CPSC 449:
Notes on matching and unification
(for discussion in labs!)

Robin Cockett

October 27, 2021

1 Matching

We have already met the idea of matching in hand evaluating Haskell programs. However, we can actually program it!! The idea is given two trees the first of which we view as a "template" and the second of which we call the "subject", can a substitution of the variables in the template be found which allow us to reconstruct the subject of the matching? Of course, there may be no such substitution, that is matching may fail ... so we shall have to return a success or fail of a list of substitutions.

Now in the matching algorithm the subject tree is not really supposed to have any variables. The template is the same sort of tree except it must have variables. So really the template and the subject are actually slightly different types of trees. In these notes we shall make the template be an expression tree, `Exp f v`, and the subject be a rose tree, `Rose f`. Furthermore, the template must not have any repeated variables: this ensures that variables are replaced in a unique way.

Expressions are Rose trees with variables

Let us start by introducing the two main datatypes we shall be using:

```
data Rose f = Rs f [Rose f]

foldRose:: (f -> [c] -> c) -> (Rose f) -> c
foldRose h (Rs args) = h f (map (foldRose h) args)

data Exp f v = Var v
             | Opn f [Exp f v]

foldExp::(f -> [c] -> c) -> (v -> c) -> (Exp f v) -> c
foldExp h g (Var v) = g v
foldExp h g (Opn f args) = h f (map (foldExp h) args)
```

Below is the basic matching program: it has the template as the first argument and the subject as the second argument and outputs (if it succeeds) a list of substitutions of the variables of the template.

```
match:: (Eq f) => (Exp f v) -> (Rose f) -> (SF [(v,Rose f)])
match (Var v) t= SS [(v,t)]
match (Opn f1 targs) (Rs f2 args)
   | f1==f2 && (length targs == length args)
           = mapsf concat $ sflist (map (\(tt,t) -> match tt t)$ zip targs a
   | otherwise = FF
```

Now there is a another wrinkle to (pattern) matching which is that the template should not have repeated variables. To check for this we shall follow the steps:

(1) Obtain a list of all the variables in the template

(2) Check that the list is non-repeating (note this is unavoidably expensive, $\mathcal{O}(n^2)$) -

Now (1) can be achieved very simply using a fold:

```
collectvar:: Exp f v -> [v]
collectvar = foldExp (\f vargs -> concat vargs) (\x -> [x])
```

For (2) we have:

```
nonrepeating:: Eq a => [a] -> Bool
nonrepeating (a:as) = (nonrepeating as) && not (member a as)
    where member a [] = False
          member a (b:bs)| a==b = True
                         | otherwise = member a bs
nonrepeating [] = True
```

Note that this test is quadratic: every (unordered) pair of elements of the list must be compared.

So now we can put this together into a "pattern matching" algorithm:

```
pmatch:: (Eq f,Eq v) => (Exp f v) -> (Rose f) -> (SF [(v,Rose f)])
pmatch t template
                 | nonrepeating $ collectvar template = match t template
                 | otherwise = FF
```

Here pattern matching simply fails if one does not have a "legal" template to match against!

2 Unification

What is unification

Prolog, our next language, uses more powerful version of matching called "unification" which is a sort of matching on steroids ...

Recall that the idea of matching is that we are given a template (or a "pattern"), which is an expression/term , s, with distinct variables, and a "subject" t, which is another expression/term, which does not have any variables. The expression/term, t is said to match the template/pattern s in case there is a substitution $[x_1 := t_1, ..., x_n := t_n]$ of t such that

$$s = t[x_1 := t_1, ..., x_n := t_n]$$

The idea of **unification** is to have a completely symmetric sort of matching: thus, we are given two arbitrary terms s_1 and s_2 (both with variables and with no restriction on those variables) and the objective is to find a substitution which makes them equal. This means a substitution

$$[x_1 := t_1, ..., x_n := t_n]$$

such that

$$s_1[x_1 := t_1, ..., x_n := t_n] = s_2[x_1 := t_1, ..., x_n := t_n].$$

Such a substitution is called a **unifier** of the two terms s_1 and s_2: applying the substitution make the terms syntactically exactly equal.

Example:

(i) To unify the terms

$$f(x, g(y, z))?=?f(v, v)$$

we could use the substitutions

$$[x := g(u, z), v := g(y, z)]$$

Applying this substitution to both of these terms yields $f(g(y, z), g(y, z))$

(ii) The unification of

$$f(x, y)?=?g(v, w)$$

fails as there is no substitution which can make these terms equal.

(iii) The unification problem

$$f(x)?=?z$$

has many solutions:

$$[z := f(x)] \text{ or } [x := g(w), z := f(g(w))]$$

for example. Clearly the first of these is "more general".

However, as we note above, there can be many unifiers! Fortunately, up to naming of variables, there is a **most general unifier**. This is a substitution which is:

(a) a unifier but has the additional property that

(b) any other unifier can be obtained by substituting it.

Let us make that property (b) a bit more explicit. Suppose we are unifying s_1 and s_2 (so the unification problem is $s_1?=?s_2$) and we are given a unifier

$$[x_1 := t_1, ..., x_n := t_n]$$

so that as above

$$t = s_1[x_1 := t_1, ..., x_n := t_n] = s_2[[x_1 := t_1, ..., x_n := t_n].$$

A most general unifier $[x_1 := u_1, ..., x_n := u_n]$ is a unifier, so

$$t' = s_1[x_1 := u_1, ..., x_n := u_n] = s_2[x_1 := u_1, ..., x_n := u_n]$$

but, furthermore, there is a substitution of t' which obtains t, that is there is a $[y_1 := v_1, ..., y_m := v_m]$ so that

$$t = t'[y_1 := v_1, ..., y_m := v_m]$$

This is a bit technical! However, to understand Prolog –and type inference in Haskell –you really need to understand unification!! Of course, recall, just as for matching, unification can fail: that is given two terms s_1 and s_2 it is always quite possible that there is no unifier.

There is a huge amount of fundamental Computer Science work on unification (try googling "unification" (computer science)): the main algorithm for unification (Alberto Martelli and Ugo Montanari) was developed as recently as 1982 although the ideas go farther back. It is an important core computer science algorithm.

Unification by hand

How is the most general unifier calculated?

Basically one goes through a process of matching and trying to "reify" basic subproblems (with one side a variable) into being actual substitutions. As one does this one must remove all occurrences of the variable being reified from the remaining problems by substituting all occurrences of that variable with the term in the reified substitution. One repeats this process until one has a list of reified substitutions and no subproblems left.

Let us go through some examples to get a feel of what can happen:

(1) Consider the example of trying to calculate the unifier of:

$$f(g(x, v), y)?=?f(w, h(w, v))$$

The first step is to match the two terms overlaying one term with the other and whenever a variable end one term and the other continues one obtains a subproblem. In the above case matching produces two subproblems:

$$w?=?g(x, v), y?=?h(w, v)$$

Now what we try to do is to "reify" the first subproblem as substitution ... while removing all occurrences of w from the remaining problem(s). In this case this gives:

$$\text{Reified: } w := g(x, v) \quad \text{Subproblems: } y?=?h(g(x, v), v)$$

In this case we can proceed to reify the subproblem to obtain:

$$\text{Reified: } y := h(g(x, v), v), w := g(x, v) \quad \text{Subproblems:}$$

The reified list is the most general unifier, and as we shall see it is, by construction, a linearized list of substitutions.

(2) Consider the problem:

$$f(g(x, w), y)?=?f(w, h(w, v))$$

If we match we obtain the subproblems:

$$w?=?g(x, w) \text{ and } y?=?h(w, v)$$

The first term fails the occurs check: notice that the variable w occurs non-trivially in $g(x, w)$. Thus, unification fails.

(3) Consider the unification problem:

$$f(x, x)?=?f(g(y, z), g(z, v))$$

matching produces the subproblems:

$$x?=?g(y, z) \text{ and } x?=?g(z, v)$$

Now when we reify the first subproblem into a substitution and we remove the variable x from the remaining subproblem(s) we obtain:

$$\text{Reified: } x := g(y, z) \quad \text{Subproblems: } g(y, z)?=?g(z, v)$$

This time the effect of substituting produces a new unification problem which results in another matching step:

$$\text{Reified: } x := g(y, z) \quad \text{Subproblems: } y?=?z, z?=?v$$

Now we can reify one of these matched subproblems as a substitution:

$$\text{Reified: } y := z, x := g(y, z) \quad \text{Subproblems: } z?=?v$$

and we can keep reifying:

$$\text{Reified: } z := v, y := z, x := g(y, z) \quad \text{Subproblems:}$$

Notice that the substitutions are linearized and give a unifier – the most general one – of the original problem. We may parallelize the substitutions to obtain:

$$[z := v, y := v, x := g(v, v)]$$

(4) One more example to show what can go wrong. Consider:

$$f(x, h(y))?! =?f(g(y, z), h(g(z, x)))$$

Matching produces subproblems:

$$x?=?g(y, z) \text{ and } y?=?g(z, x)$$

We reify x to obtain:

$$\text{Reified: } x := g(y, z) \quad \text{Subproblems: } y?=?g(z, g(y, z))$$

Notice that $y?=?g(z, g(y, z))$ fails the occurs check (as y occurs in $g(z, g(y, z))$). Thus, there is no unifier for this problem.

Using unification for type inference

An important application of unification is in type inference. Haskell, together with all all modern functional languages, use the Hindley-Milner type inference system. This type inference system is quite remarkable: it is *the* most successful formal method in programming. It routinely catches most of the errors in developing a functional program! While we shall not explain exactly how the algorithm works we show a couple of examples of how it does its job.

Example:

(1) Consider the following given types:

```
foldr :: (a -> c -> c) -> c -> [a] -> c
(.) :: (b -> c) -> (a -> b) -> (a -> c)
id :: a -> a
```

We want to determine the most general type of `foldr (.) id`. We shall do this by first obtaining the most general type of `foldr (.)` and then obtaining the most general type of `foldr (.) id`.

The very first step is to "separate" the variable in the different definitions:

```
foldr :: (a1 -> c1 -> c1) -> c1 -> [a1] -> c1
(.) :: (b2 -> c2) -> (a2 -> b2) -> (a2 -> c2)
id :: a3 -> a3
```

Then notice that `(.)` is substituted into the first argument of the `foldr`: for this to work the types `a1 -> c1 -> c1` and `(b2 -> c2) -> (a2 -> b2) -> (a2 -> c2)` must be the same. We make them the sam by finding the most general unifier. This is easily seen to be:

```
[b2 := c2, c_1:= a2 -> c2, a1:= b2 -> c2]
```

Note this is only unique up to the renaming of variable.

This means that we have

```
foldr (.):: c1 -> [a1] -> c1[b2 := c2, c_1:= a2 -> c2, a1:= b2 -> c2]
       = (a2 -> c2) -> [c2 -> c2] -> (a2 -> c2)
```

The next step is to unify `a2 -> c2` with the type of `id`: this gives the unification problem

```
a2 -> c2 ?=? a3 -> a3
```

which clearly has most general unifier `[a2:=a3,c2:=a3]`. This gives the type

```
foldr (.) id:: [c2 -> c2] -> (a2 -> c2)[a2:=a3,c2:=a3]
               = [a3 -. a3] -> (a3 -> a3)
```

(2) One can also use the type inference to determine what the types of the phases of a `foldr` must be to give a particular resulting type to the fold. Thus for example if we want `foldr f g:: [[a]] -> [a]` what must the types of `f` and `g` be? To determine this we must unify the output of a fold with the desired output:

```
[a1] -> c1 ?=? [[a]] -> [a]
```

This has most general unifier `a1:=[a], c1:=[a]`. This means the types of `f` amd `g` must be:

```
f:: a1 -> c1 -> c1[a1:=[a], c1:=[a]]
          = [a] -> [a] -> [a]
g:: c1[a1:=[a], c1:=[a]] = [a]
```

3 Unification in Haskell (first cut!)

OK so let us write the unification algorithm in Haskell. We shall not shoot for efficiency but rather to an algorithm which mimics the hand unification technique above. We shall perform unification between terms of the datatype of expressions (as introduced above with its fold).

A substitution for expressions with function type f and variable type v looks like:

```
type Sub f v = (v,Exp f v)
type Subs f v = [(v,Exp f v)]
```

Substitutions

Lets start with a warm up and program a single substitution:

```
substitution:: Eq v => (Sub f v) -> (Exp f v) -> (Exp f v)
substitution (v,s) = foldExp Opn (\w -> if v==w then s else Var w)
```

Note all this involves is substituting the leaves holding a variable equal to v with the term s: otherwise one leaves the leaves as they are.

So now lets program the application of a whole substitution list ... How do we do this? Well we can use a fold over lists:

```
substitutions:: Eq v => (Exp f v) -> (Subs f v) -> (Exp f v)
substitutions t = foldr substitution t
```

There is something to notice with this substitution as it performs the substitutions sequentially in the sense that the first substitution performed is the last in the list:

$$t[x_1 := s_1, ..., x_n := s_n] = (...(...(t[x_n := s_n])...)[x_2 := s_2])[x_1 := s_1])$$

this means that substitutions earlier in the list can substitute the substitutions later in the list but not vice versa.

A substitution list is **linearized** if, for $j \geq i$, the variable x_j does not occur in the term s_i. (Note that this means in each substitution $x_i := s_i$, x_i cannot *occur* in s_i (we will see this is an important property).

When, for every i and j the variable x_j does not occur in the term s_i then the substitution is said to be **parallel** (or **parallelized**) and this means that the order in which the substitutions are performed does not matter.

It is important to realize that a linearized substitution list can be turned into a parallel substitution by substitution each substitution term by all the substitutions which preceded it:

$[x_1 := s_1, ..., x_n := s_n]$ parallelizes to

$$[x_1 := s_1, x_2 = s_2[x_1 := s_1], x_3 := s_3[x_1 :=: s_1, x_2 := s_2]..., x_n := s_n[x_1 :=: s_1, ..., x_n := s_n]]$$

Here is the program for parallelizing a linearized substitution:

```
parallelize:: Eq v => (Subs f v) -> (Subs f v)
parallelize = foldr pstep [] where
   pstep:: Eq v => (Sub f v) -> (Subs f v) -> (Subs f v)
   pstep sub subs
      = sub:(map (\(x,t) -> (x,substitution sub t)) subs)
```

As we shall see the objective of the unification algorithm is to produce a linearized substitution.

3.0.1 Handling failure ...

As the unification algorithm can fail in multiple ways we have to handle failure. For this we will use the success or fail datatype and its cast of programs. As we (hopefully) will discover with some monadic programming the somewhat clunky explicit exception handling we shall use here can be cleaned up.

Here is the success or fail datatype and some basic utilities:

```
data SF  a = SS a | FF
  deriving (Show,Eq)

sfmap::(a -> b) -> (SF a) -> (SF b)
sfmap f (SS a) = SS (f a)
sfmap f FF = FF

sflist:: [SF a] -> (SF [a])
sflist = foldr (\x xs -> case (x,xs) of
                            ((SS a),(SS as)) -> SS(a:as)
                            _ -> FF)
               (SS [])

sfflatten:: (SF (SF a)) -> SF a
sfflatten (SS x) = x
sfflatten FF = FF
```

The occurs check

The unification algorithm starts with a "matching" step in which the trees are matched against each other and whenever one tree ends in a variable where the other tree continues one obtains a substitution. The substitution is only valid, however, if it passes what is called the "occurs check" this checks that the substituted variable does *not* occur in the term which is to replace it. Without the occurs check one gets cyclic or infinite tree ....

Now let us program the occurs check. Note that v occurring in the expression which is the variable v itself, is fine: but no substitution is needed hence an empty list of substitutions is returned: this is why a substitution list is produced.

```
occurs_check::(Eq f,Eq v) => (Sub f v) -> (SF (Subs f v))
occurs_check (v,t)| t == (Var v) = SS []
```

```
                  | occurs v t = FF
                  | otherwise = SS [(v,t)]
    where
         occurs v = foldExp (\_ -> foldr (||) False) ((==) v)
```

The matching step and linearizing

Here now is the matching step:

```
matching::(Eq f,Eq v) => (Exp f v,Exp f v) -> (SF (Subs f v))
matching ((Var v),t) = occurs_check (v,t)
matching (s,(Var v)) = occurs_check (v,s)
matching ((Opn f1 args1),(Opn f2 args2))
   | f1 == f2 && (length args1) == (length args2)
              = sfmap concat $ sflist $ map matching $ zip args1 args2
   | otherwise = FF
```

Matching produces a non-linearized substitution list: we need to go through this substitution list to "tidy" it up into a linearized substitution. Thus we want a list of substitutions

$$[x_1 := t_1, x_2 := t_2, ..., x_n := t_n]$$

such that for $1 \geq i \geq j \geq n$ the variable x_j does not occur in t_i. We can achieve this by "substituting out" the variable x_i by replacing it with t_i in each later substitution t_j/x_i. This is the business of "reifying" the substitution in the hand calculations of unifiers.

There are two basic things that can go wrong when trying to reify a substitution:

(1) First x_i could equal x_j in which case we retain the first substitution but replace the second by the substitutions obtained from matching on the two right hand sides (as they must be made equal!).

(2) Second one can perform the substitution of x_i and the result can fail the occurs check as x_j occurs in t_i ... in which case everything must fail.

This gives:

```
linearize:: (Eq v, Eq f) => (Subs f v) -> (SF (Subs f v))
linearize subs = lin_helper [] subs
     where
        lin_helper:: (Eq v,Eq f) => (Subs f v) -> (Subs f v) -> (SF (Subs f
        lin_helper (lin_subs) [] = SS lin_subs
        lin_helper lin_subs (sub:subs) =
          case (reify sub subs) of
            SS nsubs -> lin_helper (sub:lin_subs) nsubs
                        -- reify the substitution!
            FF -> FF
```

The process of removing the variable of a substitution from the rest of the substitution list, which is the crucial step in "reifying" a substitution is given by:

```
reify:: (Eq f,Eq v)=>(Sub f v)->(Subs f v)->SF (Subs f v)
reify s [] = SS []
reify (sub@(x,t1)) ((y,t2):subs)
           | x==y = case matching (t1,t2) of
                        SS msubs -> case (reify sub subs) of
                             SS subouts ->  SS (msubs++subouts)
                             _ -> FF
                        _ -> FF
           | otherwise = case occurs_check (y,substitution sub t2) of
                           FF -> FF
                           SS subst -> case (reify sub subs) of
                                    SS subouts -> SS (subst++subouts)
                                    _ -> FF
```

Please note the ugly nested case statements!! ... we shall show how this code can be improved using the exception monad shortly.

Unification

Then we have a unify which gives a linearized substitution:

```
unify:: (Eq f,Eq v) => (Exp f v) -> (Exp f v) -> (SF (Subs f v))
unify t1 t2 = sfflatten $ sfmap linearize $ matching (t1,t2)
```