Assignment 4

COMP 302 Programming Languages and Paradigms
Prakash Panangaden

Due Date: 21st March 2017

This assignment is essentially one long programming assignment. I have broken it into two parts. Question 3 is not to be turned in, it is to give you practice for the final. Q4 is for your spiritual growth.

In this assignment we will implement the unification algorithm for terms. Unification is the heart of polymorphic type inference. We will, however, look at unification by itself without examining its role in type inference. Unification is one of the central operations in type-reconstruction algorithms, theorem proving and logic programming systems and is therefore worth understanding for its own sake.

A *term* is an expression constructed from *function symbols*, *variables* and *constants*. For example, if we have a function symbol f we could construct as an example, the term $f(3, x, 2)$. Every function symbol takes a fixed number of arguments called its *arity*. This *never* changes. In the previous example, f had arity 3. We do not interpret the function symbols (that is why we call them "function symbols" rather than functions). What makes a variable different from a constant is the fact that we can substitute a term for a variable thus obtaining a new term. For example, we have a term $h(y)$ (h is a function sybbol of arity 1 and y is a variable) and a term $f(3, x, 2)$; if we substitute the first term for x in the second term we get $f(3, h(y), 2)$. Please note carefully the correct usage of these words we substitute a term t_1 for a variable x in another term t_2 to obtain the term t_3 as the result.

Now a simple variable by itself and a constant by itself is also a term. These are the base cases of an inductive definition of terms. If f is a function symbol of arity n and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term[1].

A single substitution is just a pair of the form (x, t) where x is a variable and t is a term. More generally, a substitution is a *list* of such pairs. In a substitution list, each variable only appears once as the left-hand side member of a pair; otherwise we would be giving conflicting substitutions for the same variable. We use the letter s to denote a substitution. To *apply*

[1]If you cannot make sense of this please see me or a TA or a mentor. Asking someone on Facebook means that you expect the explanation to be something that fits in a short comment. It does not! It requires a detailed explanation.

a substitution s *to a term* t means that we carry out all the replacements indicated by s to the variables in t. It may happen that some of the variables in t are not mentioned in s, in which case they are left unchanged, and also it may be the case that some of the variables mentioned in s do not occur in t, in which case we just ignore them.

Here is an example of these ideas. Our term is $f(x, y, h(x))$ and the substitution is $[(x, h(z)); (y, 3)]$. The term that results from *applying* the substitution is $f(h(z), 3, h(h(z)))$. We **never** substitute function symbols, only variables.

Now we are ready to discuss unification. Consider, for example, the following two terms: $f(x, y, h(x))$ and $f(g(z), h(x), y)$. They do **not** look identical as written. They have, the same outermost symbol, f in this case. This is called the *head* symbol. There is a chance that if we find the right substitutions for the variables occurring *inside* these terms the result of applying the substitution will produce identical terms. They are said to have become *unified*. A glance at these two terms shows that the substitution $[(x, g(z)); (y, h(x))]$ will unify them. Both terms become identical once you apply the substitution shown[2]. In this assignment we will code the algorithm that finds the *most general unifier* or says that unification is not possible.

What does "most general" mean? It means that we retain the maximum flexibility in terms of constraining the variables. We do not create substitutions unless we are forced to. In the example above, we could have substituted $g(3)$ for x but this is unnecessarily restrictive. It is a theorem that of all the ways of unifying two terms there is a *unique* one such that all the others are obtained from this one by applying further substitutions. This is the most general unifier.

How can unification fail? For example, we cannot unify $f(x)$ and $g(y, z)$. The head symbols are different and nothing you do inside the term can change that. You cannot unify two different constants or a complex term and a constant. Unification can also fail if you substitute a variable with a term that contains the variable. Thus, $(x, f(x))$ is an example of an illegal substitution.

We represent terms in F# with the following type definition:

```
type id = string

type term =
  | Var of id
  | Const of int
  | Term of id * term list
```

We are using constructor functions[3] to tag the different kinds of terms. Note how the type definition mimics the inductive definition that I gave above. Here is an example term

[2]Check it for yourself if it is not evident.

[3]This is **not** casting!

```
let t1 = Term("f",[Var "x";Var "y"; Term("h",[Var "x"])])
```

We have the following type for substitutions; note we don't bother to put the `Var` constructor in the left-hand side.

```
type substitution = (id * term) list
```

Important: we will make sure that in our substitution lists we *never* have a variable that appears in the left-hand side of a pair occur earlier inside some term that appears in the right-hand side of a pair. The type definition does not enforce this so we will ensure that we adhere to this rule as we build up substitutions. Here is a substitution that violates this rule

```
[("x", Term("f",[Const 1; Term("k", [Var "u"; Var "y"])])); ("y", Const 3)]
```

Question 1[40 points]

In this question you will implement some of the auxiliary functions needed for unification. First of all note that we do not allow a substitution where a variable is replaced by a term containing it. So we need to check if a variable occurs in a term. This is called the "occur check." Before we invoke the occur check, we strip off the `Var` constructor so we need a function of the following type.

```
val occurs : x:id -> t:term -> bool
```

Recall that a substitution is defined as follows.

```
type substitution = (id * term) list
```

Now we want a function that performs a replacement of a variable with a term. This should have the following type.

```
val subst : s:term -> x:id -> t:term -> term
```

This replaces all occurrences of the variable `x` with the term `s` in the term `t`.

The above function just does one replacement. A substituion is a whole list of these, so we need another function called `apply`

```
val apply : s:substitution -> t:term -> term
```

Here is an example of a substitution

```
  [("x", Term ("k",[Term ("h",[Var "z"])])); ("y", Term ("h",[Var "z"]))]
```

When we apply this to the term

```
Term ("f",[Var "x"; Term ("h",[Var "z"]); Var "x"])
```

we get the result

```
  Term
```

```
    ("f",
     [Term ("k",[Term ("h",[Var "z"])]); Term ("h",[Var "z"]);
      Term ("k",[Term ("h",[Var "z"])])])
```

We will always apply substitutions from right to left. The ideal solution is a one-liner using `List.foldBack`.

To summarize: for this question you have to implement

- `ocurs`
- `subst` and
- `apply`.

Question 2[60 points]

In this question you are asked to implement a function `unify` which checks whether two terms are unifiable and produces the unifier if there is one. It must return the most general unifier or fail with an appropriate error message. The type is

```
val unify : s:term -> t:term -> substitution
val unify_list : s:(term * term) list -> substitution
```

Here the two functions are defined by mutual recursion.

Here are the messages I used when unification failed for one reason or another:

```
 "not unifiable: clashing constants"
 "not unifiable: term constant clash"
 "not unifiable: head symbol conflict"
"not unifiable: circularity"
```

There is no need to do fancy exception handling, a simple `failwith` is enough for each case.

I have written a template file which you can find on the course web site. There are also some more examples shown there.

Question 3[0 points] (Start practicing for the final)
Informally *derive* the type for the apply-list function defined below. This function takes a list of functions and produces a single function which is the composite of all of them. If the list is empty it returns the identity function.

The code is shown below.

```
let rec apply_list l =
  match l with
  | [] -> (fun x -> x)
  | f::fs -> (fun x -> apply_list(fs)(f x))
```

Question 4[0 points] (Spiritual growth) Show that there is a unique most general unifier. What happens if we try to solve the reverse problem. We are given a bunch of terms $t_1, \ldots, t_n$ and we want to find a term t such that all the terms t_i are obtained from t by substitution. This is called generalization. Is there a unique most special generalizer?