Unrolling and Simplifying Expressions with Template Haskell

Ian Lynagh <igloo@earth.li>

Date December 14, 2002

Abstract

In this document we define a handful of modules for manipulating Template Haskell datastructures. Some merely provide utility functions which are used by other modules, and of course can be built upon by later or third-party modules. Others actually perform complete transformations, in particular recursion unrolling and expression simplification. Applied in the right place these can give significant performance increases, as we show with an example at the end.

Contents

1 Introduction **2**

2 Utils **2**
- 2.1 Motivation and Interface . . . 2
- 2.2 Sets . . . 3
- 2.3 Finding variables bound by patterns and declarations: `defines` . 3
- 2.4 Getting new variable names: `new_var` . . . 4

3 Substitutions **8**
- 3.1 Motivation and Interface . . . 8
- 3.2 Performing substitutions . . . 8
- 3.3 Expressions . . . 8
- 3.4 Declarations . . . 9
- 3.5 Righthand-sides . . . 9
- 3.6 Arithmetic Sequences . . . 10
- 3.7 Statements . . . 10
- 3.8 Matches and Clauses . . . 10

4 Simplifications **11**
- 4.1 Motivation and Interface . . . 11
- 4.2 Performing simplifications . . . 11
- 4.3 Expressions . . . 12
- 4.4 Declarations . . . 14
- 4.5 Right-Hand Sides . . . 15
- 4.6 Arithmetic Sequences . . . 15
- 4.7 Statements . . . 15
- 4.8 Matches and Clauses . . . 15

5 Unroll **15**
5.1 Motivation and Interface 15
5.2 What are we doing? 16
5.3 Some handy types 16
5.4 Unrolling recursions 16
5.5 Recursion 20
5.5.1 Expressions 20
5.5.2 Declarations 21
5.5.3 Righthand-sides 22
5.5.4 Arithmetic Sequences 22
5.5.5 Statements 22
5.5.6 Matches and Clauses 23

6 Conclusion **23**

1 Introduction

Template Haskell extends a plain old Haskell 98 system, giving the user power previously undreamt of. Suddenly it is possible, within some restrictions imposed by pragmatics, to convert from code to datastructures and back again. However, raw power can be hard to handle, so here we raise the level of abstraction a notch or two, providing modules which export functions to perform reasonable complex tasks on these datastructures and, in some cases, complete transformations useful in their own right.

The obvious uses for Template Haskell are code generation and code optimisation; we focus here on the latter, in particular investigating expression simplification and recursion unrolling. The other functionality we provide is motivated by what we need to implement these transformation functions.

The early modules define these utility functions, starting with a function to find the name of a variable that isn't free in an expression and a function to find all the variables bound by a pattern or declaration group in Section 2. Next we have a module which makes the substitution of literal values for variables easy in Section 3, building up to the actual simplification and unrolling functions in Sections 4 and 5. Finally, we finish with a reflection on what we have accomplished in Section 6.

2 Utils

2.1 Motivation and Interface

Before we start doing any real work with Template Haskell, as with most complex systems, it will be handy to have a few utilities available to us. In this module we export a pair of such functions that, as we will say, will be useful in other modules.

The first function takes either a pattern, a declaration or a list one of these and returns the list of variables the argument binds. The second takes an expression and gives the name of a variable that is not free in that expression. Both of these will be discussed further where they are defined.

We also export the names of a pair of classes we define and the `Set` type so that other modules may use them in type contexts and types.

```
module Utils (defines, new_var, Defines, FreeVars, Set) where

import List ((\\), union, delete)
import Maybe (fromMaybe)
import Language.Haskell.THSyntax
```

2.2 Sets

Although for convenience we will be using lists a lot in this module so we can use the prelude functions on them, most of the time we will actually be treating these lists as sets. This means that we are guaranteeing that if there are two elements x and y in the underlying list at different positions then $x \neq y$. We will only have sets of a type T if there is an instance of `Eq` on T. To make it clear when we are returning a set as opposed to a list we make a type synonym so we can reflect it in the types.

```
type Set a = [a]
```

We will often want to do the equivalent of `concatMap` on sets. However, `concatMap` doesn't maintain the invariant that elements are not repeated, so we define an analogous `unionMap` which does the same thing but preserves the property that each element only appears once in the result set (assuming the invariant is guaranteed to hold for the result of the function argument).

```
unionMap :: Eq b => (a -> Set b) -> [a] -> Set b
unionMap f = foldr union [] . map f
```

2.3 Finding variables bound by patterns and declarations: `defines`

As `defines` needs to operate on multiple types and it is clear that it can't be usefully defined as a parametrically polymorphic function it follows that we will need to define it as an ad-hoc polymorphic function. As such, we first define a class `Defines` with a single method, that which we export, with the obvious type.

```
class Defines a where
    defines :: a -> Set String
```

As well as the actualy datatypes that we will cover shortly we also need to provide instances for lists of these datatypes. In general, to find out what a list of things defines we just take the union of what each one defines; naturally this means that we require an instance for what we have a list of.

Although an instance for `statement` isn't necessary it is worth noting that this would be sufficient as, despite the left-to-right dependence of statements, the set of variables bound is still the union of those bound by each statement.

```
instance Defines a => Defines [a] where
    defines = unionMap defines
```

The actual instances are also straightforward. For the `Pat` instance, a single pattern variable binds that variable, a tuple binds the union of what each constituent pattern binds and so forth. A couple of constructors, `Plit` and `Pwild`, cause nothing to be bound so are caught with a final case that returns the empty list.

```
instance Defines Pat where
    defines (Pvar v) = [v]
    defines (Ptup ps) = unionMap defines ps
    defines (Pcon _ ps) = unionMap defines ps
    defines (Ptilde p) = defines p
    defines (Paspat v p) = [v] `union` defines p
    defines _ = []
```

The `Dec` instance has a couple of minor niggles. The function and foreign import cases are straightforward, simply binding the function name they define. Similarly the `Val` case just defines whatever its pattern defines.

It is slightly less clear which of `Class` and `Instance` should be considered as binding what their declarations bind. We take the view that it is only the class declaration that is binding variables, which is pretty much what happens in Haskell.

Although the presence of a type signature for a variable means that the variable must be defined in the same declaration group, we take the view that it isn't bound as another declaration must be responsible for the actual type signature.

Finally, all other declarations are caught by the final case, defining no variables.

```
instance Defines Dec where
    defines (Fun f _) = [f]
    defines (Val p _ _) = defines p
    defines (Class _ _ _ ds) = defines ds
    defines (Foreign (Import _ _ _ f _)) = [f]
    defines _ = []
```

This completes all the code needed for `defines`, our first export.

2.4 Getting new variable names: `new_var`

The definition of `new_var` itself is deceptively simple. We take the set of all variable identifiers, remove all the identifiers which appear as free variables in the expression we are given and return the first one. The complexity is hidden in the two auxiliary functions we are about to define.

```
new_var :: Exp -> String
new_var e = head (all_varids \\ free_vars e)
```

Let us consider `all_varids`, the first of these. It is not actually necessary for this to cover all possible varids, which in Template Haskell is all possible strings which would read to illegible datastructures while debugging; it suffices to pick a set of "sensible" characters and instead return the set of all strings that can be built from them. In order to not get confused while debugging it would make sense if these all looked liked valid Haskell varids, so we choose to use the set of lower case letters.

```
all_varids :: Set String
all_varids = multiperms ['a'..'z']
```

This leaves `multiperms` to actually generate the set of strings. To be precise, `multiperms` takes a set of values and returns the set of all non-empty lists of values from this set, where each value can appear multiple times or not at all in any given list.

To do this we first define a function `multiperms'` that returns the set of all such lists rather than just the non-empty ones. We will attempt to show that this first produces all lists of length 0 and then use the inductive hypothesis that, after producing all lists of length i, it produces all lists of length $i+1$.

The first element of this set is the empty list, and this is the only list of length 0, so our first proof obligation is satisfied.

By the inductive hypothesis, and as each element is one longer than the element of `multiperms'` used, after the last list of length i is emitted the next n^i, assuming xs has `xs` elements, elements of `multiperms'` are the lists of length i. For each of these lists we output the list with each element of `xs` prepended, i.e. all the lists of length $i+1$.

Therefore, by mathematical induction, `multiperms'` fullfils its specification. The result of `multiperms` is the same but with the first element, the empty list, removed.

```
multiperms :: Set a -> Set [a]
multiperms = tail . multiperms'
    where multiperms' :: Set a -> Set [a]
          multiperms' xs = []:[ (x:ys) | ys <- multiperms' xs, x <- xs ]
```

One thing remains before `new_var` is complete, and that is `free_vars`. This needs to take an expression and return the set of free variables used in that expression. However, there are various other datatypes that we will need to support this operation on in order to define it for expressions; we therefore make it a class. In general for a list of things the free variables of the list will be the union of the free variables in each individual thing, but as we will see this is not always the case. We therefore make a function to deal with lists of elements, similar to how lists are handled by the prelude function `show`.

```
class FreeVars a where
    free_vars :: a -> Set String
    free_vars_list :: [a] -> Set String
    free_vars_list = unionMap free_vars
```

In order to get the free variables in a list of things we use `free_vars_list`, so the instance for lists of things which have a `FreeVars` instance just calls `free_vars_list` from `free_vars`. We will never have a list of lists, but the default definition seems reasonable at any rate.

```
instance FreeVars a => FreeVars [a] where
    free_vars = free_vars_list
```

The instance for expressions is largely straightforward. Variables and infix expressions are where the variables are used and most other constructors wrap up expressions that could contain more free variables. The only notable case is `Let` where we need to remove the set of variables defined by the declarations from those otherwise free in the expression, which we do using `defines` we wrote earlier. The final case catches expressions which can have no free variables in such as literals.

```
instance FreeVars Exp where
    free_vars (Var v) = [v]
    free_vars (App e1 e2) = free_vars e1 `union` free_vars e2
    free_vars (Infix e1 o e2) = [o] `union` v1 `union` v2
        where v1 = fromMaybe [] $ fmap free_vars e1
              v2 = fromMaybe [] $ fmap free_vars e2
    free_vars (Lam ps e) = free_vars e \\ defines ps
    free_vars (Tup es) = free_vars es
    free_vars (Cond e1 e2 e3) = free_vars [e1, e2, e3]
    free_vars (Let ds e) = free_vars ds `union` (free_vars e \\ defines ds)
    free_vars (Case e ms) = free_vars e `union` free_vars ms
    free_vars (Do ss) = free_vars ss
    free_vars (Comp ss) = free_vars ss
    free_vars (ArithSeq dd) = free_vars dd
    free_vars (ListExp es) = free_vars es
    free_vars (SigExp e _) = free_vars e
    free_vars _ = []
```

The `Dec` instance is also straightforward, although we need to be careful that not only do we delete what we bind from the set when we call `free_vars` individually, but that we also delete all the bound variables from all the declarations when `free_vars_list` is called in a declaration group.

```
instance FreeVars Dec where
    free_vars (Fun f cs) = delete f (free_vars cs)
    free_vars (Val p rhs ds) = (in_rhs `union` in_ds) \\ redefed
        where in_rhs = free_vars rhs
              in_ds = free_vars ds
              redefed = defines p `union` defines ds
    free_vars (Class _ _ _ ds) = free_vars ds
    free_vars (Instance _ _ ds) = free_vars ds
    free_vars _ = []
    free_vars_list ds = free_vars ds \\ defines ds
```

Both `RightHandSide` and `DotDot` are extremely trivial, and the default definition for lists suffices.

```
instance FreeVars e => FreeVars (RightHandSide e) where
    free_vars (Normal e) = free_vars e
    free_vars (Guarded ges)
     = case unzip ges of
           (gs, es) -> free_vars gs `union` free_vars es
```

```
instance FreeVars e => FreeVars (DotDot e) where
    free_vars (From e) = free_vars e
    free_vars (FromThen e1 e2) = free_vars [e1, e2]
    free_vars (FromTo e1 e2) = free_vars [e1, e2]
    free_vars (FromThenTo e1 e2 e3) = free_vars [e1, e2, e3]
```

The instance for `Statement` is the most interesting one. Let use consider the case of a list of statements first. In contrast to `Defines` the list case needs different handling to other lists we've seen—if a particular element binds a variable v then that variable is bound in that statement and all statements to the right, but not the statements to the left. We can easily code this as a `foldr` where for each statement we take the union of the newly used free variables and the accumulated free variables, initially the empty list, and then remove any variables bound by the declaration. The non-list version then becomes a simple call to the folded function.

```
instance (Defines p, Defines d, FreeVars e, FreeVars d)
       => FreeVars (Statement p e d) where
    free_vars = flip do_stmt []
    free_vars_list = foldr do_stmt []

do_stmt :: (Defines p, Defines d, FreeVars e, FreeVars d)
        => Statement p e d -> Set String -> Set String
do_stmt (BindSt p e) fvs = (free_vars e `union` fvs) \\ defines p
do_stmt (LetSt ds) fvs = (free_vars ds `union` fvs) \\ defines ds
do_stmt (NoBindSt e) fvs = free_vars e `union` fvs
do_stmt (ParSt _) _ = error "free_vars_stats: ParSt not handled"
```

There are two cases left to define, `Clause` and `Match`. The pair are virtually identical in both structure and meaning, so unsurprisingly the instances are almost identical. The definition is very simple.

```
instance (Defines p, Defines d, FreeVars e, FreeVars d)
       => FreeVars (Match p e d) where
    free_vars (Mat p rhs ds) = (in_rhs `union` in_ds) \\ redefed
        where in_rhs = free_vars rhs
              in_ds = free_vars ds
              redefed = defines p `union` defines ds

instance (Defines p, Defines d, FreeVars e, FreeVars d)
       => FreeVars (Clause p e d) where
    free_vars (Clause ps rhs ds) = (in_rhs `union` in_ds) \\ redefed
        where in_rhs = free_vars rhs
              in_ds = free_vars ds
              redefed = defines ps `union` defines ds
```

This completes all the instances we are interested in, so we are finished.

3 Substitutions

3.1 Motivation and Interface

When using Template Haskell to perform partial evaluation and other optimisations it is easy to imagine situations in which you will wish to replace all free occurences of a variable in some expression with a literal value which you have calculated[1]. While this is conceptually straightforward it is a reasonable amount of code, so we provide it in a module here to facilitate reuse.

A single function is exported which performs these substitutions, and we use the `defines` function defined in Section 2.3 to determine if the variable is free in the sub-structures.

```
module Subst (subst) where

import Language.Haskell.THSyntax
import Utils (defines, Defines)
```

3.2 Performing substitutions

Now we come to define how the actual substitutions are done. Again we use a class, with the method `subst`, to allow us to work over several types. In most cases a list of things can be handled by mapping the substitution function over the list, but the left-to-right variable shadowing of statements means we have to allow for a special case. To do this we use a second method `subst_list` to operate on lists of things in the same way that `show` works.

```
class Subst a where
    subst :: String -> Lit -> a -> a
    subst_list :: String -> Lit -> [a] -> [a]
    subst_list v l = map (subst v l)
```

If we are asked to substitute in a list of things we call the `subst_list` method for the type of a single thing. The default instance is sufficient for lists of lists.

```
instance Subst a => Subst [a] where
    subst = subst_list
```

3.3 Expressions

Perhaps the most obvious type you would wish to substitute into is `Exp`. It is also distinguished by being the only type in which actual substitution is done directly. The rest of the definition is mainly straightforward recursion through to datastructures that are or might contain expressions, using `defines` in the `Let` case to check that the variable hasn't been shadowed before we recurse.

```
instance Subst Exp where
    subst v l (Var x) | x == v = (Lit l)
    subst v l (App e1 e2) = App (subst v l e1) (subst v l e2)
```

[1] In general you might prefer to be able to substitute any expression in, but the added complexities of this are not considered here.

```
    subst v l (Infix me1 s me2) = (Infix (fmap (subst v l) me1)
                                         s
                                         (fmap (subst v l) me2))
    subst v l (Lam ps e)
     | v `elem` defines ps = Lam ps e
     | otherwise           = Lam ps (subst v l e)
    subst v l (Tup es) = Tup (map (subst v l) es)
    subst v l (Cond g t f) = Cond (subst v l g)
                                  (subst v l t)
                                  (subst v l f)
    subst v l (Let ds e)
     | v `elem` defines ds = Let ds e
     | otherwise           = Let (subst v l ds) (subst v l e)
    subst v l (Case e ms) = Case (subst v l e) (map (subst v l) ms)
    subst v l (Do ss) = Do (subst v l ss)
    subst v l (Comp ss) = Comp (subst v l ss)
    subst v l (ArithSeq dd) = ArithSeq (subst v l dd)
    subst v l (ListExp es) = ListExp (map (subst v l) es)
    subst v l (SigExp e t) = SigExp (subst v l e) t
    subst _ _ e = e
```

3.4 Declarations

Declarations are also reasonably straightforward. If we are asked to do a substitution on a single declaration and the declaration defines what we are substituting for then the declaration is unchanged. Otherwise, in the cases of functions, values, classes and instances (those that can contain expressions) we recursively call substitute on any bit that can contain an expression. Any other declaration is unchanged.

If we have a list of declarations then we first check to see if any of them rebinds what we are substituting for; if so then the entire declaration group is unchanged. Otherwise we map the substitution over everything in the group.

```
instance Subst Dec where
    subst v _ d | v `elem` defines d = d
    subst v l (Fun f cs) = Fun f (subst v l cs)
    subst v l (Val p rhs ds) = Val p (subst v l rhs) (subst v l ds)
    subst v l (Class cxt tc tvars ds) = Class cxt tc tvars (subst v l ds)
    subst v l (Instance cxt typ ds) = Instance cxt typ (subst v l ds)
    subst _ _ d = d
    subst_list v l ds
     | v `elem` defines ds = ds
     | otherwise           = map (subst v l) ds
```

3.5 Righthand-sides

The instance for right-hand sides is exceptionally simple, just recursing through the expressions. Lists of right-hand sides have no special meaning so the default method is fine.

```
instance Subst e => Subst (RightHandSide e) where
```

```
    subst v l (Guarded ges)
     = Guarded (map (\(e1, e2) -> (subst v l e1, subst v l e2)) ges)
    subst v l (Normal e) = Normal (subst v l e)
```

3.6 Arithmetic Sequences

The arithmetic sequences instance is equally simple, featuring simple recursion through to the expressions.

```
instance Subst e => Subst (DotDot e) where
    subst v l (From e) = From (subst v l e)
    subst v l (FromThen e1 e2) = FromThen (subst v l e1) (subst v l e2)
    subst v l (FromTo e1 e2) = FromTo (subst v l e1) (subst v l e2)
    subst v l (FromThenTo e1 e2 e3)
     = FromThenTo (subst v l e1) (subst v l e2) (subst v l e3)
```

3.7 Statements

The other complicated case is that of statments. If we have a list of statements and the first statement rebinds what we are substituting for then we return the list unchanged. Otherwise we recursively substitute on the relevant parts of the first statement to make the head of the result and substitute on the remaining statements to make the tail.

For the single element case we use the slightly bizarre method of wrapping the element up to form a singleton list, calling `subst` on the list and taking the head.

```
instance (Defines d, Defines p, Subst d, Subst e)
      => Subst (Statement p e d) where
    subst v l s = head $ subst v l [s]
    subst_list _ _ [] = []
    subst_list v l ss@(BindSt p e:ss')
     | v `elem` defines p = ss
     | otherwise          = BindSt p (subst v l e):subst v l ss'
    subst_list v l ss@(LetSt ds:ss')
     | v `elem` defines ds = ss
     | otherwise           = LetSt (subst v l ds):subst v l ss'
    subst_list v l (NoBindSt e:ss) = NoBindSt (subst v l e):subst v l ss
    subst_list _ _ (ParSt _:_) = error "subst_list[Statement]: ParSt"
```

3.8 Matches and Clauses

Finally we have the cases for matches and clauses, again virtually identical. If either the pattern(s) or declarations bind what we are looking for then the match or clause is unchanged. Otherwise we recursively substitute on the parts that may contain an expression.

```
instance (Defines d, Defines p, Subst d, Subst e) => Subst (Match p e d) where
    subst v l m@(Mat p rhs ds)
     | v `elem` defines ds = m
     | v `elem` defines p  = m
     | otherwise           = Mat p (subst v l rhs) (subst v l ds)
```

```
instance (Defines d, Defines p, Subst d, Subst e) => Subst (Clause p e d) where
    subst v l c@(Clause ps rhs ds)
     | v `elem` defines ds = c
     | v `elem` defines ps = c
     | otherwise           = Clause ps (subst v l rhs) (subst v l ds)
```

4 Simplifications

4.1 Motivation and Interface

So far we have built up a lot of tools for working with Template Haskell datastructures but haven't actually produced any function that themselves are capable of doing useful work on a datastructure. In this module we export a function `simplify` which, when applied to an expression[2], returns a simplified, but functionally equivalent, expression. This can be spliced in place of the original, potentially giving a speed-up (and, perhaps more importantly, never having a detrimental effect on either time or space usage). We also export the `Simplify` class so that it can be used in contexts in other modules.

```
module Simplify (simplify, Simplify) where

import Language.Haskell.THSyntax
import Monad (liftM)
import Maybe (fromJust, isJust)
import Utils (new_var)
```

4.2 Performing simplifications

As with the previouly defined operations, we will want to be able to simplify a number of the Template Haskell datatypes. As before we use a class to be able to do this. The simplifications that we require can be done with a very simple interface. We don't care about details like how variables bound affect an entire declaration group but only everything to the right in a list of statements, so there is no need to do anything special for lists. We therefore need only a single method.

```
class Simplify a where
    simplify :: a -> a
```

Also, it is likely that we will wish to simplify quasi-quoted code and then just splice the result back in, like `$( simplify [| 1 + 2 |] )`. We therefore make an instance for things in the `Q` monad, just lifting `simplify` into it.

```
instance Simplify a => Simplify (Q a) where
    simplify = liftM simplify
```

If we are given a list of things to simplify we just simplify each one in the list independently.

```
instance Simplify a => Simplify [a] where
    simplify = map simplify
```

[2]It can actually be used on anything with a `Simplify` instance.

4.3 Expressions

Most of the work is done by the expression instance. It takes an `Exp` and returns a simplified, functionally equivalent expression. In general this just requires recursively simplifying on the arguments, as is done in the case of `App`, but there are +cases where actual simplification takes place; for example, the variable +`GHC.Base:otherwise` can be replaced with the constructor +`GHC.Base:True` (which will in turn allow further +optimisations to be performed without having to check for the +`GHC.Base:otherwise` case elsewhere).

```
instance Simplify Exp where
    simplify (Var "GHC.Base:otherwise") = Con "GHC.Base:True"
    simplify (App e1 e2) = App (simplify e1) (simplify e2)
```

The interesting case is that representing infix expressions. Here, if we know the values of at least one of the operands it can be possible to simplify the expression. We start off by recursively simplifying the left and right operands and then pattern match on them along with the operator.

```
    simplify (Infix me1 op me2)
     = let (me1', me2') = (fmap simplify me1, fmap simplify me2)
       in case (me1', op, me2') of
```

AS we continue with the case analysis, note that we are making the assumption that the operands are compared in the same way as `Integer`s, which may be invalid! Even `Int` arithmetic may go wrong due to underflow! However, it is infeasible with the curernt Template Haskell system to do anything else.

If we are asked to do a comparison on the two operands and we have managed to simplify them both to literals then we can do the comparison at compile time leaving just the boolean value.

```
              (Just (Lit (Integer i1)), "GHC.Base:>", Just (Lit (Integer i2)))
                  -> if i1 > i2 then Con "GHC.Base:True"
                                else Con "GHC.Base:False"
              (Just (Lit (Integer i1)), "GHC.Base:>=", Just (Lit (Integer i2)))
                  -> if i1 >= i2 then Con "GHC.Base:True"
                                 else Con "GHC.Base:False"
              (Just (Lit (Integer i1)), "GHC.Base:<", Just (Lit (Integer i2)))
                  -> if i1 < i2 then Con "GHC.Base:True"
                                else Con "GHC.Base:False"
              (Just (Lit (Integer i1)), "GHC.Base:<=", Just (Lit (Integer i2)))
                  -> if i1 <= i2 then Con "GHC.Base:True"
                                 else Con "GHC.Base:False"
```

Simple arithmetic operations can be inlined under the same conditions. We can also simplify the units of the operators out.

```
              (Just (Lit (Integer 0)), "GHC.Num:+", Just x) -> x
              (Just x, "GHC.Num:+", Just (Lit (Integer 0))) -> x
              (Just (Lit (Integer i1)), "GHC.Num:+", Just (Lit (Integer i2)))
                  -> Lit (Integer (i1 + i2))
              (Just (Lit (Integer 0)), "GHC.Num:-", Just x) -> x
```

```
(Just x, "GHC.Num:-", Just (Lit (Integer 0))) -> x
(Just (Lit (Integer i1)), "GHC.Num:-", Just (Lit (Integer i2)))
    -> Lit (Integer (i1 - i2))
(Just (Lit (Integer 1)), "GHC.Num:*", Just x) -> x
(Just x, "GHC.Num:*", Just (Lit (Integer 1))) -> x
(Just (Lit (Integer i1)), "GHC.Num:*", Just (Lit (Integer i2)))
    -> Lit (Integer (i1 * i2))
```

The case for raising a number to a power is a bit more complex. We first check that it is actually being raised to a literal integral power[3] and that this power is greater than zero[4]. We then call `expand_exp` to build the multiplications up before finally calling `simplify` recursively on the result as we may be able to further simplify the multiplications.

```
(Just base, "GHC.Real:^", Just lit_power)
    | isJust maybe_power && power > 0
          -> simplify $ expand_exp base power
  where maybe_power = get_lit lit_power
        power = fromJust maybe_power
        get_lit :: Exp -> Maybe Integer
        get_lit (Lit (Integer i)) = Just i
        get_lit (SigExp (Lit (Integer i))
                        (Tcon (TconName "GHC.Base:Int")))
          = Just (fromIntegral (fromInteger i :: Int))
        get_lit (SigExp (Lit (Integer i))
                        (Tcon (TconName "GHC.Base:Integer")))
          = Just i
        get_lit _ = Nothing
```

The definition of `expand_exp` is heavily based on the prelude definition of ^. This is by necessity, though—it must perform exactly the same multiplications if it is to produce the same result as $*$ may not be associative.

```
expand_exp :: Exp -> Integer -> Exp
expand_exp e i | i > 0 = f e (i-1) e
 where
  f _ 0 y = y
  f x n y = g x n
   where
    g z m
     | even m = g (build_square z) (m `quot` 2)
     | otherwise = let y' = Infix (Just z) "GHC.Num:*" (Just y)
                   in f z (m-1) y'
expand_exp _ _ = error "expand_exp: bad args"
```

To actually square expression, in general we build a `Let` to define a new variable as what is being squared and then return this variable multiplied by itself. However, when the expression is already a variable or literal there is no

[3] Although it looks like we are able to check for a suitable type here, type defaulting of a non-explicitly typed power means that it may not be.

[4] Substituting in 1 if the power is 0 may alter which types are inferred.

recomputation to be saved by doing this so we just multiply what we are given by itself.

```
build_square :: Exp -> Exp
build_square e@(Var _) = Infix (Just e) "GHC.Num:*" (Just e)
build_square e@(Lit _) = Infix (Just e) "GHC.Num:*" (Just e)
build_square e = let v = new_var e
                     arg = Just (Var v)
                 in Let [Val (Pvar v) (Normal e) []]
                        (Infix arg "GHC.Num:*" arg)
```

We don't do anything with any other infix expressions, so we just put them back together with the simplified operands.

```
(_, _, _) -> Infix me1' op me2'
```

Similarly if we can work out that the guard in conditional statements is either `True` or `False` then we just return the relevant expression simplified, otherwise just simplify all the parts and put the conditional back together again.

```
    simplify (Cond g t f) = case simplify g of
                                Con "GHC.Base:True" -> simplify t
                                Con "GHC.Base:False" -> simplify f
                                g' -> Cond g' (simplify t) (simplify f)
```

For the other cases we just recursively simplify or, if there is nothing to be simplified, return the expression unchanged. Note that there are a number of other datatypes we wish to simplify, so we must now provide the functions needed to do these.

```
    simplify (Lam ps e) = Lam ps (simplify e)
    simplify (Tup es) = Tup (map simplify es)
    -- Should subst literals/vars?
    simplify (Let ds e) = Let (simplify ds) (simplify e)
    simplify (Case e ms) = Case (simplify e) (map simplify ms)
    simplify (Do ss) = Do (map simplify ss)
    simplify (Comp ss) = Comp (map simplify ss)
    simplify (ArithSeq dd) = ArithSeq (simplify dd)
    simplify (ListExp es) = Tup (map simplify es)
    simplify (SigExp e t) = SigExp (simplify e) t
    simplify e = e
```

4.4 Declarations

The declarations instance is unsurprisingly boring, simply recursively simplifying the potential interesting parts of the declarations.

```
instance Simplify Dec where
    simplify (Fun f cs) = Fun f (map simplify cs)
    simplify (Val p rhs ds) = Val p (simplify rhs) (simplify ds)
    simplify (Class cxt tycon tyvars ds) = Class cxt tycon tyvars (simplify ds)
    simplify (Instance cxt typ ds) = Instance cxt typ (simplify ds)
    simplify d = d
```

4.5 Right-Hand Sides

Similarly right-hand sides just recursively simplify the interesting bits.

```
instance Simplify e => Simplify (RightHandSide e) where
    simplify (Normal e) = Normal (simplify e)
    simplify (Guarded ges)
     = Guarded (map (\(e1, e2) -> (simplify e1, simplify e2)) ges)
```

4.6 Arithmetic Sequences

Continuing the theme, arithmetic sequences also just recursively simplify the potentially interesting bits.

```
instance Simplify e => Simplify (DotDot e) where
    simplify (From e) = From (simplify e)
    simplify (FromThen e1 e2) = FromThen (simplify e1) (simplify e2)
    simplify (FromTo e1 e2) = FromTo (simplify e1) (simplify e2)
    simplify (FromThenTo e1 e2 e3)
     = FromThenTo (simplify e1) (simplify e2) (simplify e3)
```

4.7 Statements

As mentioned earlier, this class is quite surprising in that statements are not inteesting—yet another instance that simply recursively simplifies bits.

```
instance (Simplify e, Simplify d) => Simplify (Statement p e d) where
    simplify (BindSt p e) = BindSt p (simplify e)
    simplify (LetSt ds) = LetSt (simplify ds)
    simplify (NoBindSt e) = NoBindSt (simplify e)
    simplify (ParSt _) = error "simplify[Statement]: ParSt"
```

4.8 Matches and Clauses

Finally, it will come as no surprise that both matches and clauses just recursively simplify the relevant components.

```
instance (Simplify e, Simplify d) => Simplify (Match p e d) where
    simplify (Mat p rhs ds)
     = Mat p (simplify rhs) (simplify ds)

instance (Simplify e, Simplify d) => Simplify (Clause p e d) where
    simplify (Clause ps rhs ds)
     = Clause ps (simplify rhs) (simplify ds)
```

5 Unroll

5.1 Motivation and Interface

There are a number of algorithms in which the inner loop is executed a fixed number of times, generally implemented in Haskell as recursive functions. These functions fall roughly into two classes: those that recurse over a datastructure such as a list and those that use integers to guide the recursion. In this module we deal only with the second case, but within that try to provide as powerful a solution as possible.

We export a single function `unroll` which takes a declaration (of a function), along with some other information we discuss below, and returns a functionally equivalent unrolled declaration.

```
module Unroll (unroll) where

import Language.Haskell.THSyntax
import Subst (subst)
import Simplify (simplify)
import Utils (defines, Defines)
```

5.2 What are we doing?

Before we start writing code let us have it clear in our minds what interface we wish to expose to the user. Our basic objective is to allow them to give us a definition of a function and we will then return an equivalent, but unrolled, function. The intention is that the user will also indicate which argument (by number to solve problems with argument renaming), which must be a simple pattern variable, we wish to unroll with respect to, and also what value this argument will have when it is called by other functions. It is not compulsory to use this value, but the unrolled definition of the function will not be used so no benefit will be derived (unless a recursive call *does* use this value).

We also want to be able to allow the user to protect themselves against shooting themself in the foot by allowing them to optionally specify a maximum depth to unroll to. If there are still recursive calls once this depth is exceeded then they remain as recursive calls. However, if the user is confident that the unrolling will terminate then they can fly in the face of danger and not set any bound.

5.3 Some handy types

Let us take a minute to define a couple of types to allow us to give more descriptive types to the functions we write. First we have `Depth` which is either an `Int` (wrapped up in the `Just` constructor) representing the maximum depth we are willing to unroll to or `Nothing` meaning an arbitrary (in fact infinite!) amount of unrolling is permitted. To select which argument we wish to unroll with respect to we just use an `Int`. Little benefit would be derived by using abstract datatypes, and the code would be somewhat less transparent.

```
type Depth = Maybe Int
type Argument = Int
```

5.4 Unrolling recursions

The discussion thus far practically writes the type signature of the exported function `unroll` for us, but to make sure we agree on the order of the arguments let us spell it out explicitly. It first takes a `Depth`, the maximum unrolling depth, then an Argument so we know which argument we are unrolling with respect to. Next is the Lit giving us the initial value of this argument—it needs to be a Lit rather than an Exp so that we can also use it as a pattern, as we shall see shortly. Finally it takes a Dec to be unrolled and returns the new unrolled Dec.

Explaining the actual unrolling abstractly isn't likely to be very elucidating, so instead we will follow the unrolling of a fictional example. We will start with the function foo in Figure 1(a), unrolling it with respect to the variable y which we will initially call with value 0.

For now we handle only the special case where the declaration is a function with a single clause. We first check that there are enough patterns for the argument we are looking for to exist and return an error if not.

If there is then we return a function with two clauses. The first has the initial literal value substituted in for the appropriate argument pattern, our unrolled expression as a normal right-hand side and no where clause. The second is exactly the same as the clause we were given; this catches any calls that aren't made with the value for the argument we have used to unroll the function.

```
unroll :: Depth -> Argument -> Lit -> Dec -> Dec
-- Special case
unroll m i l (Fun f [c@(Clause ps rhs decs)])
 | length ps <= i = error "unroll: Not enough patterns"
 | otherwise      = Fun f [(Clause ps' (Normal e) []), c]
```

Creating the pattern list for the first clause is straightforward; We split the original pattern list after i arguments and drop the first one in the right-hand list, substituting in the literal we were given as a pattern.

```
  where ps' = case splitAt i ps of
                  (xs, ys) -> xs ++ [Plit l] ++ tail ys
```

For the unrolling, the first thing we do is to take the right-hand side of the function and convert it into an expression rhs_expr. In the case of "normal" right-hand sides this is trivial, but in the case of guarded right-hand sides we use a fold to build a conditional expression. If none of the guards match then an error occurs, but that is fine as the same would happen in the original function.

```
        rhs_exp = case rhs of
                      Normal expression -> expression
                      Guarded ges -> foldr (uncurry Cond) err ges
          where err = App (Var "error") (Lit (String "Generated be unroll"))
```

However, rhs_expr doesn't necessarily have all of the information from the function. As well as the pattern list, which we have already dealt with, the function could have a where clause with a list of declarations. Fortunately, we can incorporate these into an expression with exactly the same semantics within the restrictions we have imposed by building a let expression which binds the declarations and then evaluates the expression within that environment. This gives us rhs_expr' which equates to Figure 1(b).

There are two things worth noting here: first, if we used rhs_expr' as a normal right-hand side with the original patterns and an empty list of declarations it would be functionally equivalent to the original; second, if we create

```
foo x y z | y > 15    = 1 + 2 + x + y + z
          | otherwise = x + y + foo x (y + 10) z
```

(a) Original function

```
if y > 15 then 1 + 2 + x + y + z
          else if otherwise then x + y + foo x (y + 10) z
                            else error "Generated be unroll"
```

(b) Convert right-hand side to expression

```
if y > 15 then 3 + x + y + z else x + y + foo x (y + 10) z
```

(c) Simplify

```
if ☼ > 15                                \x y z -> if ☼ > 15
then 3 + x + ☼ + z                                 then 3 + x + ☼ + z
else x + ☼ + foo x (☼ + 10) z                      else x + ☼ + foo x (☼ + 10) z
```

(d) Make substitution functions (SFs)

```
if 0 > 15
then 3 + x + 0 + z
else x + 0 + foo x (0 + 10) z
```

(e) Take the left SF and put 0 in the hole

```
x + foo x 10 z
```

(f) Simplify

```
x + (\x y z -> if ☼ > 15
               then 3 + x + ☼ + z
               else x + ☼ + foo x (☼ + 10) z) x 10 z
```

(g) Substitute the right SF in for `foo`...

```
x + (\x y z -> if 10 > 15
               then 3 + x + 10 + z
               else x + 10 + foo x (10 + 10) z) x 10 z
```

(h) ...putting 10 in the hole

```
x + (\x y z -> x + 10 + foo x 20 z) x 10 z
```

(i) Simplify

```
x + (\x y z -> x + 10 + (\x y z -> if ☼ > 15
                                   then 3 + x + ☼ + z
                                   else x + ☼ + foo x (☼ + 10) z) x 20 z) x 10 z
```

(j) Substitute the right SF in for `foo`...

```
x + (\x y z -> x + 10 + (\x y z -> if 20 > 15
                                   then 3 + x + 20 + z
                                   else x + 20 + foo x (20 + 10) z) x 20 z) x 10 z
```

(k) ...putting 20 in the hole

```
x + (\x y z -> x + 10 + (\x y z -> 3 + x + 20 + z) x 20 z) x 10 z
```

(l) Simplify

```
foo x 0 z = x + (\x y z -> x + 10 + (\x y z -> 3 + x + 20 + z) x 20 z) x 10 z
```

(m) Convert back to a declaration

Figure 1: Step-by-step unrolling

the lambda abstraction that binds the variables in the patterns of the function definition and then returns `rhs_expr'` then this expression can be substituted in place of any recursive calls to the function.

To save us repeating some work later on we take this opportunity to simplify the expression; at the very least, when guards are used, it is likely that the generated error can be pruned, as is the case in Figure 1(c).

```
rhs_exp' = simplify $ Let decs rhs_exp
```

The next thing we do is to create two "substitution functions". The first of these takes a literal and returns the simplified expression corresponding to the function right-hand side, i.e. Figure 1(c) in our example, with the literal substituted for all occurences of the variable we are unrolling with respect to, i.e. `y` in our example. The second is the same but for the right-hand side encapsulated in the lambda abstraction noted earlier. These are shown in Figure 1(d) with ☼ marking where the literal is substituted. The first of these functions is what we will use to start building the unrolled right-hand side while the second is what we will use to replace the recursive calls; this will become clear when we look at the definition of `unroll'` shortly.

We don't handle the cases were the pattern we are interested in is not a pattern variable; if such a case arises we just return an error.

```
sub_rhs_expr = case ps !! i of
                   Pvar v -> \literal -> subst v literal rhs_exp'
                   _ -> error "unroll: ps !! i not a Pvar"
sub_rhs_lambda = \v -> Lam ps (sub_rhs_expr v)
```

The rest of the unrolling work is carried out by `unroll'`. We pass it our first three arguments along with the name of the function being unrolled and the substitution functions. We will see shortly that this is sufficient for what it needs to do.

```
e = unroll' m i l f sub_rhs_lambda sub_rhs_expr
```

Any declarations we can't handle yet are just passed back unmodified.

```
unroll _ _ _ d = d
```

The first three arguments to `unroll'` are the maximum unrolling depth, the argument number to be replaced and the literal to replace it with; these are the same as the first three arguments to `unroll`. The fourth is name of the function we are unrolling. Then we have two arguments that takes a `Lit` and return an `Exp`; the first of these is the function that is used to create expressions representing recursive calls and the second is to build the expression to work on. Depending on what iteration you are looking at the second will either be the left SF in Figure 1(d) or the inner part (identical to the right SF in Figure 1(d)) of Figure 1(g) or 1(j) in the example; the first will always be the right SF in Figure 1(d).

The first thing we do is to substitute the literal we are passed into the second SF to give the expression we are working on. This progresses us onto Figures 1(e), 1(h) and 1(k) respectively.

Next we simplify the resulting expression. This tends to select the correct branches of conditionals, necessary if the unrolling is to terminate of its own accord, and flattens the value passed as the appropriate argument to recursive calls down to a literal, necessary if we are to be able to continue unrolling. In the example this progresses us to Figures 1(f), 1(i) and 1(l). In all three cases the appropriate branch of the conditional has been chosen, and in the first two cases, the additions have been simplified.

```
unroll' :: Depth -> Argument -> Lit -> String -> (Lit -> Exp)
                                                -> (Lit -> Exp) -> Exp
unroll' (Just 0) _ l _ _ e = e l
unroll' m i l f new_f e = e''
    where e' = simplify (e l)
          e'' = recurse (fmap (\x -> x - 1) m) i f new_f e'
```

5.5 Recursion

Finally we need to actually do the recursive unrollings. The idea here is similar to that for substitutions, so the same pattern of code is apparent. We start with a class with methods for both single things and liss of things.

```
class Recurse a where
    recurse :: Depth -> Argument -> String -> (Lit -> Exp) -> a -> a
    recurse_list :: Depth -> Argument -> String -> (Lit -> Exp) -> [a] -> [a]
```

Then we can very easily write a default list instance as with substitutions.

```
instance Recurse a => Recurse [a] where
    recurse = recurse_list
```

5.5.1 Expressions

The hard work is all done by the expression class, in particular in the in the function application case. We first work our way down the left spine of the applications, producing a list of expressions applied to each other. Then, if the first expression is the variable we are interested in and we have a literal in the appropriate argument, we recursively call `unroll'` on it (and call `recurse` on all the arguments). Otherwise we just call `recurse` on all the expressions. Finally we use a fold to put the applications back together again.

```
instance Recurse Exp where
    recurse m i f new_f a@(App _ _) = foldl1 App app_list
        where app_list = case unwrap_apps a [] of
                          (Var v:xs)
                              | v == f && length xs > i && is_lit
                                  -> let this = unroll' m i l f new_f new_f
                                             rest = map (recurse m i f new_f) xs
                                         in this:rest
                            where (is_lit, l) = case xs !! i of
                                                    Lit x -> (True, x)
                                                    _ -> (False, undefined)
```

```
                    es -> map (recurse m i f new_f) es
        unwrap_apps :: Exp -> [Exp] -> [Exp]
        unwrap_apps (App e1 e2) es = unwrap_apps e1 (e2:es)
        unwrap_apps e es = e:es
```

The other cases just involve the recursion and rebinding checks we are used to seeing.

```
    recurse m i f new_f (Infix me1 s me2)
        = Infix (fmap (recurse m i f new_f) me1)
                s
                (fmap (recurse m i f new_f) me2)
    recurse m i f new_f (Lam ps e)
     | f `elem` defines ps = Lam ps e
     | otherwise           = Lam ps (recurse m i f new_f e)
    recurse m i f new_f (Tup es) = Tup (map (recurse m i f new_f) es)
    recurse m i f new_f (Cond guard true false)
        = Cond (recurse m i f new_f guard)
               (recurse m i f new_f true)
               (recurse m i f new_f false)
    recurse m i f new_f (Let ds e)
     | f `elem` defines ds = Let ds e
     | otherwise           = Let (recurse m i f new_f ds)
                                 (recurse m i f new_f e)
    recurse m i f new_f (Case e ms) = Case (recurse m i f new_f e)
                                           (recurse m i f new_f ms)
    recurse m i f new_f (Do ss) = Do (recurse m i f new_f ss)
    recurse m i f new_f (Comp ss) = Comp (recurse m i f new_f ss)
    recurse m i f new_f (ArithSeq dd) = ArithSeq (recurse m i f new_f dd)
    recurse m i f new_f (ListExp es) = ListExp (recurse m i f new_f es)
    recurse m i f new_f (SigExp e t) = SigExp (recurse m i f new_f e) t
    recurse _ _ _ _ e = e
```

5.5.2 Declarations

Declarations are also reasonably straightforward. If we are asked to recurse on a single declaration and the declaration defines what we are looking for then the declaration is unchanged. Otherwise, in the cases of functions, values, classes and instances (those that can contain expressions) we recurse on any bit that can contain an expression. Any other declaration is unchanged.

If we have a list of declarations then we first check to see if any of them rebinds what we are looking for; if so then the entire declaration group is unchanged. Otherwise we map the recursion over everything in the group.

```
instance Recurse Dec where
    recurse m _ f _ d | f `elem` defines d = d
    recurse m i f new_f (Fun fun cs) = Fun f (recurse m i fun new_f cs)
    recurse m i f new_f (Val p rhs ds) = Val p (recurse m i f new_f rhs)
                                               (recurse m i f new_f ds)
    recurse m i f new_f (Class cxt tc tvars ds)
        = Class cxt tc tvars (recurse m i f new_f ds)
```

```
    recurse m i f new_f (Instance cxt typ ds)
        = Instance cxt typ (recurse m i f new_f ds)
    recurse _ _ _ _ d = d
    recurse_list m i f new_f ds
     | f `elem` defines ds = ds
     | otherwise           = map (recurse m i f new_f) ds
```

5.5.3 Righthand-sides

The instance for right-hand sides is exceptionally simple, just recursing through the expressions. Lists of right-hand sides have no special meaning so the default method is fine.

```
instance Recurse e => Recurse (RightHandSide e) where
    recurse m i f new_f (Guarded ges)
        = Guarded (map (\(e1, e2) -> (recurse m i f new_f e1,
                                      recurse m i f new_f e2)) ges)
    recurse m i f new_f (Normal e) = Normal (recurse m i f new_f e)
```

5.5.4 Arithmetic Sequences

The arithmetic sequences instance is equally simple, featuring simple recursion through to the expressions.

```
instance Recurse e => Recurse (DotDot e) where
    recurse m i f new_f (From e) = From (recurse m i f new_f e)
    recurse m i f new_f (FromThen e1 e2) = FromThen (recurse m i f new_f e1)
                                                    (recurse m i f new_f e2)
    recurse m i f new_f (FromTo e1 e2) = FromTo (recurse m i f new_f e1)
                                                (recurse m i f new_f e2)
    recurse m i f new_f (FromThenTo e1 e2 e3)
        = FromThenTo (recurse m i f new_f e1)
                     (recurse m i f new_f e2)
                     (recurse m i f new_f e3)
```

5.5.5 Statements

The other complicated case is that of statments. If we have a list of statements and the first statement rebinds what we are looking for then we return the list unchanged. Otherwise we recurse on the relevant parts of the first statement to make the head of the result and recurse on the remaining statements to make the tail.

For the single element case we use the slightly bizarre method of wrapping the element up to form a singleton list, calling `recurse` on the list and taking the head.

```
instance (Defines d, Defines p, Recurse d, Recurse e)
      => Recurse (Statement p e d) where
    recurse m i f new_f s = head $ recurse m i f new_f [s]
    recurse_list _ _ _ _ [] = []
    recurse_list m i f new_f ss@(BindSt p e:ss')
```

```
   | f `elem` defines p = ss
   | otherwise
      = BindSt p (recurse m i f new_f e):recurse m i f new_f ss'
  recurse_list m i f new_f ss@(LetSt ds:ss')
   | f `elem` defines ds = ss
   | otherwise
      = LetSt (recurse m i f new_f ds):recurse m i f new_f ss'
  recurse_list m i f new_f (NoBindSt e:ss)
      = NoBindSt (recurse m i f new_f e):recurse m i f new_f ss
  recurse_list _ _ _ _ (ParSt _:_) = error "recurse_list[Statement]: ParSt"
```

5.5.6 Matches and Clauses

Finally we have the cases for matches and clauses, again virtually identical. If either the pattern(s) or declarations bind what we are looking for then the match or clause is unchanged. Otherwise we recurse on the parts that may contain an expression.

```
instance (Defines d, Defines p, Recurse d, Recurse e) => Recurse (Match p e d) where
    recurse m i f new_f mat@(Mat p rhs ds)
     | f `elem` defines ds = mat
     | f `elem` defines p  = mat
     | otherwise           = Mat p (recurse m i f new_f rhs)
                                   (recurse m i f new_f ds)

instance (Defines d, Defines p, Recurse d, Recurse e) => Recurse (Clause p e d) where
    recurse m i f new_f c@(Clause ps rhs ds)
     | f `elem` defines ds = c
     | f `elem` defines ps = c
     | otherwise           = Clause ps (recurse m i f new_f rhs)
                                       (recurse m i f new_f ds)
```

6 Conclusion

We have succeeded in our goal of providing a handful of tools to the Template Haskell user. We have had to make a few concessions to correctness along the way, but hopefully future improvements to Template Haskell will allow us to do without these.

The tools we provide can make a significant difference; for example, in Fraskell[5], a Haskell fractal visualiser, the application of these tools yields almost an order of magnitude speed-up with minimal intrusion into the source. As use of Template Haskell takes off we hope that more uses for this tools can be found, and that more and better tools will be developed as experience is amassed.

As always, any comments you have are welcomed. Drop me an e-mail!

Happy hacking!

Ian Lynagh
igloo@earth.li

[5] See http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/Fraskell/