, then compiled, and then executed — but the situation for Template Haskell is a little more complicated. For example consider again our very first example:

```
$(printf "Error: %s on line %d") "urk" 341
```

It cannot readily be type-checked in this form, because the type of the spliced expression depends, in a complicated way, on the value of its string argument. So in Template Haskell type checking takes place in stages:

- First type check the body of the splice; in this case it is `(printf "Error: %s on line %d") :: Expr`.
- Next, compile it, execute it, and splice the result in place of the call. In our example, the program now becomes:

```
(\ s0 -> \ n1 ->
   "Error: " ++ s0 ++ " on line " ++ show n1)
      "urk" 341
```

- Now type-check the resulting program, *just as if the programmer had written that program in the first place*.

Hence, type checking is intimately interleaved with (compile-time) execution.

Template Haskell is a compile-time only meta-system. The meta-level operators (brackets, splices, reification) should not appear in the code being generated. For example, `[| f [| 3 |] |]` is illegal. There are other restrictions as well. For example, this definition is illegal (unless it is inside a quotation):

```
f x = $(zipN x)
```

Why? Because the "`$`" says "evaluate at compile time and splice", but the value of `x` is not known until `f` is called. This is a common staging error.

To enforce restrictions like these, we break the static-checking part of the compiling process into three states. *Compiling* (C) is the state of normal compilation. Without the meta-operators the compiler would always be in this state. The compiler enters the state *Bracket* (B) when compiling code inside quasi-quotes. The compiler enters the state *Splicing* (S) when it encounters an expression escape inside quasi-quoting brackets. For example, consider:

```
f :: Int -> Expr
f x = [| foo $(zipN x) |]
```

The definition of `f` is statically checked in state C, the call to `foo` is typed in state B, but the call to `zipN` is typed in state S.

In addition to the states, we count levels, by starting in state 0, incrementing when processing under quasi-quotes, and decrementing when processing inside `$` or `splice`. The levels are used to distinguish a top-level splice from a splice inside quasi-quotes. For example

```
g x = $(h [| x*2 |])
```

The call to `h` is statically checked in state S at level -1, while the `x*2` is checked in state B at level 0. These three states and their legal transitions are reflected in Figure 1. Transitions not in the diagram indicate error transitions. It is tempting to think that some of the states can be merged together, but this is not the case. Transitions on `$` from state C imply compile-time computation, and thus require more complicated static checking (including the computation itself!) than transitions on `$` from the other states.

The rules of the diagram are enforced by weaving them into the type checker. The formal typing judgments of the type checker are given in Figure 2; they embody the transition diagram by supplying cases only for legal states. We now study the rules in more detail.

7.1 Expressions

We begin with the rules for expressions, because they are simpler; indeed, they are just simplifications of the well-established rules for MetaML [16]. The type judgment rules for expressions takes the conventional form

$$\Gamma \vdash_s^n e : \tau$$

where Γ is an environment mapping variables to their types and binding states, e is an expression, τ is a type. The state s describes the *state* of the type checker, and n is the level, as described above.

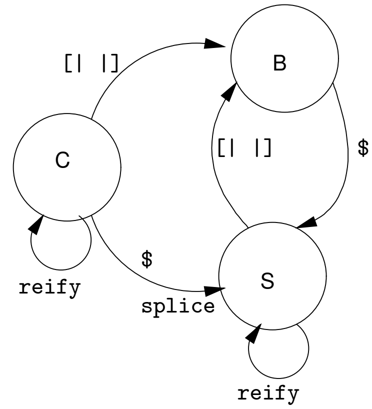

Figure 1. Typing states for Template Haskell

Rule BRACKET says that when in one of the states C or S, the expression `[|e|]` has type Q Exp, *regardless of the type of e*. However, notice that e is still type-checked, but in a new state B, and we increment the level. This reflects the legal transitions from Figure 1, and emphasizes that we can only use the BRACKET typing rule when in one of the listed states.

Type checking the term `e` detects any internal type inconsistencies right away; for example `[| 'a' + True |]` would be rejected immediately. This represents an interesting design compromise: meta-functions, including the code fragments that they generate, are statically checked, but that does not guarantee that the meta-function can produce only well-typed code, so completed splices are re-checked. We believe this is a new approach to typing meta-programs. This approach catches many errors as early as possible, avoids the need for using dependent types, yet is still completely type-safe.

Notice, too, that there is no rule for quasi-quotes in state B – quasi-quotes cannot be nested, unlike multi-stage languages such as MetaML.

Rule ESCB explains how to type check a splice `$`e inside quasi-quotes (state B). The type of e must be Q Exp, but that tells us nothing about the type of the expression that e will evaluate to; hence the use of an unspecified τ. There is no problem about soundness, however: the expression in which the splice sits will be type-checked later.

Indeed, that is precisely what happens in Rule ESCS, which deals with splicing when in state C. The expression e is type checked, and then evaluated, to give a new expression e'. This expression is then type checked from scratch (in state C), just as if the programmer had written it in the first place.

Rules LAM and VAR deal with staging. The environment Γ contains assumptions of the form $(x : (\sigma, m))$, which records not only x's type but also the level m at which it was bound (rule LAM). We think of this environment as a finite function. Then, when a variable x is used at level n, we check that n is later than ($\geq$) its binding level, m (rule VAR).

7.2 Declarations

Figure 2 also gives the rules for typing declarations, whose judgments are of form:

$$\Gamma \vdash_s^n decl : \Gamma'$$

States: $s \subseteq C, B, S$

EXPRESSIONS: $\Gamma \vdash_s^n \; expr : \tau$

$$\frac{\Gamma \vdash_B^{n+1} \; e : \tau}{\Gamma \vdash_{C,S}^n \; \texttt{[|}e\texttt{|]} : \mathsf{Q\ Exp}} \;\text{BRACKET}$$

$$\frac{\Gamma \vdash_S^n \; e : \mathsf{Q\ Exp}}{\Gamma \vdash_B^{n+1} \; \$e : \tau} \;\text{ESCB} \qquad \frac{\Gamma \vdash_C^0 \; e' : \tau \quad \mathsf{runQ}\; e \mapsto e' \quad \Gamma \vdash_S^0 \; e : \mathsf{Q\ Exp}}{\Gamma \vdash_C^0 \; \$e : \tau} \;\text{ESCS}$$

$$\frac{x \in \Gamma}{\Gamma \vdash_{C,S}^n \; \texttt{reifyDecl} \; x \; : \mathsf{Q\ Dec}} \;\text{REIFYDECL}$$

$$\frac{\Gamma; (x : (\tau_x, n)) \; \vdash_s^n \; e : \tau}{\Gamma \vdash_s^n \; \backslash\, x \,\texttt{->} e : \tau_x \to \tau} \;\text{LAM} \qquad \frac{\Gamma\, x = (\tau, m) \quad n \geq m}{\Gamma, \vdash_s^n \; x : \tau} \;\text{VAR}$$

DECLARATIONS: $\Gamma \vdash_s^n \; decl : \Gamma' \qquad \Gamma \vdash_s^n \; [decl] : \Gamma'$

$$\frac{\Gamma; (x : (\tau_1, n)); (f : (\tau_1 \to \tau_2, n)) \; \vdash_s^n \; e : \tau_2}{\Gamma \vdash_s^n \; f\, x = e : \{(f : \tau_1 \to \tau_2)_s\}} \;\text{FUN}$$

$$\frac{\Gamma \vdash_C^0 \; [d_1, \ldots, d_n] : \Gamma' \quad \mathsf{runQ}\; e \mapsto [d_1, \ldots, d_n] \quad \Gamma \vdash_C^0 \; e : \mathsf{Q\ [Dec]}}{\Gamma \vdash_C^0 \; \mathsf{splice}\; e : \Gamma'} \;\text{SPLICE}$$

Figure 2. Typing rules for Template Haskell

Here, Γ is the environment in which the declarations should be checked, while Γ' is a mini-environment that gives the types of the variables bound by *decl*[4].

Most rules are quite conventional; for example, Rule FUN explains how to type function definitions. The rule for splicing is the interesting one, and it follows the same pattern as for splicing expressions. First type-check the spliced expression e, then run it, then typecheck the declarations it returns.

The ability to generate a group of declarations seems to be of fundamental usefulness, but it raises an interesting complication: *we cannot even resolve the lexical scoping of the program, let alone the types, until splicing has been done*.

For example, is this program valid?

```
splice (genZips 20)
foo = zip3 "fee" "fie" "fum"
```

Well, it is valid if the splice brings `zip3` into scope (as we expect it to do) and not if it doesn't. Similar remarks naturally apply to the `instance` declaration produced by the `genEq` function of Section 5.1. If the module contains several `splice`s, it may not be at all obvious in which order to expand them.

We tackle this complication by assuming that the programmer intends the `splice`s to be expanded top-to-bottom. More precisely, to type-check a group of declarations $[d_1, \ldots, d_N]$, we follow the following procedure:

- Group the declarations as follows:

$$\begin{array}{c} [d_1, \ldots, d_a] \\ \texttt{splice}\; e_a \\ [d_{a+2}, \ldots, d_b] \\ \texttt{splice}\; e_b \\ \ldots \\ \texttt{splice}\; e_z \\ [d_{z+2}, \ldots, d_N] \end{array}$$

  where the only `splice` declarations are the ones indicated explicitly, so that each group $[d_1, \ldots, d_a]$, etc, are all ordinary Haskell declarations.
- Perform conventional dependency analysis, followed by type checking, on the first group. All its free variables should be in scope.
- In the environment thus established, type-check and expand the first `splice`.
- Type-check the result of expanding the first `splice`.
- In the augmented environment thus established, type-check the next ordinary group,
- And so on.

It is this algorithm that implements the judgment for declaration lists that we used in the rule SPLICE:

$$\Gamma \vdash_s^n \; [d_1, \ldots, d_n] : \Gamma'$$

7.3 Restrictions on declaration splicing

Notice that the rule for SPLICE assumes that we are in state C at level 0. We do not permit a declaration `splice` in any other state. For example, we do not permit this:

```
f :: Int -> Expr
f x = [| let
           splice (h x)
         in (p,q)
      |]
```

where `h :: Int -> Decl`. When type-checking `f` we cannot run the computation `(h x)` because `x` is not known yet; but until we have run `(h x)` we do not know what the `let` binds, and so we cannot sensibly type-check the body of the `let`, namely `(p,q)`. It would be possible to give up on type-checking the body since, after all, the result of every call to `f` will itself be type-checked, but the logical conclusion of that line of thought would be give up on type-checking the body of any quasi-quote expression. Doing so would be sound, but it would defer many type errors from the definition site of the meta-function to its call site(s). Our choice, pending further experience, is to err on the side of earlier error detection.

If you want the effect of the `f` above, you can still get it by dropping down to a lower level:

```
f :: Int -> Expr
f x = letE (h x) (tup [var "p", var "q"])
```

In fact, we currently restrict `splice` further: it must be a *top-level* declaration, like Haskell's `data`, `class`, and `instance` declarations. The reason for this restriction concerns usability rather than technical complexity. Since declaration splices introduce unspecified new bindings, it may not be clear where a variable that occurs in the original program is bound. The situation is similar for Haskell's existing `import` statements: they bring into scope an unspecified

[4] A single Haskell declaration can bind many variables.

collection of bindings. By restricting `splice` to top level we make a worthwhile gain: *given an occurrence of* `x`*, if we can see a lexically enclosing binding for* `x`*, that is indeed* `x`*'s binding*. A top level `splice` cannot hide another top-level binding (or import) for `x` because Haskell does not permit two definitions of the same value at top level. (In contrast, a nested `splice` could hide the enclosing binding for `x`.) Indeed, one can think of a top-level `splice` as a kind of programmable `import` statement.

8 The quotation monad revisited

So far we have used the quotation monad only to generate fresh names. It has other useful purposes too, as we discuss in this section.

8.1 Reification

Reification is Template Haskell's way of allowing the programmer to query the state of the compiler's internal (symbol) tables. For example, the programmer may write:

```
module M where

data T a = Tip a | Fork (T a) (T a)

repT :: Decl
repT = reifyDecl T

lengthType :: Type
lengthType = reifyType length

percentFixity :: Q Int
percentFixity = reifyFixity (%)

here :: Q String
here = reifyLocn
```

First, the construct `reifyDecl T` returns a computation of type `Decl` (i.e. `Q Dec`), *representing* the type declaration of `T`. If we performed the computation `repT` (perhaps by writing `$repT`) we would obtain the `Dec`:

```
Data "M:T" ["a"]
  [Constr "M:Tip" [Tvar "a"],
   Constr "M:Fork"
      [Tapp (Tcon (Name "M:T")) (Tvar "a"),
       Tapp (Tcon (Name "M:T")) (Tvar "a")]]
  []
```

We write "`M:T`" to mean unambiguously "the `T` that is defined in module `M`" — we say that `M:T` is its *original name*. Original names are not part of the syntax of Haskell, but they are necessary if we are to describe (and indeed implement) the meta-programming correctly. We will say more about original names in Section 9.1.

In a similar way, `reifyDecl f`, gives a data structure that represents the value declaration for `f`; and similarly for classes. Indeed, reification provides a general way to get at compile-time information. The construct `reifyType length` returns a computation of type `Type` (i.e. `Q Typ`) representing the compiler's knowledge about the type of the library function `length`. Similarly `reifyFixity` tells the fixity of its argument, which is useful when figuring out how to print something. Finally, `reifyLocn`, returns a computation with type `Q String`, which represents the location in the source file where the `reifyLocn` occurred. Reify always returns a computation, which can be combined with other computations at compile-time. Reification is a language construct, not a function; you cannot say `(map reifyType xs)`, for example.

It is important that reification returns a result in the quotation monad. For example consider this definition of an assertion function:

```
assert :: Expr        -- Bool -> a -> a
assert = [| \ b r ->
            if b then r else
              error ("Assert fail at "
                      ++ $reifyLocn |]
```

(Notice the comment giving the type of the expression generated by `assert`; here is where the more static type system of MetaML would be nicer.) One might invoke `assert` like this:

```
find xs n = $assert (n<10) (xs !! n)
```

When the `$assert` splice is expanded, we get:

```
find xs n
  = (\ b r -> if b then r else
              error ("Assert fail at " ++
                     "line 22 of Foo.hs"))
    (n < 10) (xs !! n)
```

It is vital, of course, that the `reifyLocn` captures the location of the *splice site* of `assert`, rather than its *definition site* — and that is precisely what we achieve by making `reifyLocn` return a computation. One can take the same idea further, by making `assert`'s behaviour depend on a command-line argument, analogous to `cpp`'s command mechanism for defining symbols `-Dfoo`:

```
cassert :: Expr       -- Bool -> a -> a
cassert = do { mb <- reifyOpt "DEBUG"
             ; if isNothing mb then
                  [| \b r -> r |]
               else
                  assert }
```

Here we assume another reification function `reifyOpt :: String -> Maybe String`, which returns `Nothing` if there is no `-D` command line option for the specified string, and the defined value if there is one.

One could go on. It is not yet clear how much reification can or should be allowed. For example, it might be useful to restrict the use of `reifyDecl` to type constructors, classes, or variables (e.g. function) declared at the top level in the current module, or perhaps to just type constructors declared in `data` declarations in imported modules. It may also be useful to support additional kinds of reification making other compiler symbol table information available.

8.2 Failure

A compile-time meta-program may *fail*, because the programmer made some error. For example, we would expect `$(zipN (-1))` to fail, because it does not make sense to produce an n-ary `zip` function for -1 arguments. Errors of this sort are due to inappropriate use, rather than bogus implementation of the meta-program, so the meta-programmer needs a way to cleanly report the error.

This is another place where the quotation monad is useful. In the case of `zipN` we can write:

```
zipN :: Int -> Expr
zipN n
  | n <= 1    = fail "Arg to zipN must be >= 2"
  | otherwise = ...as before...
```

The `fail` is the standard monadic `fail` operator, from class `Monad`, whose type (in this instance) is

```
fail :: String -> Q a
```

The compiler can "catch" errors reported via `fail`, and gracefully report where they occured.

8.3 *Input/output*

A meta-program may require access to input/output facilities. For example, we may want to write:

```
splice (genXML "foo.xml")
```

to generate a Haskell data type declaration corresponding to the XML schema stored in the file `"foo.xml"`, together with some boilerplate Haskell functions to work over that data type.

To this end, we can easily provide a way of performing arbitrary input/output from the quotation monad:

```
qIO :: IO a -> Q a
```

Naturally, this power is open to abuse; merely compiling a malicious program might delete your entire file store. Many compromise positions are possible, including ruling out I/O altogther, or allowing a limited set of benign operations (such as file reading only). This is a policy choice, not a technical one, and we do not consider it further here.

8.4 *Printing code*

So far we have only produced code in order to splice it into the module being compiled. Sometimes we want to write programs that generate a Haskell program, and put it in a file (rather than compiling it). The Happy parser generator is an example of an existing program that follows this paradigm. Indeed, for pedagogic reasons, it is extremely convenient to display the code we have generated, rather than just compile it.

To this end, libraries are provided that make `Exp`, `Dec`, etc instances of class `Show`.

```
instance Show Exp
instance Show Dec
..etc..
```

To display code constructed in the computational framework we supply the function `runQ :: Q a -> IO a`. Thus, if we compile and run the program

```
main =  do { e <- runQ (sel 1 3) ; putStr (show e) }
```

the output "`\x -> case x of (a,b,c) -> a`" will be produced. Notice the absence of the splicing `$`! (`sel` was defined in Section 4.)

8.5 *Implementing* `Q`

So far we have treated the `Q` monad abstractly, but it is easy to implement. It is just the `IO` monad augmented with an environment:

```
newtype Q a = Q (Env -> IO a)
```

The environment contains:

- A mutable location to serve as a name supply for `gensym`.
- The source location of the top-level splice that invoked the evaluation, for `reifyLocn`.
- The compiler's symbol table, to support the implementation of `reifyDecl`, `reifyFixity`, `reifyType`.
- Command-line switches, to support `reifyOpt`.

Other things could, of course, readily be added.

9 Quasi-quotes and Lexical Scoping

We have introduced the quasi-quote notation informally, and it is time to pay it direct attention.

The quasi-quote notation is a convenient shorthand for representing Haskell programs, and as such it is *lexically scoped*. More precisely:

> *every occurrence of a variable is bound to the value that is lexically in scope at the occurrence site in the original source program, before any template expansion.*

This obvious-sounding property is what the Lisp community calls *hygienic macros* [10]. In a meta-programming setting it is not nearly as easy to implement as one might think.

The quasi-quote notation is implemented on top of the quotation monad (Section 6), and we saw there that variables bound inside quasi-quotes must be renamed to avoid inadvertent capture (the `cross2a` example). But that is not all; what about variables bound *outside* the quasi-quotes?

9.1 *Cross-stage Persistence*

It is possible for a splice to expand to an expression that contains names that are not in scope where the splice occurs, and we need to take care when this happens. Consider this rather contrived example:

```
module T( genSwap ) where
  swap (a,b) = (b,a)
  genSwap x = [| swap x |]
```

Now consider a call of `genswap` in another module:

```
module Foo where
  import T( genSwap )
  swap = True
  foo  = $(genSwap (4,5))
```

What does the splice `$(genSwap (4,5))` expand to? It cannot expand to `(swap (4,5)` because, in module `Foo`, plain "`swap`" would bind to the boolean value defined in `Foo`, rather than the `swap` defined in module `T`. Nor can the splice expand to `(T.swap (4,5))`, using Haskell's qualified-name notation, because "`T.swap`" is not in scope in `Foo`: only `genSwap` is imported into `Foo`'s name space by `import T( genSwap )`.

Instead, we expand the splice to `(T:swap (4,5))`, using the *original name* `T:swap`. Original names were first discussed in Section 8.1 in the context of representations returned by `reify`. They solve a similar problem here. They are part of code representations that must unambiguously refer to (global, top-level) variables that may be hidden in scopes where the representations may be used. They are an extension to Haskell that Template Haskell uses to implement static scoping across the meta-programming extensions, and are not accessible in the ordinary part of Haskell. For example, one cannot write `M:map f [1,2,3]`.

The ability to include in generated code the value of a variable that exists at compile-time has a special name — *cross-stage persistence* — and it requires some care to implement correctly. We have just seen what happens for top-level variables, such as `swap`, but nested variables require different treatment. In particular, consider the status variable `x`, which is free in the quotation `[| swap x |]`. Unlike `swap`, `x` is not a top-level binding in the module `T`. Indeed, nothing other than `x`'s type is known when the module `T` is compiled. There is no way to give it an original name, since its value will vary with every call to `genSwap`.

Cross-stage persistence for this kind of variable is qualitatively different: it requires turning arbitrary values into code. For example, when the compiler executes the call `$(genSwap (4,5))`, it passes the *value* $(4,5)$ to `genSwap`, but the latter must return a *data structure* of type `Exp`:

```
App (Var "T:swap") (Tup [Lit (Int 4), Lit (Int 5)])
```

Somehow, the code for `genSwap` has to "lift" a value into an `Exp`. To show how this happens, here is what `genSwap` becomes when the quasi-quotes are translated away:

```
genSwap :: (Int,Int) -> Expr
genSwap x = do { t <- lift x
               ; return (App (Var "T:swap") t) }
```

Here, we take advantage of Haskell's existing type-class mechanism. `lift` is an overloaded function defined by the type class `Lift`:

```
class Lift t where
  lift :: t -> Expr
```

Instances of `Lift` allow the programmer to explain how to lift types of his choice into an `Expr`. For example, these ones are provided as part of Template Haskell:

```
instance Lift Int
  lift n = lit (Int n)

instance (Lift a,Lift b) => Lift (a,b) where
  lift(a,b) = tup [lift a, lift b]
```

Taking advantage of type classes in this way requires a slight change to the typing judgment VAR of Figure 2. When the stage s is B — that is, when inside quasi-quotes — and the variable x is bound outside the quasi quotes but not at top level, then the type checker must inject a type constraint `Lift` τ, where x has type τ. (We have omitted all mention of type constraints from Figure 2 but in the real system they are there, of course.)

To summarize, lexical scoping means that the free variables (such as `swap` and `x`) of a top-level quasi-quote (such as the right hand side of the definition of `genSwap`) are statically bound to the closure. They do not need to be in scope at the application site (inside module `Foo` in this case); indeed some quite different value of the same name may be in scope. There is nothing terribly surprising about this — it is simply lexical scoping in action, and is precisely the behaviour we would expect if `genSwap` were an ordinary function:

```
genSwap x = swap x
```

9.2 Dynamic scoping

Occasionally, the programmer may instead want a *dynamic* scoping strategy in generated code. In Template Haskell we can express dynamic scoping too, like this:

```
genSwapDyn x = [| $(var "swap") x |]
```

Now a splice site `$(genSwapDyn (4,5))` will expand to `(swap (4,5))`, *and this* `swap` *will bind to whatever* `swap` *is in scope at the splice site*, regardless of what was in scope at the definition of `genSwapDyn`. Such behaviour is sometimes useful, but in Template Haskell it is clearly flagged by the use of a string-quoted variable name, as in `(var "swap")`. All un-quoted variables are lexically scoped.

It is an open question whether this power is desirable. If not, it is easily removed, by making `var` take, and `gensym` return, an abstract type instead of a `String`.

9.3 Implementing quasi-quote

The quasi-quote notation can be explained in terms of original names, the syntax constructor functions, and the use of `gensym`, `do` and `return`, and the `lift` operation. One can think of this as a translation process, from the term within the quasi-quotes to another term. Figure 3 makes this translation precise by expressing the translation as an ordinary Haskell function. In this skeleton we handle enough of the constructors of `Pat` and `Exp` to illustrate the process, but omit many others in the interest of brevity.

The main function, `trE`, translates an expression inside quasi-quotes:

```
trE :: VEnv -> Exp -> Exp
```

The first argument is an environment of type `VEnv`; we ignore it for a couple more paragraphs. Given a term `t :: Exp`, the call `(trE cl t)` should construct another term `t' :: Exp`, *such that* `t'` *evaluates to* `t`. In our `genSwap` example, the compiler translates `genSwap`'s body, `[| swap x |]`, by executing the translation function `trE` on the arguments:

```
trE cl (App (Var "swap") (Var "x"))
```

The result of the call is the `Exp`:

```
(App (App (Var "app")
          (App (Var "var") (str "T:swap")))
     (App (Var "lift") (Var "x")))
```

which when printed as concrete syntax is:

```
app (var "T:swap") (lift x)
```

which is what we'd expect the quasi-quoted `[| swap x |]` to expand into after the quasi-quotes are translated out:

```
genSwap x = app (var "T:swap") (lift x)
```

(It is the environment `cl` that tells `trE` to treat `"swap"` and `"x"` differently.)

Capturing this translation process as a Haskell function, we write:

```
trE cl (App a b)
  = App (App (Var "app") (trans a)) (trans b)
trE cl (Cond x y z)
  = App (App (App (Var "cond") (trans x))
             (trans y))
        (trans z)
trE cl ... = ...
```

There is a simple pattern we can capture here:

```
trE cl (App a b)    = rep "app"  (trEs cl [a,b])
trE cl (Cond x y z) = rep "cond" (trEs cl [x,y,z])

trEs :: VEnv -> [Exp] -> [Exp]
trEs cl es = map (trE cl) es

rep :: String -> [Exp] -> Exp
rep f xs = apps (Var f) xs
  where apps f []     = f
        apps f (x:xs) = apps (App f x) xs
```

Now we return to the environment, `cl :: VEnv`. In Section 9.1 we discovered that variables need to be treated differently depending on how they are bound. The environment records this information, and is used by `trE` to decide how to translate variable occurrences:

```
type VEnv     = String -> VarClass
data VarClass = Orig ModName | Lifted | Bound
```

The `VarClass` for a variable v is as follows:

```
trE :: VEnv -> Exp -> Exp
trE cl (Var s)
  = case cl s of
      Bound    -> rep "var" [Var s]
      Lifted   -> rep "lift" [Var s]
      Orig mod -> rep "var" [str (mod++":"++s)])

trE cl e@(Lit(Int n)) = rep "Lit" [rep "Int" [e]]
trE cl (App f x)  = rep "app" (trEs cl [f,x])
trE cl (Tup es)   = rep "tup" [ListExp (trEs cl es)]
trE cl (Lam ps e) = Do (ss1 ++ [NoBindSt lam])
  where (ss1,xs) = trPs ps
        lam      = rep "lam" [ListExp xs,trE cl e]
trE cl (Esc e) = copy e
trE cl (Br e)  = error "Nested Brackets not allowed"

trEs :: VEnv -> [Exp] -> [Exp]
trEs cl es = map (trE cl) es

copy :: VEnv -> Exp -> Exp
copy cl (Var s)    = Var s
copy cl (Lit c)    = Lit c
copy cl (App f x)  = App (copy cl f) (copy cl x)
copy cl (Lam ps e) = Lam ps (copy cl e)
copy cl (Br e)     = trE cl e

trP :: Pat -> ([Statement Pat Exp Dec],Pat)
trP (p @ Pvar s)
  = ( [BindSt p (rep "gensym" [str s])]
    , rep "pvar" [Var s])
trP (Plit c)      = ([],rep "plit" [Lit c])
trP (Ptup ps)     = (ss,rep "ptup" [ListExp qs])
    where (ss,qs) = trPs ps
trP (Pcon c ps)   = (ss,rep "pcon" [str c,ListExp qs])
    where (ss,qs) = trPs ps
trP Pwild         = ([],Var "pwild")

trPs :: [Pat] -> ([Statement Pat Exp Dec],[Pat])
trPs ps = (concat ss,qs)
        where (ss,qs) = unzip (map trP ps)
```

Figure 3. The quasi-quote translation function `trExp`.

- `Orig` *m* means that the *v* is bound at the top level of module *m*, so that *m* : *v* is its original name.
- `Lifted` means that *v* is bound outside the quasi-quotes, but not at top level. The translation function will generate a call to `lift`, while the type checker will later ensure that the type of *v* is in class `Lift`.
- `Bound` means that *v* is bound inside the quasi-quotes, and should be alpha-renamed.

These three cases are reflected directly in the case for `Var` in `trE` (Figure 3).

We need an auxiliary function `trP` to translate patterns

```
trP :: Pat -> ([Statement Pat Exp Dec],Pat)
```

The first part of the pair returned by `trP` is a list of `Statements` (representing the gensym bindings generated by the translation). The second part of the pair is a `Pat` representing the alpha-renamed pattern. For example, when translating a pattern-variable (such as `x`), we get one binding statement (`x <- gensym "x"`), and a result (`pvar x`).

With `trP` in hand, we can look at the `Lam` case for `trE`. For a lambda expression (such as `\ f x -> f x`) we wish to generate a local `do` binding which preserves the scope of the quoted lambda.

```
do { f <- gensym "f"
   ; x <- gensym "x"
   ; lam [Pvar f,Pvar x] (app (var f) (var x))}
```

The bindings (`f <- gensym "f"; x <- gensym "x"`) and renamed patterns `[Pvar f,Pvar x]` are bound to the meta-variables `ss1` and `xs` by the call `trPs ps`, and these are assembled with the body `(app (var f) (var x))` generated by the recursive call to `trE` into the new `do` expression which is returned.

The last interesting case is the `Esc` case. Consider, for example, the term

```
[| (\ f -> f, \ f (x,y) -> f y $(w a) |]
```

The translation `trE` translates this as follows:

```
tup [ do { f <- gensym "f"
         ; lam [Pvar f] (var f) }
    , do { f <- gensym "f"
         ; x <- gensym "x"
         ; y <- gensym "y"
         ; lam [Pvar f,Ptup [Pvar x,Pvar y]]
               (app (app (var f) (var y)) (w a) }
    ]
```

Notice that the body of the splice `$(w a)` should be transcribed literally into the translated code as `(w a)`. That is what the `copy` function does.

Looking now at `copy`, the interesting case is when we reach a nested quasi-quotation; then we just resort back to `trE`. For example, given the code transformer `f x = [| $x + 4 |]`, the quasi-quoted term with nested quotations within an escape `[| \ x -> ( $(f [| x |]), 5 ) |]` translates to:

```
do { x <- gensym "x"
   ; lam [Pvar x] (tup [f (var x),lit (Int 5)])}
```

10 Related work

10.1 C++ templates

C++ has an elaborate meta-programming facility known as *templates* [1]. The basic idea is that static, or compile-time, computation takes place entirely in the *type system* of C++. A template class can be considered as a function whose arguments can be either types or integers, thus: `Factorial<7>`. It returns a type; one can extract an integer result by returning a `struct` and selecting a conventionally-named member, thus: `Factorial<7>::RET`.

The type system is rich enough that one can construct and manipulate arbitrary data structures (lists, trees, etc) *in the type system*, and use these computations to control what object-level code is generated. It is (now) widely recognized that this type-system computation language is simply an extraordinarily baroque functional language, full of *ad hoc* coding tricks and conventions. The fact that C++ templates are so widely used is very strong evidence of the need for such a thing: the barriers to their use are considerable.

We believe that Template Haskell takes a more principled approach to the same task. In particular, the static computation language is the *same* as the dynamic language, so no new programming idiom is required. We are not the first to think of this idea, of course: the Lisp community has been doing this for years, as we discuss next.

10.2 Scheme macros

The Lisp community has taken template meta-programming seriously for over twenty years [11], and modern Scheme systems support elaborate towers of language extensions based entirely on macros. Early designs suffered badly from the name-capture problem, but this problem was solved by the evolution of "hygienic" macros [10, 4]; Dybvig, Hieb and Bruggeman's paper is an excellent, self-contained summary of the state of the art [7].

The differences of vocabulary and world-view, combined with the subtlety of the material, make it quite difficult to give a clear picture of the differences between the Scheme approach and ours. An immediately-obvious difference is that Template Haskell is statically typed, both before expansion, and again afterwards. Scheme macro expanders do have a sort of static type system, however, which reports staging errors. Beyond that, there are three pervasive ways in which the Scheme system is both more powerful and less tractable than ours.

- Scheme admits new binding forms. Consider this macro call:

  ```
  (foo k (+ k 1))
  ```

  A suitably-defined macro `foo` might require the first argument to be a variable name, *which then scopes over the second argument*. For example, this call to foo might expand to:

  ```
  (lambda k (* 2 (+ k 1)))
  ```

  Much of the complexity of Scheme macros arises from the ability to define new binding forms in this way. Template Haskell does this too, but much more clumsily.

  ```
  $(foo "k" [| $(var "k") + 1 |])
  ```

  On the other hand, at least this makes clear that the occurrence `var "k"` is not lexically scoped in the source program.

  A declaration splice (`splice e`) *does* bind variables, but declaration splices can only occur at top level (outside quasi-quotes), so the situation is more tractable.

- Scheme macros have a special binding form (`define-syntax`) but the call site has no syntactic baggage. Instead a macro call is identified by observing that the token in the function position is bound by `define-syntax`. In Template Haskell, there is no special syntax at the definition site — template functions are just ordinary Haskell functions — but a splice (`$`) is required at the call site.

  There is an interesting trade-off here. Template Haskell "macros" are completely higher-order and first class, like any other function: they can be passed as arguments, returned as results, partially applied, constructed with anonymous lambdas, and so on. Scheme macros are pretty much first order: they must be called by name. (Bawden discussed first-class macros [2].)

- Scheme admits side effects, which complicates everything. When is a mutable value instantiated? Can it move from compile-time to run-time? When is it shared? And so on. Haskell is free of these complications.

10.3 MetaML and its derivatives

The goals of MetaML [16, 14, 13] and Template Haskell differ significantly, but many of the lessons learned from building MetaML have influenced the Template Haskell design. Important features that have migrated from MetaML to Template Haskell include:

- The use of a template (or Quasi-quote notation) as a means of constructing object programs.
- Type-safety. No program fragment is ever executed in a context before it is type-checked, and all type checking of constructed program fragments happens at compile-time.
- Static scoping of object-variables, including alpha renaming of bound object-variables to avoid inadvertent capture.
- Cross-stage persistence. Free object-variables representing run-time functions can be mentioned in object-code fragments and will be correctly bound in the scope where code is created, not where it is used.

10.3.1 MetaML

But there are also significant difference between Template Haskell and MetaML. Most of these differences follow from different assumptions about how meta-programming systems are used. The following assumptions, used to design Template Haskell, differ strongly from MetaML's.

- Users can compute portions of their program rather than writing them and should pay no run-time overhead. Hence the assumption that there are exactly two stages: Compile-time, and Run-time. In MetaML, code can be built and executed, even at run-time. In Template Haskell, code is meant to be compiled, and all meta-computation happens at compile-time.
- Code is represented by an algebraic datatype, and is hence amenable to inspection and case analysis. This appears at first, to be at odds with the static-scoping, and quasi-quotation mechanisms, but as we have shown can be accomplished in rather interesting way using monads.
- Everything is statically type-checked, but checking is delayed until the last possible moment using a strategy of just-in-time type checking. This allows more powerful meta-programs to be written without resorting to dependent types.
- Hand-written code is reifiable, I.e. the data representing it can be obtained for further manipulation. Any run-time function or data type definition can be reified – i.e. a data structure of its representation can be obtained and inspected by the compile-time functions.

Quasi-quotes in in MetaML indicate the boundary between stages of execution. Brackets and run in MetaML are akin to quote and eval in Scheme. In Template Haskell, brackets indicate the boundary between compile-time execution and run-time execution.

One of the main breakthroughs in the type system of MetaML was the introduction of quasi-quotes which respect both scoping *and* typing. If a MetaML code *generating* program is type-correct then so are all the programs it *generates* [16]. This property is crucial, because the generation step happens at run-time, and that is too late to start reporting type errors.

However, this security comes at a price: MetaML cannot express many useful programs. For example, the `printf` example of Section 2 cannot be typed by MetaML, because the type of the call to `printf` depends on the value of its string argument. One way to address this problem is using a *dependent* type system, but that approach has distinct disadvantages here. For a start, the programmer would have the burden of writing the function that transforms the format string to a type; and the type system itself becomes much more complicated to explain.

In Template Haskell, the second stage may give rise to type errors, but *they still occur at compile time*, so the situation is much less serious than with run-time code generation.

A contribution of the current work is the development of a semantics for quasi-quotes as monadic computations. This allows quasi-quotes to exist in a pure language without side effects. The process of generating fresh names is encapsulated in the monad, and hence quasi-quotes are referentially transparent.

10.3.2 MetaO'Caml

MetaO'Caml [3] is a staged ML implementation built on top of the O'Caml system. Like MetaML it is a run-time code generation system. Unlike MetaML it is a compiler rather than an interpreter, generating compiled byte-code at run-time. It has demonstrated some impressive performance gains for staged programs over their non-staged counterparts. The translation of quasi-quotes in a manner that preserves the scoping-structure of the quoted expression was first implemented in MetaO'Caml.

10.3.3 MacroML

MacroML [8] is a proposal to add compile-time macros to an ML language. MacroML demonstrates that even macros which implement new binding constructs can be given precise semantics as staged programs, and that macros can be strongly typed. MacroML allows the introduction of new hygenic local binders. MacroML supports only generative macros. Macros are *limited to constructing new code and combining code fragments*; they cannot analyze code fragments.

10.3.4 Dynamic Typing

The approach of just-in-time type-checking has its roots in an earlier study [15] of dynamic typing as staged type-inference. In that work, as well as in Template Haskell, typing of code fragments is split into stages. In Template Haskell, code is finally type-checked only at top-level splice points (`splice` and `$` in state C). In that work, code is type checked at all splice points. In addition, code construction and splice point type-checking were run-time activities, and significant effort was placed in reducing the run-time overhead of the type-checking.

11 Implementation

We have a small prototype that can read Template Haskell and perform compile-time execution. We are in the throes of scaling this prototype up to a full implementation, by embodying Template Haskell as an extension to the Glasgow Haskell Compiler, `ghc`.

The `ghc` implementation fully supports separate compilation. Indeed, when compiling a module `M`, only functions defined in modules compilied earlier than `M` can be executed a compile time. (Reason: to execute a function defined in `M` itself, the compiler would need to compile that function — and all the functions it calls — all the way through to executable code before even type-checking other parts of `M`.) When a compile-time function is invoked, the compiler finds its previously-compiled executable and dynamically links it (and all the modules and packages it imports) into the running compiler. A module consisting completely of meta-functions need not be linked into the executable built by the final link step (although `ghc --make` is not yet clever enough to figure this out).

12 Further work

Our design represents work in progress. Our hope is that, once we can provide a working implementation, further work can be driven directly by the experiences of real users. Meanwhile there are many avenues that we already know we want to work on.

With the (very important) exception of reifying data type definitions, we have said little about user-defined code manipulation or optimization, which is one of our advertised goals; we'll get to that.

We do not yet know how confusing the error messages from Template Haskell will be, given that they may arise from code that the programmer does not see. At the least, it should be possible to display this code.

We have already found that one often wants to get earlier type security and additional documentation by saying "this is an `Expr` whose type will be `Int`", like MetaML's type `<Int>`. We expect to add parameterised code types, such as `Expr Int`, using `Expr *` (or some such) to indicate that the type is not statically known.

C++ templates and Scheme macros have a lighter-weight syntax for *calling* a macro than we do; indeed, the programmer may not need to be aware that a macro is involved at all. This is an interesting trade-off, as we discussed briefly in Section 10.2. There is a lot to be said for reducing syntactic baggage at the call site, and we have a few speculative ideas for inferring splice annotations.

13 Acknowledgments

We would like particularly to thank Matthew Flatt for several long conversations in which we explored the relationship between Template Haskell and Scheme macros — but any errors on our comparison between the two remain ours. We also thank Magnus Carlsson, Fergus Henderson, Tony Hoare, Dick Kieburtz, Simon Marlow, Emir Pasalic, and the Haskell workshop referees, for their helpful comments on drafts of our work.

We would also like to thank the students of the class *CSE583 – Fundamentals of Staged Computation*, in the winter of 2002, who participated in many lively discussions about the uses of staging, especially Bill Howe, whose final project motivated Tim Sheard to begin this work.

The work described here was supported by NSF Grant CCR-0098126, the M.J. Murdock Charitable Trust, and the Department of Defense.

References

[1] A. Alexandrescu. *Modern C++ design*. Addison Wesley, 2001.

[2] A. Bawden. First-class macros have types. In *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 133–141, Boston, Jan. 2000. ACM.

[3] C. Calcagno, W. Taha, L. Huang, and X. Leroy. A bytecode-compiled, type-safe, multi-stage language. Technical report, Computer Science Department, Yale University, 2002.

[4] W. Clinger and J. Rees. Macros that work. In *19th ACM Symposium on Principles of Programming Languages (POPL'91)*, pages 155–162. ACM, Jan. 1991.

[5] O. Danvy. Functional unparsing. *Journal of Functional Programming*, 8, Nov. 1998.

[6] J. de Wit. A technical overview of Generic Haskell. Master's thesis, INF-SCR-02-03, Department of Information and Computing Sciences, Utrecht University, 2002.

[7] K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5:295–326, 1993.

[8] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-2001)*, New York, September 2001. ACM Press.

[9] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 Haskell Workshop, Montreal*, number NOTTCS-TR-00-1 in Technical Reports, Sept. 2000.

[10] E. Kohlbecker, D. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161. ACM, 1986.

[11] K. Pitman. Special forms in Lisp. In *ACM Symposium on Lisp and Functional Programming*, pages 179–187. ACM, 1980.

[12] A. D. Robinson. Impact of economics on compiler optimization. In *Proceedings of the ACM 2001 Java Grande Conference, Stanford*, pages 1–10. ACM, June 2001.

[13] T. Sheard. Accomplishments and research challenges in metaprogramming. In W. Taha, editor, *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01)*, volume 2196 of *LNCS*, pages 2–44, Berlin, September 2001. Springer Verlag. Invited talk.

[14] T. Sheard, Z. Benaissa, and M. Martel. *Introduction to Multistage Programming Using MetaML*. Pacific Software Research Center, Oregon Graduate Institute, 2nd edition, 2000. Available at `http://cse.ogi.edu/~sheard/papers/manual.ps`.

[15] M. Shields, T. Sheard, and S. L. Peyton Jones. Dynamic typing by staged type inference. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 289–302, San Diego, Jan. 1998. ACM.

[16] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, volume 32 of *SIGPLAN Notices*, pages 203–217. ACM, Amsterdam, June 1997.

[17] N. Winstanley. A type-sensitive preprocessor for Haskell. In *Glasgow Workshop on Functional Programming, Ullapool*, 1997.

A Library of Monadic Syntax Operators

```
-- The Monad
instance Monad Q
gensym :: String -> Q String
runQ :: Q a -> IO a
qIO :: IO a -> Q a

-- Type Synonyms. 3 letter Algebraic, 4 letter Monadic
type Expr = Q Exp
type Patt = Pat
type Decl = Q Dec
type Type = Typ

type Mat  = Match Pat Exp Dec
type Mtch = Match Patt Expr Decl

type Cls  = Clause Pat Exp Dec
type Clse = Clause = Patt Expr Decl

type Stm  = Statement Pat Exp Dec
type Stmt = Statement Patt Expr Decl

-- Lowercase Patterns
plit   = Plit;   pvar = Pvar
ptup   = Ptup;   pcon = Pcon
ptilde = Ptilde; paspat = Paspat
pwild  = Pwild

-- Lowercase Expressions
var s       = return(Var s)
con s       =  return(Con s)
lit c       = return(Lit c)
app x y     = do { a <- x; b <- y; return(App a b)}
lam ps e    = do { e2 <- e; return(Lam ps e2) }
lam1 p e    = lam [p] e
tup es      = do { es1 <- sequence es; return(Tup es1)}
doE ss      = do { ss1 <- stmtsC ss; return(Do ss1) }
comp ss     = do { ss1 <- stmtsC ss; return(Comp ss1) }
arithSeq xs = do { ys <- dotdotC xs; return(ArithSeq ys) }
listExp es  = do { es1 <- sequence es; return(ListExp es1)}
cond x y z  = do { a <- x; b <- y; c <- z;
                   return(Cond a b c)}
letE ds e   = do { ds2 <- sequence ds; e2 <- e;
                   return(Let ds2 e2) }
caseE e ms  = do { e1 <- e; ms1 <- mapM matchC ms;
                   return(Case e1 ms1) }

-- Helper functions for Auxillary Types
stmtC :: Stmt Pattern Expr Decl -> Q(Stmt Pat Exp Dec)
stmtC (NoBindSt e) = do { e1 <- e; return(NoBindSt e1) }
stmtC (BindSt p e) = do { e1 <- e; return(BindSt p e1) }
stmtC (ParSt zs)   = fail "No parallel comprehensions yet"
stmtC (LetSt ds)   = do { ds2 <- sequence ds;
                          return(LetSt ds2) }

stmtsC ss = sequence (map stmtC ss)

bodyC :: Body Expr -> Q(Body Exp)
bodyC (Normal e)   = do { e1 <- e; return(Normal e1) }
bodyC (Guarded ps) = do { ps1 <- mapM f ps;
                          return(Guarded ps1) }
     where f (g,e) = do { g1 <- g; e1 <- e; return(g1,e1) }

matchC :: Match Pattern Expr Decl -> Q(Match Pat Exp Dec)
matchC (p,b,ds)  = do { b1 <- bodyC b; ds1 <- sequence ds;
                        return(p,b1,ds1)}

dotdotC (From x) = do { a <- x; return(From a)}
dotdotC (FromThen x y)
  = do { a <- x; b <- y; return(FromThen a b)}
dotdotC (FromTo x y)
  = do { a <- x; b <- y; return(FromTo a b)}
dotdotC (FromThenTo x y z)
  = do { a <- x; b <- y; c <- z; return(FromThenTo a b c)}

-- Other useful functions
genPE s n = (map pvar ns,map var ns)
 where ns = [ s++(show i) | i <- [1..n]]

apps :: [Expr] -> Expr
apps [x]      = x
apps (x:y:zs) = apps ( [| $x $y |] : zs )

simpleM p e = (p,Normal e,[])
clauseC x   = matchC x
```

B Algebraic Datatype Representation of Haskell

```
module SimpleData where

data Lit = Int Int | Char Char

data Pat
  = Plit Lit                        -- { 5 or 'c' }
  | Pvar String                     -- { x }
  | Ptup [Pat]                      -- { (p1,p2) }
  | Pcon String [Pat]               -- data T1 = C1 t1 t2; {C1 p1 p1} = e
  | Ptilde Pat                      -- { ~p }
  | Paspat String Pat               -- { x @ p }
  | Pwild                           -- { _ }

type Match p e d  = ( p ,Body e,[d])   -- case e of { pat -> body where decs }
type Clause p e d = ([p],Body e,[d])   -- f { p1 p2 = body where decs }

data Exp
  = Var String                      -- { x }
  | Con String                      -- data T1 = C1 t1 t2; p = {C1} e1 e2
  | Lit Lit                         -- { 5 or 'c'}
  | App Exp Exp                     -- { f x }
  | Lam [Pat] Exp                   -- { \ p1 p2 -> e }
  | Tup [Exp]                       -- { (e1,e2) }
  | Cond Exp Exp Exp                -- { if e1 then e2 else e3 }
  | Let [Dec] Exp                   -- { let x=e1;   y=e2 in e3 }
  | Case Exp [Match Pat Exp Dec]    -- { case e of m1; m2 }
  | Do [Statement Pat Exp Dec]      -- { do { p <- e1; e2 }  }
  | Comp [Statement Pat Exp Dec]    -- { [ (x,y) | x <- xs, y <- ys ] }
  | ArithSeq (DotDot Exp)           -- { [ 1 ,2 .. 10 ] }
  | ListExp [ Exp ]                 -- { [1,2,3] }

data Body e
  = Guarded [(e,e)]                 -- f p { | e1 = e2 | e3 = e4 } where ds
  | Normal e                        -- f p = { e } where ds

data Statement p e d
  = BindSt p e                      -- { p <- e }
  | LetSt [ d ]                     -- { let f x = e }
  | NoBindSt e                      -- { print e }
  | ParSt [[Statement p e d]]       -- { x <- xs | y <- ys, z <- zs }

data DotDot e
  = From e                          -- [ { 0 .. } ]
  | FromThen e e                    -- [ { 0,1 .. } ]
  | FromTo e e                      -- [ { 0 .. 10 } ]
  | FromThenTo e e e                -- [ { 0,2 .. 12 } ]

data Dec
  = Fun String [Clause Pat Exp Dec]         -- { f p1 p2 = b where decs }
  | Val Pat (Body Exp) [Dec]                -- { p = b where decs }
  | Data String [String] [Constr] [String]  -- { data T x = A x | B (T x) deriving (Z,W)}
  | Class [Typ] Typ [Dec]                   -- { class Eq a => Eq [a] where ds }
  | Instance [Typ] Typ [Dec]                -- { instance Show w => Show [w] where ds }
  | Proto Name Typ                          -- { length :: [a] -> Int }

data Constr = Constr String [Typ]

data Tag
  = Tuple Int                      -- (,,)
  | Arrow                          -- (->)
  | List                           -- ([])
  | Name String deriving Eq        -- Tree

data Typ
  = Tvar String                    -- a
  | Tcon Tag                       -- T or [] or (->) or (,,) etc
  | Tapp Typ Typ                   -- T a b

-- Left out things implicit parameters, sections, complicated literals, default declarations
```