Unrolling and Simplifying Expressions with Template Haskell

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 introduce a handful of modules for manipulating Template Haskell datastructures, building up to modules that allow simplification and recursion unrolling of Haskell code. To illustrate their use, and to demonstrate the performance improvements that motivate this work, we also discuss an example application Fraskell [5], showing how we can get more than an order of magnitude performance increase with minimal changes to the source.

1. INTRODUCTION

The idea of improving performance by unrolling inner loops is almost as old as the idea of optimisation itself; it's particularly common in the imperative world [3]. Meta-programming systems enable programmers to write libraries that perform such optimisations for other programmers [1, 2]; for example, Blitz++ [8] uses C++ templates provides loop unrolling and other optimisations to the user. For Haskell we are not aware of any previous work on unrolling, although the GHC inliner [6] and Pan [4] work on similar problems. Our contribution is a set of libraries that allow the user to specify that some of their recursions should be unrolled.

In order to provide this functionality we use Template Haskell. Template Haskell extends a plain old Haskell 98 system, giving the user some of the benefits of the meta-programming facilities long enjoyed by the LISP community, without compromising type safety. 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 reasonably complex tasks on these datastructures, building up to complete transformations useful in their own right.

We start, in Section 2, by discussing the motivation behind this work and giving some numbers which show why it is worthwhile. The body of the paper follows, describing the modules we provide and how they work. The early modules define utility functions which are only really of use in building more complex modules, including those which follow. We start with a basic utilities module containing a function to find a variable name that isn't free in an expression and a function to find all the variables bound by a pattern or declaration group in Section 3 followed by a module which makes the substitution of literal values for variables easy in Section 4. This gives us the tools we need for a module to perform simplification in Section 5, which in turn allows us to write the unrolling code described in Section 6. Finally, we finish with a reflection on what we have accomplished in Section 7.

2. MOTIVATION

Our motivation behind this work is to speed up Haskell code by means of transformations applied at compile time; in particular we are interested in unrolling fixed depth recursions. To illustrate the features we are implementing, and to give us something concrete on which to measure the performance improvements, we use Fraskell, a fractal imager which produces pictures of the Mandelbrot set.

2.1 Fraskell and the Mandelbrot Set

A point (x, y) is said to be in the Mandelbrot set if, when you set the complex number variable z to zero and iteratively apply the Mandelbrot function $z = z^2+(x+yi)$, $|z|$ is always at most 2. Figure 1 shows the values of z when we try this with the point $(0.5, 0.5)$; after the 5th iteration $|z| > 2$ (i.e. the point is outside of the radius 2 circle centered on the origin of the Argand diagram[1]) so this point is not in the Mandelbrot set. For the point $(0, 0)$ the value of z is always 0, and for $(-1, 0)$ it cycles between 0 and -1; both of these are therefore in the set. Membership of the set is conjectured to be undecidable, so the convention is to iterate up to a maximum number of times for each point and, if the point has still not been proven to not be in the set we assume that it is in it.

Set membership just gives us a binary value for each point;

[1] A representation of the complex plane, with the real part of a number represented by the distance along the x-axis and the imaginary part by the distance up the y-axis.

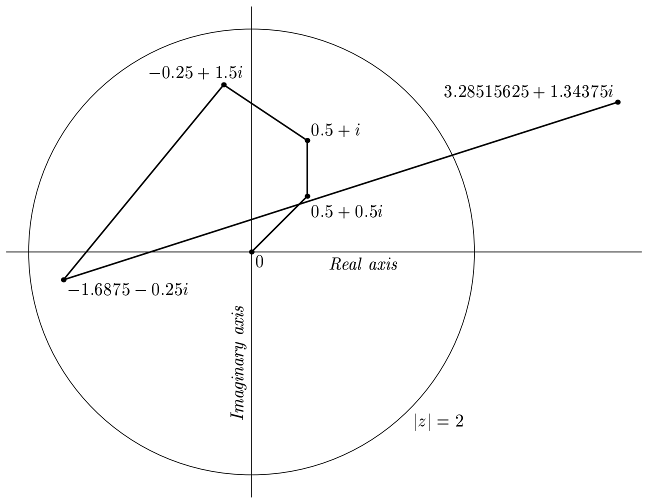

Figure 1: z values while testing $(0.5, 0.5)$ for inclusion in the Mandelbrot set

to get the colourful pictures we are used to seeing we need an outcome with more possibilities. The simplest is to pick a colour based on the number of iterations before we get a value of z such that $|z| > 2$.

Most of the work for a Mandelbrot set imager is performing this repeated calculation and testing. In Fraskell it is handled by the function given in Figure 2.1, which is called for each point in our image. Ignore the '#ifdef'ed sections for now and consider only the Haskell 98 function between them. The first argument is how large a value of $|z|$ we are looking for; for the Mandelbrot set itself this will always be 2, but taking it as an argument allows us to create images of related fractals. Next it takes the number of times we have applied the Mandelbrot function so far (well, nearly—actually it is 10 times this as this value is used directly in determining the RGB colour value, and we prefer the effect of larger colour steps) followed by the z value after this many iterations, and finally it takes the point we are considering as a complex number. It returns the colour we should colour this point represented as a red-green-blue triple of 8 bit integers.

As you would expect it is a recursive function, with each call first checking we haven't hit the maximum iterations already and, if not, calculating the next z value and seeing if it is outside the relevant circle. If we have neither left the circle nor reached the maximum number of iterations then we recurse, with the iteration count 10 higher and the new z value; otherwise we return the appropriate colour.

2.2 Template Haskell

It that was the whole story then this would not be a very interesting paper. Fortunately things get a little more interesting when we define `TEMPLATE_HASKELL`. The code fragment in Figure 2.1 changes from being just a function definition into a 'splice'. Splices are a new feature provided by Template Haskell, with concrete syntax given by the lexemes '`$(`' and '`)`'.

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

#ifdef TEMPLATE_HASKELL
$(
  do
     [typ, fun] <- [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 fun' = unroll (Just 30) 1 (Integer 0) fun
     return [ typ, fun' ]
 )
#endif
```

Figure 2: Core code for Fraskell

At compile time, the contents of all splices[2] are treated as if they were programs in their own right and run to produce a value in the `Q` monad. The value is of type `[Dec]`, where `Dec` is a type that represents the abstract syntax tree of Haskell declarations in a way familiar to those who have taken compiler courses.

Some of the constructors of some of the Template Haskell datatypes are shown in Figure 3 which should help understand the code in the remainder of this paper; for a more complete description see the Template Haskell paper [7]. In particular, note that the `Infix` constructor takes `Maybe Exp`s for its two operands; by setting one or the other of these to `Nothing` we get left and right sections, e.g. `(1 +)` or `(+ 2)`.

The `Q` monad serves a number of purposes: all the reification functions return values in it, it can generate fresh names and it allows meta-programs to perform IO. However, none of these turn out to be useful in the modules we discuss here; the interested reader is referred to [7] for a discussion of these features.

Within our splice everything is standard Haskell 98 except the '`[|d`' and '`|]`' lexemes. These indicate that their contents is to be reified, i.e. the result of evaluating them is a list of abstract syntax trees, represented as values of the `Dec` type, corresponding to the code of each declaration in the enclosed block.

[2]This is a slight simplification, but it suffices for the purposes of this paper. For the details see [7].

```
data Lit = Integer Integer
         | Char Char
         | String String
         | ...

data Exp = Var String
         | Con String
         | Lit Lit
         | App Exp Exp
         | Lam [Pat] Exp
         | Case Exp [Match]
         | Infix (Maybe Exp) Exp (Maybe Exp)
         | ArithSeq DotDot
         | Tup [Exp]
         | Let [Dec] Exp
         | ...

data Match = Match Pat RightHandSide [Dec]
```

```
data RightHandSide = Guarded [(Exp,Exp)]
                   | Normal Exp

data Dec = Fun String [Clause]
         | Val Pat RightHandSide [Dec]
         | Data Cxt String [String] [Con] [String]
         | ...

data Con = Constr String [(Strictness, Typ)]
         | ...

data Pat = Pvar String
         | Pcon String
         | Plit Lit
         | Ptup [Pat]
         | Pwild
         | ...
```

Figure 3: Some Template Haskell datatypes

	Normal	Unrolled
Compilation time	17s	29s
Running time	66m 52s	5m 58s
Total	67m 09s	6m 27s

Figure 4: Time taken to compile and run Fraskell with and without unrolling and simplification

What happens then is, at compile time, the compiler takes the type and definition of `do_mb` as we wrote it and puts the abstract syntax trees into a pair of variables. It then calls `unroll` on the definition of our function and splices in the unchanged type and transformed function definition. The idea here is that `unroll` should produce an optimised, but functionally equivalent, definition to its input. As Figure 2.2 shows the effect is significant, giving more than an order of magnitude speed-up. Simply unrolling the definition accounts for about a 25% speedup while the rest is due to other simplifications performed along the way; we will see why this simplification is necessary later.

We hope that some of the functions we define here, or descendants of them, will be available to users in the future as part of the meta-programming system.

3. UTILS

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 see, will be useful in other modules.

The first function takes either a pattern, a declaration or a list of 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. We won't worry too much about the efficiency of these operations as the functions they will be applied to tend to be small so an easily understood implementation is more worthwhile.

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. Before we discuss these further we will first define a `Set` type.

3.1 Sets

We will use sets of values a lot in this module. In order to make this clear we would like to use `Set` in the type signatures. However, we don't want to lose easy access to the multitude of prelude functions which operate on lists. We therefore define the `Set` type to be a type synonym for lists; note that it is our responsibility to ensure that an element can appear only once in a set.

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

It turns out 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
```

3.2 Finding bound variables

We will also want to be able to find out which variables are bound by both patterns and declarations; we define a function `defines` for this. As `defines` needs to operate differently on different types it makes sense to define it as a member of a Haskell class. The type follows naturally.

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

A Template Haskell abstract syntax tree has lists of both declarations and patterns; therefore we would like a list instance. To find out what a list of things defines we just take the union of what each one defines.

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

The `Pat` instance is similarly straightforward—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 set.

```
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 _ = []
```

Finally we need an instance for the `Dec` datatype. 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.

No binding is done by `instance` declarations so `Instance` can be caught by the default case. By contrast, we treat `class` declarations as defining everything bound by their declarations. As declarations in classes commonly give only a type signature without a default method we must also treat type signatures as defining variables.

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 (Proto f _) = [f]
    defines (Foreign (Import _ _ _ f _)) = [f]
    defines _ = []
```

3.3 Getting new variable names

The definition of `new_var` itself is deceptively simple. We take an infinite set of variable identifiers, remove all the identifiers which appear as free variables in the expression we are given and return the first one remaining. 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; 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']
```

Finally `multiperms` returns the infinite set of non-empty permutations of its input where each element can be repeated.

```
multiperms :: Set a -> Set [a]
multiperms xs = tail $ concat $ iterate next [[]]
    where next ys = [ x:y | y <- ys, 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 put it in 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, so some of the uninteresting cases have been omitted here for brevity. 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 cases are `Lambda` and `Let` where we need to remove the set of variables defined by the patterns and declarations respectively which are otherwise free in the expression; for this we use `defines` which we wrote earlier.

```
instance FreeVars Exp where
    free_vars (Var v) = [v]
    free_vars (App e1 e2) = free_vars e1
                    `union` free_vars e2
    free_vars (Infix l o r)
        = free_vars o `union` vl `union` vr
```

```
        where vl = fromMaybe [] $ fmap free_vars l
              vr = fromMaybe [] $ fmap free_vars r
    free_vars (Lam ps e)
        = free_vars e \\ defines ps
    free_vars (Tup es) = free_vars es
    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 (SigExp e _) = free_vars e
    ...
```

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) = frees \\ redefed
     where in_rhs = free_vars rhs
           in_ds = free_vars ds
           frees = in_rhs `union` in_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 trivial so we omit them here for conciseness.

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 statement binds a variable v then that variable is bound only in later statements. 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 FreeVars Statement where
    free_vars = flip do_stmt []
    free_vars_list = foldr do_stmt []

do_stmt :: Statement -> 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
```

There are two cases left to define, `Clause` and `Mat`. The pair are virtually identical in both structure and meaning, so unsurprisingly the instances are almost identical; they are also very simple.

```
instance FreeVars Mat 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 FreeVars Clause 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
```

4. SUBSTITUTIONS

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 occurrences of a variable in some expression with a literal value which you have calculated[3]. While this is conceptually straightforward it is a reasonable amount of code, so we provide it in a module to facilitate reuse.

Throughout this section we will again omit uninteresting clauses of large instances, and even entire instances when they are straightforward.

4.1 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 that we again 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)
```

As you would expect, to substitute into a list of things we just use the `subst_list` method.

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

4.2 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

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

datastructures that are or might contain expressions, using `defines` in the `Lam` and `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 (Tup es) = Tup (map (subst v l) es)
    subst v l (Do ss) = Do (subst v l ss)
    subst v l (Lam ps e)
     | v `elem` defines ps = Lam ps e
     | otherwise = Lam ps (subst v l e)
    subst v l (Let ds e)
     | v `elem` defines ds = Let ds e
     | otherwise = Let (subst v l ds) (subst v l e)
    ...
```

4.3 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 `subst` 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 ctxt tc tvars ds)
             = Class ctxt tc tvars (subst v l ds)
    subst v l (Instance ctxt typ ds)
             = Instance ctxt typ (subst v l ds)
    subst _ _ d = d
    subst_list v l ds
     | v `elem` defines ds = ds
     | otherwise           = map (subst v l) ds
```

4.4 Statements

The other complicated case is that of statements. If we have a list of statements and the first is a let statement that rebinds what we are looking for then the while list is unchanged. Alternatively, if the first is a binding statement then we substitute only on its right hand side, leaving the tail of the list unchanged. For any other head we just recursively substitute on both the head and tail of the list as normal.

For the single element case we use the slightly bizarre technique of wrapping the element up to form a singleton list, calling `subst` on the list and taking the head. Otherwise we would have to write the cases out again.

```
instance Subst Statement where
    subst v l s = head $ subst v l [s]
    subst_list _ _ [] = []
    subst_list v l (BindSt p e:ss)
     | v `elem` defines p = this:ss
     | otherwise = this:subst v l ss'
       where this = BindSt p (subst v l e)
    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
```

4.5 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 Subst Mat 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)
```

5. SIMPLIFICATIONS

So far we have built up some 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[4], returns a simplified, but functionally equivalent, expression. This can be used 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).

5.1 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 we are interested in are things like constant folding, so we are not concerned with the different semantics of lists of different things.

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

Also, it is likely that we will wish to simplify quasi-quoted code and then immediately 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
```

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

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
```

5.2 Expressions

Although most of the cases, such as `Let`, are handled by just recursing on the arguments, `Exp` is also where all the interesting cases are.

```
instance Simplify Exp where
    simplify (Let ds e)
     = Let (simplify ds) (simplify e)
```

The first case where we actually do something is to replace the variable `GHC.Base:otherwise` with the constructor `GHC.Base:True`; this doesn't seem useful itself, but it will allow optimisations to be defined more easily later on. It is unfortunate that we need to state the "`GHC.Base:`" prefix, which specifies where the value was defined to distinguish variables of the same name from different modules, in the source; not only is it ugly but it means that all implementations must agree on these names. When using the quasi-quotes we could simply write `[| otherwise |]`, but this is monadic and cannot be used for pattern matching.

```
    simplify (Var "GHC.Base:otherwise")
     = Con "GHC.Base:True"
```

The really interesting case is that for infix expressions. Here, if we know the values of at least one of the operands it may 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 (Var op) me2)
     = let me1' = fmap simplify me1
           me2' = fmap simplify me2
       in case (me1', op, me2') of
```

We can now do the various simplifications you would expect such as constant and operation identity folding in comparisons and arithmetic.

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

Note that we are making the assumption that the operands are compared, summed etc. in the same way as `Integer`s, which may be invalid. Even `Int` arithmetic may go wrong due to the lack of overflow! However, it is infeasible with the current Template Haskell system to do anything else.

For any other infix expression we just put it back together with the simplified operands.

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

The final interesting case is for conditionals. If we find that, after simplification, the guard in a conditional statement is either `True` or `False` then we just return the relevant expression simplified; otherwise we 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)
```

5.3 Other datatypes

Instances are needed for a number of other datatypes, but they are all simply the obvious recursion, so we omit them here.

6. UNROLL

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.

6.1 What are we doing?

Before we start writing code let us have it clear in our minds what interface we wish to expose to users. 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 avoid problems with argument renaming), which must be a simple pattern variable, to unroll with respect to, and also what value this argument will have when it is called by other functions. The function will continue to behave correctly if this value is not used, but the unrolled definition 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 users to protect themselves from accidentally making the compiler not terminate by optionally specifying 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.

6.2 Some handy types

Let us take a minute to define a couple of types to allow us to give more descriptive type signatures 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`, with 0 meaning the first argument.

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

6.3 Unrolling recursions

To actually unroll the recursion we will use a pair of mutually recursive functions. A function `recurse` recurses over the datastructure and wherever it finds a recursive call to the function we are interested in it calls `unroll'`. This creates an inlined copy of the function definition and calls `recurse` on it. A function `unroll` is the front-end to this pair; it sets up some values to be passed around by the recursive pair and then calls `unroll'` to build the new right hand side.

6.3.1 The wrapper function

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 the `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 5(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 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
unroll m i l (Fun f [c@(Clause ps rhs decs)])
 | length ps <= i = error "Not enough patterns"
 | otherwise = Fun f [Clause ps' (Normal e) [], c]
```

To create the pattern list for the new clause 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
           (x, _:y) -> x ++ [Plit l] ++ y
```

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_exp`. 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 xs -> foldr (uncurry Cond) err xs
  err = App (Var "error")
            (Lit (String "Generated by unroll"))
```

However, `rhs_exp` 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 get the same effect by building a let expression which binds the declarations and then evaluates the expression within that environment. This gives us `rhs_exp'` which equates to Figure 5(b).

There are two things worth noting here: first, if we used `rhs_exp'` 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 the lambda abstraction that binds the variables in the patterns of the function definition and then returns `rhs_exp'` 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 5(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's right-hand side, i.e. Figure 5(c) in our example, with the literal substituted for all occurrences 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 5(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.

```
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 by 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 5: Step-by-step unrolling

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_exp = case ps !! i of
        Pvar v -> \lit -> subst v lit rhs_exp'
        _ -> error "unroll: ps !! i not a Pvar"
    sub_rhs_lambda = \v -> Lam ps (sub_rhs_exp 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.

6.3.2 The worker function

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 substitution function (SF) arguments that take 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 SF will either be the left SF in Figure 5(d) or the inner part (identical to the right SF in Figure 5(d)) of Figure 5(g) or 5(j) in the example; the first will always be the right SF in Figure 5(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 5(e), 5(h) and 5(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 5(f), 5(i) and 5(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 m' = fmap (\x -> x - 1) m
          e' = simplify (e l)
          e'' = recurse m' i f new_f e'
```

6.4 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 write the obvious list instance as with substitutions.

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

We will continue to omit uninteresting instances and clauses.

6.4.1 Expressions

The hard work is, as with substitutions, all done by the expression class; this case differs in that it is done in the function application case rather than the variable 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
              -> unroll' m i l f new_f new_f
               : map (recurse m i f new_f) xs
            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 (Tup es)
        = Tup (map (recurse m i f new_f) es)
    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 (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)
```

6.4.2 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 _ _ f _ d | f `elem` defines d = d
    recurse m i f new_f (Fun func cs)i
        = Fun f (recurse m i func 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_list m i f new_f ds
     | f `elem` defines ds = ds
     | otherwise = map (recurse m i f new_f) ds
```

6.4.3 Statements

The other complicated case is that of statements. 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 strange technique of wrapping the element up to form a singleton list, calling `recurse` on the list and taking the head.

```
instance Recurse Statement 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 (BindSt p e:ss)
     | f `elem` defines p = this:ss
     | otherwise = this:recurse m i f new_f ss
      where this = BindSt p (recurse m i f new_f e)
    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
```

6.4.4 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 Recurse Mat 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 Recurse Clause 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)
```

7. CONCLUSION

We have succeeded in our goal of providing a handful of tools to the Template Haskell user, but due to the current system we have had to make a few concessions to correctness along the way.

The reason we have had to sacrifice correctness in this case is because we are not able to ask the meta-programming system what type it could infer for expressions. Therefore when we come to simplify arithmetic of constants we have to make assumptions about what their real type is. For this relatively simple case it would be sufficient to annotate leaves of the tree with their type by giving the `Var`, `Con` and `Infix`[5] constructors a `Typ` component; if nothing could be inferred then this would just be a type variable.

It would be nice to have access to the type of a whole expression tree without having to work it out from the leaves yourself, particularly as this may be done repeatedly over various subtrees. This is not even possible with the current implementation as there is no way to determine the types of functions without having the function name in the source. It is not clear how to allow access to type information anywhere in an expression though; simply annotating all over the tree would be very cumbersome for applications that aren't interested in the type information.

It could be argued that everything we have done here could instead be done by introducing some new pragmas to GHC. This is true—indeed, anything that could be written with Template Haskell could be implemented as pragmas. However, Template Haskell allows the user to extend the system with familiar syntax and performs typechecking, so it is easier to write correct code. It also means that it is not necessary to recompile the GHC source code or wait for the next release to take advantage of new functionality.

The tools we provide can make a significant difference, as we showed by applying them to Fraskell, a Haskell fractal visualiser, to get more than 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 these tools can be found, and that more and better tools will be developed as experience is amassed.

[5]For `Infix` the type would be that of the operator.

8. ACKNOWLEDGEMENTS

I would like to thank Manuel Chakravarty, Shae Erisson, Oege de Moor, Simon Peyton Jones and Ganesh Sittampalam for their helpful comments on a draft of this paper. Oege and Ganesh deserve special thanks for being available for discussion and offering advice during both the work carried out and the writing of this paper.

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

9. REFERENCES

[1] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[2] K. Czarnecki, U. W. Eisenecker, R. Glck, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming 1998*, volume 1766 of *LNCS*, pages 25–39, 2000.

[3] J. W. Davidson and S. Jinturkar. An aggressive approach to loop unrolling. Technical Report CS-95-26, Department of Computer Science, University of Virginia, Charlottesville, June 1995.

[4] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generations (SAIG 2000)*, volume 1924 of *LNCS*, pages 9–27. Springer-Verlag, Sept. 2000.

[5] I. Lynagh. Fraskell. URL: `http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/Fraskell/`.

[6] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4&5):393–433, July 2002.

[7] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[8] T. L. Veldhuizen. C++ templates as partial evaluation. In *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio, January 1999.*, pages 13–18. University of Aarhus, Dept. of Computer Science, Jan. 1999.