Template Haskell: A Report From The Field

Ian Lynagh
Programming Tools Group
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD, UK
ian.lynagh@comlab.ox.ac.uk

ABSTRACT

We have now had a year since the introductory Template Haskell paper was written. In this paper I look back upon what I have used it for in this time, the problems I have encountered and where I see myself going in the future.

1. INTRODUCTION

It has been a year since Template Haskell was introduced to the world in Tim Sheard and Simon Peyton-Jones' paper [10]. I have spent much of the year working with Template Haskell, and in this paper I share my experiences with you.

I will go through the applications I have worked on in approximately chronological order. For each one I will give an overview of what was involved, a summary of the non-Template Haskell alternatives around, the advantages of the Template Haskell method and a list of the problems I encountered. Finally I draw some conclusions in Section 7.

2. CURSES

My first application of Template Haskell was a rather cheeky one, using the ability to do IO in the quotation monad in a way the designers hadn't anticipated. The inspiration came from Hetris, a simple Tetris-like game I had previously written in Haskell to demonstrate something else.

2.1 The application

The game used curses, a C library which allows you to easily and portably draw on text terminals. To use the C library from Haskell I wrote a thin interface using the FFI; this leads to a load of code duplication, and it gets particularly bad when you need a small C stub function to sit in the middle.

For example, to get the graphics characters first requires a foreign import line in the Haskell source, such as shown in Figure 1(a). Then we need C wrapper functions to turn each of the constants into functions, shown in Figure 1(b), and a matching set of function prototypes in a header file, shown in Figure 1(c). Not only is there lots of repetition in the structure of the code but it is split over 3 files.

But there is another way! First consider what we might do without Template Haskell—we could write a small Haskell program that has a list of symbols and writes a Haskell source, C source and C header file out. We would then compile these as normal.

But with Template Haskell we can splice declarations, which includes FFI foreign imports, into a Haskell source file at compile time. We could therefore remove the bit of program that writes out the Haskell source file and instead have a splice which also has a list of the symbols and constructs a list of corresponding foreign imports to be spliced in.

This seems like a step backwards—now we are back to having duplication of the list and the code split across multiple files. But we can do arbitrary computation inside a splice, and what's more we can do IO too! We can thus build the C source and header files as Haskell strings, write them out to files and even call `gcc` to compile it inside a splice! This is shown in Figure 2.

2.2 Other approaches

A number of have been written to solve this and similar problems. The general approach used is to introduce special syntax for calling foreign functions; a preprocessor then works whatever magic is necessary to produce Haskell code that can be compiled as normal. The two most common programs that work in this way are Green Card [9] and C->Haskell [1].

The other common alternative is HaskellDirect [4]. This takes an IDL [2] description of an interface and automatically generates the marshalling code for you.

One advantage of the Template Haskell approach is that compilation is performed in a single step so we don't have to worry about preprocessors and intermediate temporary files. This can be particularly important when the preprocessor's parser gets confused as can sometimes happen when other extensions are used in the code. Also, as the parsing is done for us the implementation is easier to write; type checking is also done so some errors will be identified more easily. Finally, although the syntax isn't quite as concise it is both

```
foreign import ccall unsafe "wrap.h acs_block" acs_block :: IO ChType
foreign import ccall unsafe "wrap.h acs_hline" acs_hline :: IO ChType
foreign import ccall unsafe "wrap.h acs_ulcorner" acs_ulcorner :: IO ChType
```

(a) Haskell foreign imports

```
#include "wrap.h"
chtype acs_block(void) { return ACS_BLOCK; }
chtype acs_hline(void) { return ACS_HLINE; }
chtype acs_ulcorner(void) { return ACS_ULCORNER; }
```

(b) C wrappers

```
#include <curses.h>
chtype acs_block(void);
chtype acs_hline(void);
chtype acs_ulcorner(void);
```

(c) C headers

Figure 1: Original curses interface code

```
$(
    let fs = ["acs_block", "acs_hline", "acs_ulcorner"]
        imp f = Foreign (
                    Import CCall Unsafe ("wrap.h " ++ f) f
                    (Tapp (Tcon (TconName "IO")) (Tcon (TconName "ChType")))
                )
        h f = "chtype " ++ f ++ "(void);\n"
        c f = "chtype " ++ f ++ "(void) {\n"
           ++ "    return " ++ map toUpper f ++ ";\n"
           ++ "}\n"
        c_header = "#include \"wrap.h\"\n"
        h_header = "#include <curses.h>\n"
    in do qIO $ writeFile "wrap.h" $ h_header ++ concatMap h fs
          qIO $ writeFile "wrap.c" $ c_header ++ concatMap c fs
          qIO $ system "gcc -Wall -pedantic -c wrap.c -o wrap.o"
          return $ map imp fs
 )
```

Figure 2: Template Haskell curses interface code

more familiar and more general.

2.3 Problems encountered

This worked out quite nicely, but there are a few improvements that could be made. First let us concentrate on the part of the splice that build the foreign import declaration, shown in Figure 3(a). It would be a lot easier to see what is going on if we could write this as a reified declaration with the symbol name spliced in, as shown in Figure 3(b). However, it is not possible in the current design to splice strings in like this.

It would also be nice if we could add the declarations we're splicing in to the export list. As it stands, if we want to export these declarations, we either have to write the list out again in the source code or to have all declarations in the module exported. For this we would want a function `addExport :: String -> Q ()`.

Finally, if we wanted to splice in a number of such groups it would be convenient to have a different splice for each group. However, it might also be useful to be able to share the functions used to build the Template Haskell datastructures or C source files between the splices. The current implementation wouldn't allow such shared functions to be defined in the same module as the splices, but it would seem like overkill to create a module just for a couple of tiny helper functions. In such a case it would be preferable to pay the price of compiling the code to bytecode to be able to use it so that the code can all be kept together in one file. A static analysis of the code could determine which functions might be called so no more work need be done than is necessary.

3. DERIVING INSTANCES OF CLASSES

Haskell allows the programmer to specify, when he defines a new datatype, that the implementation should provide an instance of any of a handful of standard classes. Such instances all follow a standard form. However, if you want to define such an instance of a datatype defined in a library you are using, to define a number of instances in a different form or define an instance of any other class then with plain old Haskell 98 you are out of luck—you have to write each one out separately. Template Haskell provides a bit more flexibility.

3.1 The application

With Template Haskell we can write a function that takes the datastructure representing a datatype declaration and returns a datastructure representing an instance declaration for that datatype in the class we are interested in. I have found this technique to be more generally applicable than the others—it came in handy many times when implementing the later applications.

As an example, consider a function that makes the same `Show` instances that can be derived by the implementation. In other words, if a datatype is declared by

```
data Foo a = Foo String a
           | ...
```

then we want to create the instance

```
instance Show Foo where
    showsPrec p (Foo x0 x1)
        = showParen (p > 9) (showString "Foo" .
                             showChar ' ' .
                             showsPrec 10 x0 .
                             showChar ' ' .
                             showsPrec 10 x1)
    ...
```

Let us start with a function that makes a single clause of the instance given a single constructor. There are basically two styles that we can use: we could write the code using the Template Haskell datastructures directly; or we could use the code fragment notation. The first of these options might seem the more tempting as it is familiar and we wouldn't have to worry about learning the new syntax. However, consider

```
show_var v = App (App (Var "showsPrec")
                      (Lit (Integer 10)))
                 (Var v)
```

versus the equivalent[1]

```
show_var v = [| showsPrec 10 $(var v) |]
```

The more familiar style is not only more verbose but it is much harder to see at a glance what the expression does. We therefore use the new quotation fragment style where possible.

The code then follows straightforwardly; Figure 4 shows it for the case of simple non-infix constructors. The code that uses this function to build up complete instances, shown in Figure 5, follows equally naturally. We can then splice in `Show` instances for a couple of datatypes with

```
$(
    do dec_Data1 <- reifyDecl Data1
       dec_Data2 <- reifyDecl Data2
       mapM derive_show [dec_Data1, dec_Data2]
 )
```

3.2 Other approaches

Efforts have been made in this direction in the Haskell world for some time. Strafunski [6] and DrIFT [7] are amongst those that have stood the test of time, and more recently work has concentrated on Generic Haskell [5].

The advantages of Template Haskell are largely the same as in the previous section; although these tools are fine for what they do, rather than learning special syntax for just this task we can use Template Haskell for many.

[1] This is not quite equivalent as it returns a value in the `Q` monad, but the distinction isn't important to the point being made here.

```
$(
    let fs = ["acs_block", "acs_hline", "acs_ulcorner"]
        imp f = Foreign (
                    Import CCall Unsafe ("wrap.h " ++ f) f
                    (Tapp (Tcon (TconName "IO")) (Tcon (TconName "ChType")))
                )
    in return $ map imp fs
 )
```

(a) Current foreign import declaration construction

```
$(
    let fs = ["acs_block", "acs_hline", "acs_ulcorner"]
        imp f = [d| foreign import ccall unsafe $("wrap.h " ++ f) $f :: IO ChType |]
    in liftM concat $ mapM imp fs
 )
```

(b) Wanted foreign import declaration construction

Figure 3: Creating foreign import declarations

```
mk_show_clause :: Con -> ClauseQ
mk_show_clause (Constr c ts)
 = let shown_c = [| showString $(lift $ post_colon c) |]
       vars = take (length ts) (map (('x':) . show) [0..])
       show_var v = [| showsPrec 10 $(var v) |]
       shown = foldl (\e v -> [| $e . showChar ' ' . $(show_var v) |]) shown_c vars
       want_parens = [| $(var "p") > 9 |]
       shown_with_parens = [| showParen $want_parens $shown |]
   in clause [Pvar "p", Pcon c (map Pvar vars)] (normal shown_with_parens) []

post_colon :: String -> String
post_colon s = case break (== ':') s of
                   (x, "") -> x
                   (_, _:cs) -> cs
```

Figure 4: Making `Show` clauses

```
derive_show :: Dec -> DecQ
derive_show (Data _ name tyvars constrs _) = inst cxt inst_of decs
  where cxt = mapM (\v -> tapp tcon_show (tvar v)) tyvars
        tcon_show = tcon (TconName "Show")
        typ = foldl (\t s -> tapp t (tvar s)) (tcon (TconName name)) tyvars
        inst_of = tapp tcon_show typ
        decs = [fun "showsPrec" $ map mk_show_clause constrs]
```

Figure 5: Making `Show` instances

3.3 Problems encountered

The major issue here is that `reifyDecl` is syntax, not a function, i.e. you have to put the name of the declaration you want to reify in the source code. If it was a function that took a string then it would be possible to not only create an instance for what you are told, but also to create instances for any other types needed by such an instance. For example, when creating a `Show` instance that does the same as the instance that Haskell 98 would derive, you could also create instances for all of the types used by constructors of the datatype that do not already have an instance. It would also need to be possible to ask whether there was already an instance of a class for a particular datatype.

Another problem, that was neatly skimmed over in the description, is infix constructors. We need a slightly different clause—`showParen` needs to have a different argument—depending on the fixity of the of the constructor, but there is no way to find out what its fixity is. This boils down to a similar problem to the previous one—we need a function that takes an operator and returns its fixity.

The third problem is apparent when you compare Figure 4 to Figure 5. Although we aimed to use the reification and splicing brackets where possible, we weren't able to use them for building the instance header—they aren't general enough.

4. PRINTF

The next application was inspired by the original paper on Template Haskell. To demonstrate how to use Template Haskell the paper gives a tiny printf-style program as an example. I decided to try extending this to cover the full range of format strings the C printf function allows.

4.1 The application

The idea here is to take a "format string", a string which describes a show function, and to build and splice in the function that shows its arguments in this way. For example,

```
$(printf "%d %s") 58 "foo"
```

should produce the string "`58 foo`", as someone used to the C printf family of functions would expect.

We again have the option of building up the expression with the familiar Haskell syntax or the new reification brackets. Again we choose the latter on the grounds of conciseness and clarity. The code is not included here as it is large and very simple; the important point is the code construction that can be seen in Section 3.

There are two sub-problems we can tackle here. First, we have a specification for our function, the C printf function. By making use of the technique from Section 2 we can automatically generate test cases by writing, compiling and running C programs and reading their output. We can then use HUnit to run them all and check that our printf function is behaving correctly.

The second sub-problem is simplification. There are a number of values which can either be given as an argument or explicitly put in the format string; for example, `"%5d"` takes a single integer and prints it as a field at least 5 characters wide, left padding with spaces as necessary, while `"%*d"` takes 2 integers and uses the first as the field width and the second as the number to be displayed. There are some cases where we know when building the expression what the result of some of the arithmetic will be or which branch of a conditional will be taken. Building the optimal expression straight off involves a huge number of cases and makes the code very hard to understand. Instead, we can take the simple expression built and walk over it, simplifying bottom up. For example,

```
simplify (Infix (Just (Lit (Integer x)))
                (Var "-")
                (Just (Lit (Integer y))))
    = Lit (Integer (x - y))
simplify (App e1 e2) = App (simplify e1)
                           (simplify e2)
...
```

4.2 Other approaches

A number of attempts to emulate printf in Haskell have been made. In a change from the norm it is the Template Haskell implementation this time that needs less syntactic overhead than a plain Haskell 98 implementation such as [8]. The reason for this is that we produce an expression that takes the arguments we are giving it rather than having to worry about wrapping the arguments or similar tricks. Depending on the details of a standard Haskell 98 implementation the type checker may not be able to check that the types of the arguments match those needed by the format string either, although this does allow dynamic format strings to be allowed.

However, various other extensions also permit better printf implementations. Dependent type would allow us to give a type to a printf function provided the string was known statically while with Generic Haskell we could write a function that took any argument and failed at runtime if given an argument incompatible with its format string. Again the advantage of Template Haskell is that it is applicable to more problems than these other solutions.

4.3 Problems encountered

As perhaps should be expected given it was used as an example in the paper, once you have got to grips with the new syntax there are no real problems with this application. The only minor issue is that the simplification function has to be written using the more verbose syntax, at least on the left hand side. It would be hard to make a pattern matching form of reification brackets that was expressive enough to be useful.

5. FRASKELL

The idea behind this next application is to optimise programs by unrolling fixed-depth recursions. There a number of such applications; for example, many cryptographic algorithms perform a fixed number of rounds for each block they process. Here we take a Mandelbrot visualiser as the example we will work with.

5.1 The application

The definition of the Mandelbrot set is very simple, so a very simple function lies at the core of the visualiser:

```
type Iterations = Int
type T = Double
type C = Complex T
type Colour = (Int, Int, Int)

do_mb :: T -> Iterations -> C -> C -> Colour;
do_mb k i z xy
 | i > 255 = (0, 0, 0)
 | otherwise = if (realPart z')^2
                + (imagPart z')^2 > k^2
               then (0, 0, i)
               else do_mb k (i+10) z' xy
    where z' = z^2 + xy;
```

We know this function will always be called with `i` initially 0. What we want to do is to have this function automatically unrolled at compile time. We know that we will never need to unroll more than 30 levels deep, so we pass this to our unrolling function as a safety cut-off. We also pass it the index of the argument we are unrolling with respect to, 1 in this case, and the literal that we expect it to be called with. Modifying the code to do this is relatively low-impact:

```
type Iterations = Int
type T = Double
type C = Complex T
type Colour = (Int, Int, Int)

#ifdef TEMPLATE_HASKELL
$(
  do
     [typ, func] <- [d| {
#endif

do_mb :: T -> Iterations -> C -> C -> Colour;
do_mb k i z xy
 | i > 255 = (0, 0, 0)
 | otherwise = if (realPart z')^2
                + (imagPart z')^2 > k^2
               then (0, 0, i)
               else do_mb k (i+10) z' xy
    where z' = z^2 + xy;

#ifdef TEMPLATE_HASKELL
                      } |]
     let func' = unroll 30 1 (Integer 0) func
     return [ typ, func' ]
 )
#endif
```

Going into the details of how the code works is outside of the scope of this report, but intuitively it adds a clause to the function in which recursive calls are replaced by another copy of the function body. By simplifying at each stage eventually the expression being worked on will contain no recursive calls and we are done. The result will be nested conditionals along the lines of

```
do_mb :: T -> Iterations -> C -> C -> Colour;
do_mb k 0 z xy
 = let z1 = z^2 + xy in
   if (realPart z1)^2 + (imagPart z1)^2 > k^2
   then (0, 0, 0)
   else
   let z2 = z1^2 + xy in
   if ...
   ...
   else
   let z26 = z5^2 + xy in
   if (realPart z26)^2 + (imagPart z26)^2 > k^2
   then (0, 0, 250)
   else (0, 0, 255)
do_mb k i z xy
 | i > 255 = (0, 0, 0)
 | otherwise = if (realPart z')^2
                + (imagPart z')^2 > k^2
               then (0, 0, i)
               else do_mb k (i+10) z' xy
    where z' = z^2 + xy;
```

5.2 Other approaches

I am not aware of any equivalent packages for this, probably due to the difficulty of implementation and relatively limited applicability. Template Haskell makes things easier by type checking both the translation code and the input, as well as parsing the input for us.

5.3 Problems encountered

There is one major issue with Template Haskell as it is today that makes this application quite dubious. In order to know when to stop the recursion we need to simplify the expression that tells us whether we have a base case or a recursive case; in this example the key expression was `i > 255`. The way I chose to implement this was to simplify the entire expression; not only is this easier than finding only the right bit to simplify but it means we benefit from the precomputation of anything else that can be simplified. This means that, for example, whereever we see

```
simplify (Infix (Just (Lit (Integer x)))
                (Var "+")
                (Just (Lit (Integer y))))
```

we replace it with `Lit (Integer (x + y))`. This is fine the type of the expression is `Integer`, and probably OK if it is `Int`, `Float` or one of many other types[2]. The problem comes when it is used on a type where the properties we assume don't hold; for example, `fromInteger x + fromInteger y` may have been defined to be equal to `fromInteger (x * y)` for some reason. Our simplification will then change the meaning of the code! The underlying problem is that Template Haskell only gives the datastructure when we reify, no information about what the type of it or its components is. It is not clear what the best way for doing this is, but it must be done if it is to be possible to write safe program transformers with Template Haskell.

[2]It is possible that you can get different results with types other than `Integer` due to the difference in overflowing, but you are unlikely to want this with the standard Haskell 98 types.

6. PAN

My most recent application has been a reimplementation of Pan [3], producing what is essentially an optimiser for Haskell.

6.1 The application

The idea here might be described as extreme optimisation. The basic idea can be described as follows: Given a Haskell program, first inline absolutely everything; note that this precludes recursive functions. Then simplify as much as possible, for example constant folding and beta reduction. We also float conditionals up, so for example given the expression `(if g then 1 else 2) + 3` then we can float the conditional up through the addition operator to give us `if g then 1 + 3 else 2 + 3`. This can then be simplified to `if g then 4 else 5`. Finally we do common subexpression elimination, pulling repeated expressions out as let bindings.

Unlike the original Pan implementation, which transformed a Haskell-like language to C which was then compiled by a normal C compiler, we will transform a Haskell program to another Haskell program.

We get a headstart on a traditional Haskell implementation by getting the parsing for free; by surrounding the input with reification brackets the metaprogramming system will parse the user's program for us and give us the corresponding Template Haskell datastructure. Not only that, but it will even typecheck the input for us!

To do the common subexpression elimination we use the traditional method of building a DAG. Rather than doing the inlining, simplification and if floating as separate passes over the Template Haskell datastructures we can perform them as we build the DAG.

Template Haskell comes in handy on a few more occassions as we write the code. Most of them come under the technique discussed in Section 3, but there is one new technique worth mentioning. Before we can explain it, we first need to know how the DAG works.

Template Haskell gives us a number of datastructures, for example:

```
data Exp = App Exp Exp
         | Lam [Pat] Exp
         | Tup [Exp]
         ...
data Pat = Pvar String
         | Ptup [Pat]
         ...
```

We build the DAG as if we were doing so in a memory of some sort. We have a type `Addr`, which is really just an `Integer`, representing addresses in this memory, and at each address there can be a piece of one of the datastructures. We thus need a datatype for these pieces:

```
data Piece = AppP Addr Addr
           | LamP Addr Addr
           | TupP Addr
           ...
           | PvarP String
           | PtupP Addr
           ...
           | ConsP Addr Addr
           | NilP
```

Now, when we insert a new piece into the memory this may be a node which which can be simplified. However, in order to check this, and to find out what it can be simplified to, we need to check what pieces its children are, and in some cases their children, or even their children's children. If we were to write the code doing all these lookups by hand it would not be clear what was going on, so maintaining and debugging the transformations would be very difficult. Instead we use code fragments to specify the transformations we need, for example:

```
[| \x -> App (App (Var "GHC.Num:+") x)
              (Lit (Integer 0))
          --> x |]
[| \g x y z -> App (Cond g x y) z
          --> Cond g (App x z) (App y z) |]
```

We can then write a function that takes one of these fragments and returns the expression that performs the transformation on the memory as we need. The details are beyond the cope of this paper.

6.2 Other approaches

This application was inspired by Pan [3]. The main advantages Template Haskell gives us are, once more, parsing and typechecking of the input. It also makes transformations easier to write, as we have just seen.

6.3 Problems encountered

Again we suffer from the problem that, when we are transforming the input, we need to assume that integral types behave in the same way as integers. Also, along similar lines, it is sometimes necessary to know what can be inferred about the strictness of some parts of the expression in order to know if a transformation is valid. For example, the dual to the second of the earlier fragments, where if floating of the right-hand side of an application is done, is only correct if the left-hand side is strict in its argument. These problems are very similar, so if a solution can be found for one it probably also covers the other.

The other problem encountered is rather more subtle. Consider floating a conditional out of a tuple, so we might have

```
(if g then 1 else 2, 3)
```

and want to end up with

```
if g then (1, 3) else (2, 3)
```

The representation stored in the DAG is clearly of a valid Haskell datastructure before and after the complete transformation, but if we perform it stepwise then the conditional will be floated up through the list argument of the `Tup` constructor meaning the DAG doesn't represent a valid datastructure in the middle. This in itself is fine, there is no need for the memory to be valid before we have finished all our transformations and are flattening it again. However, we need to be able to specify the transformations for these steps, i.e. we want to be able to say

```
[| \g x y z -> (:) (Cond g x y) z
          --> Cond g ((:) x z) ((:) y z) |]
```

However, this fragment doesn't type check, and the metaprogramming system will type check anything we reify and give an error if it fails! One solution to this problem is to modify the system slightly, to make it possible to tell the system that it needn't typecheck certain reification brackets. It turns out to be very easy in the existing implementation to give an alternative end lexeme, so the following would be accepted without type checking:

```
[| \g x y z -> (:) (Cond g x y) z
          --> Cond g ((:) x z) ((:) y z) |n]
```

7. CONCLUSION

In some cases Template Haskell as-is works fine. In others it makes things "mostly possible", meaning that it is possible to write something that works 99% of the time but there are cases where correctness has had to be sacrificed as all the information needed wasn't available. Unfortunately no good solutions to these problems are obvious.

The other issues encountered have been restrictions on what is possible with the current system. As experience gathers it should become clear what extensions will prove the most useful; I hope that this report will be useful in making those decisions.

8. ACKNOWLEDGEMENTS

I would like to thank Oege de Moor and Ganesh Sittampalam for their discussions and advice during the writing of this paper.

This work, as part of my dPhil, was sponsored by a grant from Microsoft Research.

9. REFERENCES

[1] M. Chakravarty. C->haskell. URL: http://www.cse.unsw.edu.au/~chak/haskell/c2hs/.

[2] M. F. Coallier et al. Interface definition language. URL: http://www.omg.org/gettingstarted/omg_idl.htm.

[3] C. Elliott, S. Finne, and O. de Moor. Pan. URL: http://www.conal.net/pan/.

[4] S. Finne. Haskelldirect. URL: http://haskell.org/hdirect/.

[5] R. Hinze et al. Generic haskell. URL: http://www.generic-haskell.org/.

[6] R. Lämmel and J. Visser. Strafunski. URL: http://www.cs.vu.nl/Strafunski/.

[7] J. Meacham. Drift. URL: http://repetae.net/john/computer/haskell/DrIFT/.

[8] J. Meacham. printf-a-like in haskell 98. URL: http://repetae.net/john/computer/haskell/Format.hs.

[9] A. Reid, S. Finne, and T. Nordin. Green card. URL: http://haskell.org/greencard/.

[10] T. Sheard and S. P. Jones. Template metaprogramming for haskell. May 2002. URL: http://research.microsoft.com/Users/simonpj/papers/meta-haskell/.