

CS637: Unification Programming Project

Prof. Boris Aronov

November 30, 2004

1 The Problem

In this project, you will get acquainted with the concept of unification. *Unification* is a generalization of pattern matching that is used in Haskell (in particular, in function definitions and **case** expressions) and also in Prolog. For example, in Haskell, to define a function `fun::[(Int,String)]->Int`, we might use several different rules, depending on *values* of the actual parameters to the function. One such rule might look like this

```
fun ((17,a):rest) = ...
```

Here the *pattern* “(17,a):rest” is matched against the value of the actual parameter, which is a list of integer-string pairs. The head of the list is checked first (if the list is empty, the match fails immediately) and matching process succeeds if the first element is a constant 17. The second element technically is matched (*unified*) with the variable `a`, which means that `a` is *bound* to the value of this second element. This last unification, of a free variable with a value, never fails in Haskell, because in this language, any variable can appear in a pattern at most once. So one will never try to match it to two or more values at the same time. The general problem is a little more involved, see examples below. (Returning to the above example, when evaluating `fun []` the pattern match fails immediately, as the list is empty. For `fun [(1,“boo”)]` the pattern match fails, as the number is 1, not 17. For `fun [(17,“wow”),(10,“how”)]` the pattern succeeds with, after unification, variable `a` bound to the string “wow” and `rest` bound to the list [(10,“how”).])

So what exactly *is* unification? We are not talking about the political process in some country across the pond, but about the following concept, roughly: First, for simplicity, let the set of all *values* be the integers, Haskell **Ints**, for example) and that the set of variable names are Haskell **Strings** (in Lisp or Prolog, picking *atoms* to represent variables would more be natural). With this restriction, matching a variable to a value would be just binding it to an integer; matching two values is just checking if they are the same integers (and producing an error if they are not, which will cause the unification process to *fail*); matching two variables is asserting they must share a common value. This is the complete description of the unification algorithm, provided we only run it on a single pair of items, each either a variable or an (integer) value. One can then naturally extend unification to a pair of lists,

where corresponding elements of two lists have to be matched up, *and where the bindings that you get from all these match-ups have to be globally consistent*. In this case, more interesting things can and do happen. More precisely, let us model the possible patterns, for now, as follows (“deriving Show” just makes the type nicely printable in Haskell)

```
data MyElement = MyNum Int | MyAtom String deriving Show
```

We then define a type for returning bindings, which you will probably think is an overkill:

```
type MyBinding = ([String],Maybe Int)
type MyBindings = [MyBinding]
```

I.e., MyBinding is a single binding, represented as a list of variables, all bound together to the same value, which is an **Maybe Int**, meaning that it is a data type that can store *either* an **Int** value, such as 7 (represented in Haskell as **Just 7**), *or* nothing at all, represented as **Nothing**. So for example ([“a”, “b”, “c”], **Just 17**) represents the binding of three variables to the same value, 17; ([“x”, “y”], **Nothing**) represents the fact that variables “x” and “y” are bound together (to have the same value) but the actual value has not been determined yet. MyBindings represent a list of such bindings. The lists of variables mentioned in different bindings in the list have to be disjoint, to be meaningful. (Enforcing this condition is precisely the purpose of the function merge_bindings, see the discussion below.)

Finally, we come to the actual definition of the unification function. We first define it to match two elements:

```
unify1 :: MyElement -> MyElement -> MyBindings
unify1 (MyNum x) (MyNum y)
  | x == y = [] -- no variable binding is created here, as only values are present
  -- This is just to produce a nicer error. Could have omitted the
  -- next line and failed.
  | x /= y = error ("Unification:␣" ++ (show x) ++ "!=" ++ (show y))
unify1 (MyAtom x) (MyAtom y)
-- binding a variable to itself is pretty useless
  | x == y = []
-- create a single binding that identifies x with y, no value is recorded.
  | x /= y = [[x,y],Nothing]

unify1 (MyNum x) (MyAtom y) = unify1 (MyAtom y) (MyNum x) -- swap arguments
unify1 (MyAtom x) (MyNum y) = [[x],Just y] -- variable x is bound to value y
```

More generally, we would like to handle, say, unifying several such pairs simultaneously or unifying two lists of elements. For this purpose we define two auxiliary functions

```
unify2lists :: [MyElement] -> [MyElement] -> MyBinding
unify_pairs :: [(MyElement,MyElement)] -> MyBinding
```

which we could define via something like this:

```

unify2lists x y -- check that lists have equal lengths and reduce to unify_pairs
| length x == length y = unify_pairs (zip x y)
| otherwise == error "Unequal length lists" ++ (show x) ++ " and " ++ (show y)

-- nothing to do for an empty list, else handle the first pair,
-- handle the rest recursively, combine the bindings by merge_bindings
unify_pairs [] = []
unify_pairs (x,y):tail = merge_bindings (unify x y) unify_pairs tail

merge_bindings :: MyBindings -> MyBindings -> MyBindings

```

One still has to define of the function `merge_bindings`. In the source code file you can play with, mentioned below, I defined simply as

```
merge_bindings x y = x ++ y
```

which is not sufficient. Indeed, if a variable is mentioned in both `x` and `y`, one should merge its two bindings, unless they conflict. In the latter case, one should produce an error and give up. More formally, the purpose of the function `merge_bindings` is to take two lists of bindings and merge them together, producing a single consistent list, if it is possible, or an error, if there is an inconsistency found. All the variables that participated in the input bindings have to be present in the output bindings and vice versa. Each variable has to appear in exactly one output binding.

See the code listing for a simplistic implementation of `strToMyElement :: String -> MyElement`. With it, we can make the following definition:

```

nice_unify1 :: String -> String -> MyBindings
nice_unify1 x y = unify1 (strToMyElement x) (strToMyElement y)

```

and, using an interactive Haskell interpreter, such as HUGS, evaluate simple expressions of the form

```
nice_unify1 "7" "a"
```

instead of the more cumbersome

```
nice_unify1 (MyNum 7) (MyAtom "a")
```

see examples below for even hairier syntax, for extended versions of `unify`.

2 The project

Basic version: Implement the function `merge_bindings` above. Give enough examples to test it well. Make sure to have tests trigger all possible failures and all different kinds of successful return values for `unify`. Some examples (in the abbreviated notation, not showing **Maybe/Just**, see discussion below):

```
unify :: MyElement -> MyElement -> MyBindings
unify2lists :: MyElement -> MyElement -> MyBindings
```

```
unify2lists [1,2,3] [1,4,5] -- an error, as 2 does not match 4
unify2lists [1,2,3] [1,a,5] -- matches a to 2, but then yields an error, as 3 does not match 5
unify2lists [1,2,3] [a,b,c] -- should produce [(a,1),(b,2),(c,3)], in any order
unify2lists [1,b,3] [a,2,c] -- also should produce [(a,1),(b,2),(c,3)]
unify2lists [1,2,3] [b,b,c] -- should produce an error,
-- since variable "b" is associated with two different values.
```

```
unify2lists [1,a,3] [a,b,c] --- should produce [(a,b,1),(c,3)], since b is bound to a and a to 1
unify2lists [a,b,c] [b,c,a] --- [(a,b,c),Nothing]
```

For an illustration, the second to last input and output written in full would be

```
unify [MyNum 1, MyAtom "a", MyNum 3]
      [MyAtom "a", MyAtom "b", MyAtom "c"]
-- [(["a","b"], Just (MyAtom 1)), (["c"], Just (MyNum3))]
```

A more extensive version Implement the following version of the unification algorithm in Haskell: The primitive objects are numbers and variable names (strings), as above. The general patterns that you have to handle are more general, namely recursive lists. Since Haskell is a strongly-typed language, Haskell lists are homogeneous, thus `[1,"a",10]` is an illegal Haskell object. To simulate heterogeneous lists, we modify our definition of `MyElement` given above as follows:

```
data MyElement = MyNum Int | MyAtom String | MyNested [MyElement]
```

For example, here are several objects of our new type:

```
MyNested [] :: MyElement -- corresponds to []
MyNested [(MyNum 7), (MyAtom "a")] :: MyElement -- [7,a]
MyNested [(MyNested [MyNum 7]), (MyAtom "a")] :: MyElement -- [[7],a]
```

From this point on, I will use the (syntactically illegal in Haskell, but legal in languages with inhomogeneous lists, such as LISP) form, as in the comments above.

In addition, extend the definition of `strToMyElement` given in the listings to handle recursive nested lists. It's not as hard as it looks. I will post some hints on how to do it later. (If you are not scared of monsters lurking in closets and under the bed, look at the definition of `readList` in the Prelude. It's a bit too general for us, but figuring out what it does is... ahem... a good learning experience. One hint to understanding it is that the string-processing functions that "scan" a string (for example, that skip until the next comma character) return a *pair*—the string they skipped over and the string that remains.) This extended definition will allow you to write `"[[7],a]"` instead of the very cumbersome

```
MyNested [(MyNested [MyNum 7]), (MyAtom "a")]
```

above.

Here are a few examples of what this more general unify function is supposed to accomplish (in the abbreviated notation, not showing **Maybe/Just**):

```
unify :: MyElement -> MyBindings
```

```
unify [1,[4],3] [1,4,5] -- an error, as [4] does not match 4
unify [1,2,[3]] [1,a,[3,1]] -- matches a to 2, but then yields an error, as [3] does not match [3,1]
unify [1,[2],3] [a,[b],c] -- should produce [(a,1),(b,2),(c,3)], in any order
unify [1,[b],[[3]]] [a,[2],[[c]]] -- also should produce [(a,1),(b,2),(c,3)]
unify [1,[1],3] [b,b,c] -- should produce an error,
  -- since variable "b" is associated with two different values.
```

In this version, you may assume that the following does *NOT* happen: a variable is bound to a list which has a variable (this or another one) somewhere in it, on top or on some other level, as in

```
unify a [1,3,b]
```

where your implementation does not have to check for this (it would be helpful if it did, but it does not have to), but it may assume this never happens.

More challenging version: The above restriction is removed. When binding a variable to a list, another variable name might appear in it, possibly any number of variables, and the matching should continue recursively. For example, `unify [1 [2 3] [3 a]] [a [2 b] c]` should produce `[(a,1),(b,3),(c,[3 1])]`. A really challenging question is to figure out what to do when arbitrary cyclic dependencies are present—how to detect them and what a meaningful output could be, as in `unify [a,[b]] [b,[a]]`. A slightly less challenging question is to determine what restriction you need to put on the input so that these problems do not arise and have an implementation that can handle only such inputs.

Some relevant files: The complete example without nested lists can be found in `unify1.hs`. The skeleton of the second version is in `unify2.hs`. Pretty-printed version of the code is in `unify-haskell.pdf`.