Computer Science 320 (Fall 2016)
Concepts of Programming Languages

Problem Set 5: Unification

Out: Thursday, October 27, 2016
Due: Saturday, November 12, 2016, by 11:59 pm

In this assignment, you will construct a constraint solver and a few unification algorithms. You will implement four modules: `Unify`, `Equation`, `Tree`, and `Natural`. You should submit each in its own `*.hs` or `*.lhs` file. **File names are case sensitive.**

Note: All function definitions at top level must have explicit type annotations. Unless otherwise specified, your solutions may utilize functions from the standard prelude as well as material from the lecture notes and textbook. You may **not** use any other materials without explicit citation.

Problem 1. (20 pts)

Unification is defined in terms of *substitutions*. We say a substitution `s` *unifies* two values `v1` and `v2` if `subst s v1 == subst s v2`, where `==` is derived equality on the type of those values and `subst s v` is the application of a substitution `s` on a value `v`. You will define a representation for substitutions in the `Unify` module.

(a) Substitutions will be represented as a list of pairs. Each pair will have a variable (represented as a `String`) and a value of type `a` representing the value for which that variable can be substituted. Define a polymorphic data type `Subst a` with a single constructor `S::[(String, a)] -> Subst a`. For debugging purposes, you may want to derive `Show` for this data type.

(b) Define a value `emp::Subst a` that represents the empty substitution, and a function `sub::String -> a -> Subst a` for constructing a substitution for only a single variable.

(c) Define a function `get::String -> Subst a -> Maybe a` that takes a variable name and a substitution, and returns the value for which that variable must be substituted. If the substitution is not defined on the variable, the function should return `Nothing`.

(d) Define a type class `Substitutable` that specifies that a function `subst::Subst a -> a -> a` must be defined for any type `a` inside this class.

(e) Define a type class `Unifiable` that specifies that a function `unify::a -> a -> Maybe (Subst a)` must be defined for any type `a` inside this class. A type `a` can only be in the `Unifiable` class if it is already in the `Eq` and `Substitutable` classes, and this should be represented in your definition.

(f) Modify your definition of the `Unify` module so that only the values `emp`, `sub`, `get`, the type constructor `Subst`, the type classes `Substitutable` and `Unifiable`, and the functions `subst` and `unify` are exported from the module.

Problem 2. (15 pts)

In the module `Natural`, you will be working with the following data type:

```
data Natural = Zero | Succ Natural | Var String
```

(a) Add an `instance` declaration so that `Natural` is in the `Substitutable` class.

(b) Add an `instance` declaration so that `Natural` is in the `Unifiable` class. Recall that `unify` must return one of the set of simplest substitutions under which the two arguments to `unify` are equivalent under derived equality. If the two arguments are syntactically equivalent, `unify` should return the empty substitution.

In the module `Tree`, you will be working with the following data type:

```
data Tree = Leaf | Node Tree Tree | Var String
```

(c) Add an `instance` declaration so that `Tree` is in the `Substitutable` class.

It is not yet possible to define `unify` on values of type `Tree`, as unifying two subtrees produces two potentially contradictory substitutions.

Problem 3. (40 pts)

In order to combine multiple substitutions, it is necessary to resolve conflicts in which both substitutions map the same variable to two different values of type `a`.

(a) Define a function `unresolved::[(String,a)] -> Maybe (String, a, a)` that takes a list of pairs of variable names and values as its only argument, and behaves in the following way. If the input list contains no two pairs `(x,v)` and `(y,v')` such that `x` = `y` (note that the values `v` and `v'` may or may not be different), then there is no conflict in the list, so the function should return `Nothing`. However, if there exist at least two such elements `(x,v)` and `(y,v')`, it should return a tuple containing the conflicting variable, and the two values `v` and `v'`.

(b) Define a function `resolve::Unifiable a => Subst a -> Maybe (Subst a)` that takes a substitution as an argument. If the substitution has no unresolved conflicts, it should simply return it. Otherwise, it should resolve the conflict if possible (by using `unify`), and call itself recursively to resolve any remaining conflicts. Note that resolving a conflict may introduce new conflicts, as the new substitution returned by `unify` must be appended back onto the original substitution.

(Hint: If the substitution's list does have a conflict, it can take two of the conflicting pairs and remove *only one* of these conflicting pairs from the list (it may be easier to remove both, and insert one of them back into the list). It can then unify the values contained in these two pairs. If this unification fails, the substitution cannot be resolved, and `resolve` should return `Nothing`. Otherwise, this unification will produce a new substitution. This new substitution's list should then be appended onto the list from which one value was already removed. This result should then be resolved again, as it is possible that new conflicts were introduced.)

(c) Define a function `cmb::Unifiable a => Maybe (Subst a) -> Maybe (Subst a) -> Maybe (Subst a)` that takes two substitutions (one or both of which might be `Nothing`) as arguments. If neither one is `Nothing`, it should combine them into a single substitution, resolve it if it contains any conflicts, and return it. If even one of the arguments is `Nothing`, `cmb` should return `Nothing`. Modify your declaration of the `Unify` module so that `cmb` is exported from the module.

Problem 4. (25 pts)

In the module `Equation`, equations on a type `a` are represented in the following way:

```
data Equation a = a `Equals` a
```

(a) Define a function `solveEqn::(Unifiable a) => Equation a -> Maybe (Subst a)` that returns the substitution that solves an equation, if possible.

(b) Define a function `solveSystem::(Unifiable a) => [Equation a] -> Maybe (Subst a)` that takes a list of equations as an argument, and returns the substitution that solves the system of equations represented by it, if possible.

It is also now possible to define `unify` on values of type `Tree`.

(c) In the `Tree` module, add an `instance` declaration so that `Tree` is in the `Unifiable` class.

(d) Solve the following equations or systems of equations. You are allowed to share your solutions to these equations, but you may not share any of your Haskell code. For each equation, define a value that holds its solution (for example, `s0` can be the solution for `e0`, and so on).

```
e0 = Node (Node (Node (Var "x") (Var "y")) (Node (Var "y") (Var "x"))) (Var "z")
         `Equals`
     Node (Node (Node Leaf (Var "z")) (Node Leaf (Var "y"))) (Var "x")

e1 = let f b 0 = b
         f b n = Node (f b (n-1)) (f b (n-1))
     in f (Var "x") 10 `Equals` f Leaf 13

e2 = [ (Var "z") `Equals` Leaf
     , Node (Var "y") Leaf `Equals` Node  Leaf (Var "x")
     , (Var "x") `Equals` Node (Var "z") (Var "z")
     ]
```