CS450: Progrmg Languages & Compilers
Summer 2015

Homework 6

University of Illinois, Urbana-Champaign,

Due: July 20, 2015, 5.00 pm. Submit PDF + code files on Moodle.

1 Theoretical Questions (30 points)

2 Objectives and Background

The purpose of this HW is to test your understanding of

- translating informal descriptions of language into regular expressions
- test your understanding of regular expressions and distinguish between regular grammars and other grammars.

Another purpose of HWs is to provide you with experience answering non-programming written questions of the kind you may experience on the final.

Problem 1: (10 points)

For the following two problems give a regular expression that represents the strings in the described language. In both cases, assume that our alphabet is the set $\{a, b, c\}$.

1. The language where a occurs in every third position.
2. The language where each string contains exactly 3 c's.

Problem 2: (20 points)

Decide whether the following (informally described) languages are regular. For each regular language, write a **regular expression** and a **regular grammar** of the language. For each language that is not regular, make an argument of why the language is not regular.

1. $L_1 = \{w | w$ starts with symbol 0 and contains the symbol 1 at least once$\}$ where $\Sigma = \{0, 1\}$
2. $L_2 = \{w | w$ contains an equal number of 0s and 1s$\}$ where $\Sigma = \{0, 1\}$
3. $L_3 = \{w |$the length of w is odd$\}$ where $\Sigma = \{a, b\}$
4. $L_4 = \{w | w$ does *not* contain symbol a immediately followed by symbol $b\}$ where $\Sigma = \{a, b, c\}$
5. $L_5 = \{w |$the length of w is a perfect cube$\}$ where $\Sigma = \{a, b, c\}$

3 Machine Problems (90 points)

Objectives

Your objective for this assignment is to understand the details of the basic algorithm for first order unification.

4 What to submit

You have to submit your solutions in mp6.ml. Solutions to problems 0 and 5 are optional, and these won't be graded.

5 Preliminaries

In HW 5, you have implemented the first part of the type inferencer for the MicroML language. In this HW, you will implement the second step of the inferencer: the unification algorithm *unify* that solves constraints generated by the inferencer. The unifier in HW 5 was a black box that gave you the solution when fed with the constraints generated by your implementation of the inferencer.

It is recommended that before or in tandem with completing this assignment, you go over lecture notes covering type inference and unification to have a good understanding of how types are inferred.

6 Datatypes for Type Inference

Below is some code for your use in the `mp6common` module. This module includes the following data types to represent the types of MicroML:

```
type typeVar = int

type monoTy = TyVar of typeVar | TyConst of (string * monoTy list)
```

You can use `string_of_monoTy` in `mp6common` to convert your types into a readable concrete syntax for types.

7 Substitutions

A substitution function returns a replacement type for a given type variable. Here, we use the the type `int` for type variables, which we give the synonym `typeVar`. Our substitutions will have the type `(typeVar * monoTy) list`. The first component of a pair is the index (or "name") of a type variable. The second is the type that should be substituted for that type variable. If an entry for a type variable index does not exist in the list, the identity substitution should be assumed for that type variable (i.e. the variable is substituted with itself). For instance, the substitution:

$$\varphi(\tau_i) = \begin{cases} \texttt{bool} \rightarrow \tau_2 & \text{if } i = 5 \\ \tau_i & \text{otherwise} \end{cases}$$

would be implemented as

```
# let phi = [(5, mk_fun_ty bool_ty (TyVar(2)))];;
val phi : (int * monoTy) list =
  [(5, TyConst ("->", [TyConst ("bool", []); TyVar 2]))]
```

Throughout this MP you may assume that substitutions we work on are always well-structured: there are no two pairs in a substitution list with the same index.

Given a substitution list, we can convert it to a function that takes a `typeVar` and returns a `monoTy`.

```
# let subst_fun s = ...
val subst_fun : (typeVar * monoTy) list -> typeVar -> monoTy = <fun>
# let subst = subst_fun phi;;
val subst : typeVar -> monoTy = <fun>
# subst 1;;
- : monoTy = TyVar 1
# subst 5;;
- : monoTy = TyConst ("->", [TyConst ("bool", []); TyVar 2])
```

We can also *lift* a substitution to operate on types. So a substitution φ, when lifted, replaces all the type variables occurring in its input type with the corresponding types.

```
# let rec monoTy_lift_subst s = ...
val monoTy_lift_subst : (typeVar * monoTy) list -> monoTy -> monoTy = <fun>
# let lifted_sub = monoTy_lift_subst phi;;
val lifted_sub : monoTy -> monoTy = <fun>
# lifted_sub (TyConst ("->", [TyVar 1; TyVar 5]));;
- : monoTy =
TyConst ("->", [TyVar 1; TyConst ("->", [TyConst ("bool", []); TyVar 2])])
```

You will need to implement the `subst_fun` and `monoTy_lift_subst` functions (see the Problems section).

8 Unification

The unification algorithm takes a set of pairs of types that are supposed to be equal. A system of constraints looks like the following set

$$\{(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\}$$

Each pair is called an *equation*. A (lifted) substitution φ *solves* an equation (s, t) if $\varphi(s) = \varphi(t)$. It solves a constraint set if $\varphi(s_i) = \varphi(t_i)$ for every (s_i, t_i) in the constraint set. The unification algorithm will return a substitution that solves the given constraint set (if a solution exists).

You will remember from lecture that the unification algorithm consists of four transformations. These transformations can be expressed in terms of how an action on the first element of the unification problem affects the remaining elements.

Given a constraint set C

1. If C is empty, return the identity substitution.
2. If C is not empty, pick an equation $(s, t) \in C$. Let C' be $C \setminus \{(s, t)\}$.

(a) **Delete rule:** If s and t are are equal, discard the pair, and unify C'.

(b) **Orient rule:** If t is a variable, and s is not, then discard (s, t), and unify $\{(t, s)\} \cup C'$.

(c) **Decompose rule:** If $s = \texttt{TyConst}(name, [s_1; \ldots; s_n])$ and $t = \texttt{TyConst}(name, [t_1; \ldots; t_n])$, then discard (s, t), and unify $C' \cup \bigcup_{i=1}^{n} \{(s_i, t_i)\}$.

(d) **Eliminate rule:** If s is a variable, and s does not occur in t, substitute s with t in C' to get C''. Let φ be the substitution resulting from unifying C''. Return φ updated with $s \mapsto \varphi(t)$.

(e) If none of the above cases apply, it is a unification error (your `unify` function should return the `None` option in this case).

In our system, function, integer, list, etc. types are the terms; `TyVar`s are the variables.

9 Problems

Problem 0: Preliminary Uncredited Problem (0 points)

Make sure that you understand the `monoTy` data type. You should be comfortable with how to represent a type using `monoTy`. HW5 should have given you enough practice of this. If you still do not feel fluent enough, do the exercise below. This exercise will not be graded; it is intended to warm you up.

In each item below, define a function `asMonoTy`X`:unit -> monoTy` that returns the `monoTy` representation of the given type. In these types, $\alpha, \beta, \gamma, \delta, \ldots$ are type variables.

- `bool` $\rightarrow$ `int list`

```
#  let asMonoTy1 () = ...
val asMonoTy1 : unit -> monoTy = <fun>
# string_of_monoTy(asMonoTy1());;
- : string = "bool -> int list"
```

- $\alpha \rightarrow \beta \rightarrow \delta \rightarrow \gamma$

```
# let asMonoTy2 () = ...
val asMonoTy2 : unit -> monoTy = <fun>
# string_of_monoTy(asMonoTy2());;
- : string = "'d -> 'c -> 'b -> 'a"
```

- $\alpha \rightarrow (\beta$ `*` `int)list`

```
# let asMonoTy3 () = ...
val asMonoTy3 : unit -> monoTy = <fun>
# string_of_monoTy(asMonoTy3());;
- : string = "'f -> ('e * int) list"
```

- (`string` $*$ (β `list` $\rightarrow \alpha$))

```
#  let asMonoTy4 () = ...
val asMonoTy4 : unit -> monoTy = <fun>
# string_of_monoTy(asMonoTy4());;
- : string = "string * 'h list -> 'g"
```

Problem 1: (8 points)

Implement the `subst_fun` function as described in Section 7.

```
# let subst_fun s = ...
val subst_fun : (typeVar * monoTy) list -> typeVar -> monoTy = <fun>
# let subst = subst_fun [(5, mk_fun_ty bool_ty (TyVar(2)))];;
val subst : typeVar -> monoTy = <fun>
# subst 1;;
- : monoTy = TyVar 1
# subst 5;;
- : monoTy = TyConst ("->", [TyConst ("bool", []); TyVar 2])
```

Problem 2: (8 points)

Implement the `monoTy_lift_subst` function as described in Section 7.

```
# let rec monoTy_lift_subst s = ...
val monoTy_lift_subst : (typeVar * monoTy) list -> monoTy -> monoTy = <fun>
# monoTy_lift_subst [(5, mk_fun_ty bool_ty (TyVar(2)))]
            (TyConst ("->", [TyVar 1; TyVar 5]));;
- : monoTy =
TyConst ("->", [TyVar 1; TyConst ("->", [TyConst ("bool", []); TyVar 2])])
```

Problem 3: (10 points)

Write a function `occurs : typeVar -> monoTy -> bool`. The first argument is the integer component of a `TyVar`. The second is a target expression. The output indicates whether the variable occurs within the target.

```
# let rec occurs v ty = ...
val occurs : typeVar -> monoTy -> bool = <fun>
# occurs 0 (TyConst ("->", [TyVar 0; TyVar 0]));;
- : bool = true
# occurs 0 (TyConst ("->", [TyVar 1; TyVar 2]));;
- : bool = false
```

Problem 4: (64 points)

Now you are ready to write the unification function. We will represent constraint sets simply by lists. If there exists a solution, your function should return `Some` of that substitution. Otherwise it should return `None`. Here's a sample run.

```
# let rec unify eqlst = ...
val unify : (monoTy * monoTy) list -> substitution option = <fun>
# let Some(subst) =
```

```
   unify [(TyVar 0,
           TyConst ("list",
             [TyConst ("int", [])]));
          (TyConst ("->", [TyVar 0; TyVar 0]),
           TyConst ("->", [TyVar 0; TyVar 1]))];;
... (* Warning message suppressed *)
val subst : substitution =
  [(0, TyConst ("list", [TyConst ("int", [])]));
   (1, TyConst ("list", [TyConst ("int", [])]))]
# subst_fun subst 0;;
- : monoTy = TyConst ("list", [TyConst ("int", [])])
# subst_fun subst 1;;
- : monoTy = TyConst ("list", [TyConst ("int", [])])
# subst_fun subst 2;;
- : monoTy = TyVar 2
```

Hint: You will find the functions you implemented in Problems 2, 3, 4 very useful in some rules.

Point distribution: Delete is 6 pts, Orient is 6 pts, Decompose is 16 pts, Eliminate is 36 pts. This distibution is approximate. Correctness of one part impacts the functioning of other parts. Machine grader will not be able to detect this, however the human grader will fix propagating errors.

Problem 5: Additional Uncredited Problem (0 points)

Two types τ_1 and τ_2 are equivalent if there exist two substitutions φ_1, φ_2 such that $\varphi_1(\tau_1) = \tau_2$ and $\varphi_2(\tau_2) = \tau_1$. Write a function `equiv_types : monoTy -> monoTy -> bool` to indicate whether the two input type expressions are equivalent.

Hint: find τ_3 such that τ_1 is equivalent to τ_3 and τ_2 is also equivalent to τ_3 by reducing τ_1 and τ_2 to a canonical form.

```
# let equiv_types ty1 ty2 = ...
val equiv_types : monoTy -> monoTy -> bool = <fun>
# equiv_types
    (TyConst ("->", [TyVar 4; TyConst ("->", [TyVar 3; TyVar 4])]))
    (TyConst ("->", [TyVar 3; TyConst ("->", [TyVar 4; TyVar 3])]));;
- : bool = true
# equiv_types
    (TyConst ("->", [TyVar 4; TyConst ("->", [TyVar 3; TyVar 4])]))
    (TyConst ("->", [TyVar 4; TyConst ("->", [TyVar 3; TyVar 2])]));;
- : bool = false
```