Computer Science 320 (Fall, 2009)
Concepts of Programming Languages

Assignment 5: Unification

Out: Tuesday, October 27, 2009
Due: Friday, November 6, 2009

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. (5 pts)

A *substitution* is any mapping (in a mathematical sense) of variables to values. Substitutions can be represented in various ways. We will represent a substitution by a list of pairs. In each pair in the list, the first component specifies a variable name as a string and the second component specifies the value to which that variable is mapped. For example, the substitution

```
[("x", 1), ("y", 2)]
```

indicates that all instances of `"x"` map to `1` and all instances of `"y"` map to `2`. To *apply* a substitution is to replace all variables in a data type instance (i.e. a value) with their corresponding value in the substitution. You should introduce into your `Unify` module the following type synonym declaration:

```
type Subst a = [(String, a)]
```

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) 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.

(b) 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`.

Problem 2. (15 pts)

Because every data type has different constructors, each data type requires its own specific definition of a substitution function. However, we may want to write polymorphic code that uses substitutions and performs unification without explicitly referencing a particular data type. In order to accomplish this, we define some new type classes.

(a) Define a type class `Substitutable` that specifies that two functions, `subst::Subst a -> a -> a` and `vars::a -> [String]`, must be defined for any type `a` inside this class. The `vars` function is meant to return a list of all the variables in a value of type `a`.

(b) Within the `Substitutable` class declaration, define a function `solved::Subst a -> Bool`. This function should return `True` only if the second component of every pair in its input list has *no* variables (as determined by the `vars` function). Otherwise, it should return `False`. To avoid generating an `Eq a` constraint, you may want to use the `null::[a] -> Bool` function from the Haskell libraries.

(c) Within the `Substitutable` class declaration, define a function `reduce::Subst a -> Subst a`. This function should take as input a substitution s and collect all the pairs in s whose second component contains no variables. Call this new list of pairs s'. It should then apply s' to *all* the second components in s, returning the result.

Problem 3. (10 pts)

(a) 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.

(b) Within the `Unifiable` class declaration, define a function `combine::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 using concatenation. If even one of the arguments is `Nothing`, `combine` should return `Nothing`.

(c) 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`, `vars`, `unify`, and `combine` are exported from the module.

Problem 4. (20 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. The `unify` function must return a substitution under which the two arguments to `unify` are equivalent. 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.

(d) Add an `instance` declaration so that `Tree` is in the `Unifiable` class.

Problem 5. (30 pts)

Within the `Unifiable` class declaration, define a function `resolve::Maybe (Subst a) -> Maybe (Subst a)` that takes a substitution or `Nothing` as an argument. If its input is `Nothing`, it can also return `Nothing`. If its input is a substitution, however, `resolve` should do the following.

(1) Repeatedly apply `reduce` to the input substitution until no more changes occur when `reduce` is applied. You may want to write a helper function to accomplish this. You may also want to use the `nub::Eq a => [a] -> [a]` function from the Haskell libraries to make sure your substitutions do not contain duplicated entries after each reduction.

(2) Once the substitution is stable with respect to `reduce`, check if it has any conflicts. A conflict occurs if the substitution contains two pairs `(x,u)` and `(y,v)` such that $\texttt{x} = \texttt{y}$ but $\texttt{u} \neq \texttt{v}$. If a conflict exists, `resolve` should return `Nothing`.

(3) Check if the substitution is solved by using the `solved` function. If it is solved, it should be returned; otherwise, `resolve` should return `Nothing`.

You will need to add `resolve` to the list of functions that you export from the `Unify` module.

Problem 6. (20 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. Be careful: you cannot implement this correctly if you use your solution from part (a) directly, but the two functions in parts (a) and (b) may share a helper function.

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

(c) 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 in the `Tree` module 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")
     ]
```