Some notes on Type Inference for $\lambda_{\rightarrow}$ (Take Home Exam) DUE Dec. 15 at noon

Jim Lipton
Dept. of Mathematics and Computer Science
Wesleyan University
COMP 321 fall 2022

December 8, 2022

1 Introduction

In your take home you are being asked to write an ocaml function `infer_type` that takes

- a context `ctx` such as `[(x,A),(y,B),(z,C)]`
- an untyped lambda-term such as $\lambda x.\lambda y.xy$ as an **unparsed string** `"\\x.\\y. x y"`.

and returns

- *either* the exception FAIL
- *or* a *judgment* in the form of a string consisting of the context, followed by the symbols `|-`, followed by a *typed* term, followed by the colon (:), followed by a type.

For example, if the input is the empty context `[ ]` and the term `\\x.\\y.x y`, the output would be the following string

```
[ ] |- \\x:(A5->A4).(\\y:A5.(x y)) : ((A5->A4)->(A5->A4))          (*)
```

Note that the initial context is usually empty but there is no need to assume this.

The intermediate stages

The untyped lambda term (represented as a human readable string) `"\\x.\\y. x y"` input to `infer_type` will be parsed into a term in the form e.g.
`TmAbs("x",TmAbs("y",TmApp(TmVar("x"),TmVar("y"))))`.

Then the term is decorated, with dummy variables, and then a type-derivation is built with these variables, and a list of constraints produced at the end. They are unified to yield a pair (t, T) where t is the term

```
TyAbs("x",TyArr(TyVar("A5"),TyVar("A4")),TmAbs("y", TyVar("A5"), TyApp(TyVar("x"),TyVar("y"))))
```

representing the *typed* term $\lambda x : A5 \rightarrow A4, \lambda y : A5.xy$ and T is the *type*

```
TyArr(TyArr(TyVar("A5"),TyVar("A4")), TyArr(TyVar("A5"),TyVar("A4")))
```

representing the type $(A5 \rightarrow A4) \rightarrow (A5 \rightarrow A4)$.

Then this output and the context can be fed to a function that displays the answer as a string as shown above in (*).

The steps in type inference

There are a number of steps that have to be taken in order to get the result, which are outlined here.

Step 1: decorate the input term transform the untyped term $\lambda x.\lambda y.xy$ to a "dummy" typed term $\lambda x : A2.\lambda y : A1.xy$. Assign the whole term a placeholder type A for example, *or*, given the shape of the term, $A_1 \to A_2 \to A$, to save a few steps. The names $A1$ and $A2$ would have to be generated by some function like `make_fresh_var` which we defined in an earlier homework, which returns, each time it is invoked, a string in the form An where n is a natural number.

Step 2: generating constraints The three pieces of information you now have

- context: in this case the empty list
- decorated term: `TyAbs("x",Var("A2"),TyAbs("y",Var("A1"),TyApp(TyVar("x''),TyVar("y"))))`
- type: `TyVar("A")` or `TyArr(Var("A", TyArr(Var("A2"),TyVar("A"))))`

should be passed to a function

```
make_constraints: context -> tylam -> arrow_type -> eqn list
```

where `tylam` is the type of terms in the simply typed lambda calculus, and `arrow_type` is the type of type expressions, defined so that

```
make_constraints ctx t ty
```

returns a *list of type equations* $\mathcal{E}$ as follows, for example:

```
- : eqn list =
[Eq (Var "A2", Arr (Var "A5", Var "A4")); Eq (Var "A1", Var "A5");
 Eq (Var "A3", Arr (Var "A1", Var "A4"));
 Eq (Var "A", Arr (Var "A2", Var "A3"))]
```

representing the equations

```
[A2 = (A5->A4),A1 = A5,A3 = (A1->A4),A = (A2 -> A3)]
```

Step 3: Unifying the list of equations The list of equations $\mathcal{E}$ must then be *unified*, using the so-called unification algorithm. This means converting $\mathcal{E}$ to so-called *solved form* $\mathcal{E}'$ of the form $[X_1 = T_1, \ldots, X_n = T_n]$ which has the properties that

- $X_1, \ldots, X_n$ are distinct type variables (such as $A2$ or $A3$),
- $T_1, \ldots, T_n$ are types (such as e.g. $A5 \to A4$ or just $A2$),
- None of the X_i appear in any of the T_j,
- $\mathcal{E}'$ defines a *substitution*
$$\sigma = [X_1 \mapsto T_1, \ldots, X_n \mapsto T_n]$$
which satisfies the original list of equations.

Definition 1 *A substitution σ satisfies a list of equations*

$$[t_1 = u_1, ..., t_n = u_n]$$

if σt_i is **identical** *to σu_i for every i.*

In the case of the example list of equations defined above, the unification algorithm produces the following list of equations in solved form:

$$A2 = (A5 \to A4), A1 = A5, A3 = (A5 \to A4), A = ((A5 \to A4) \to (A5 \to A4))$$

corresponding to the substitution (sigma)

$$\sigma = [A2 \mapsto (A5 \to A4), A1 \mapsto A5, A3 \mapsto (A5 \to A4), A \mapsto ((A5 \to A4) \to (A5 \to A4))]$$

If the original list of type equations is inconsistent (cannot be solved because of an occurs check failure) the unification algorithm returns `FAIL`.

Step 4: Applying the resulting substitution to the decorated term and its type We now take the substitution σ produced by the unification algorithm and apply it to the types in the decorated term $\lambda x : A2.\lambda y : A1.xy$ and the placeholder type A obtaining:

- The typed term $\lambda x : (A5 \to A4).\lambda y : A5.(xy)$
- The type of the resulting term: $(A5 \to A4) \to (A5 \to A4)$.

Step 5: making it pretty Display your results in the form

```
[ ] |- \\x:(A5->A4).(\\y:A5.(x y)) : ((A5->A4)->(A5->A4))
```

2 The Unification Algorithm

Here is a description of the Unification algorithm.
Input: A set $\mathcal{E}$ of equations between type expressions.
Output: An equivalent set of equations in solved form, or FAIL.

repeat

select an arbitrary equation $S = T \in \mathcal{E}$;
case $S = T$ of:

1. $T = T \Rightarrow$ remove the equation from $\mathcal{E}$;
2. $(S_1 \to S_2) = (T_1 \to T_2) \Rightarrow$ replace the equation by $S_1 = T_1$, $S_2 = T_2$;
3. $T = X$ where X is a type variable and T is not a type variable $\Rightarrow$ replace this equation by $X = T$
4. $X = T$ where T is not identical to X and X occurs more than once in $\mathcal{E}$ $\Rightarrow$
   - **5a. if** the variable X occurs in T then halt with **FAIL**
   - **5b. else** apply the substitution $[X \mapsto T]$ to both sides of every equation in $\mathcal{E}$, i.e. replace all other occurrences of X in $\mathcal{E}$ by T, and keep the equation $X = T$ in $\mathcal{E}$.

Compose the resulting list of constraints with the substitution $[X \mapsto T]$.

If X does not occur more than once in $\mathcal{E}$, just leave the equation $X = T$ in $\mathcal{E}$

until no more action is possible on any equation in $\mathcal{E}$
Halt with $\mathcal{E}$

Some Data Types

Here are some `ocaml` data type definitions for the three kinds of terms you need to consider:

- Untyped lambda terms:

```
type term =
   TmVar of string
  |TmApp of (term * term)
  |TmAbs of (string * term)
```

- Type expressions:

```
type arrow_type = Bool|Var of string|Arr of (arrow_type*arrow_type)
```

- Typed lambda terms:

```
type tylam = TyVar of string|TyApp of (tylam*tylam) |
    TyAbs of (string*arrow_type*tylam)
```

- Substitutions: `type subst =  (arrow_type*arrow_type) list`
- Equations: `type eqn = Eq of (arrow_type*arrow_type)`
- Constraint sets are just items of type `eqn list`
- In the attached template `type_inf_skel2.ml`, I did not bother to define contexts as a data type. They are just lists of `(string,arrow_type)` pairs

What's missing, what's there?

I have added a whole bunch of utilities, and a the most important workhorse function `make_constraints`, but left some key functions to be defined, or completed.

Some included functions, calling helpers not shown:

```
(* compose: subst -> subst -> subst *)
let rec compose sub1 sub2 =
  let dom = domain sub1 in
  let sub1_sub2_applied = apply_to_range sub1 sub2 in
  let clean_sub2 = remove dom sub2 in
  let s12 = sub1_sub2_applied@clean_sub2
  in
    remove_ids s12

(* USE THE INFIX DOLLAR SIGN TO compose substitutions, instead of the
```

```
 * function COMPOSE
 *)
let ($) x y = compose x y;;

(* occurs: CHECKS TO SEE if an equation of the form
 * A =  arrow_type
 * has an occurrence of A on the right, which will cause FAILURE in
 * the unification algorithm. *)

let rec occurs x t =
  match (x,t) with
    |(Var z, Var u) -> z=u
    |(Var z, Arr(t1,t2)) -> occurs x t1 || occurs x t2
    |_ -> false
```

Where it says **FINISH** you must complete the code.

```
(* unify a list of equations: you must FINISH two of the cases *)
let rec unify lst =
  match lst with
      [] -> []
  | (Eq(x,y)::rest) when x=y -> (* you FINISH this. Just skip this
                                 * identity equation *)
  | (Eq(Var(x),t)::rest) when occurs (Var(x)) (t) -> raise FAIL
  | (Eq(Var(x),t) :: rest) -> (* FINISH: x does NOT OCCUR in t so COMPOSE
       * the subst [(Var(x),t)] with UNIFY the result of applying  [(Var(x),t)]
                                 * to the rest of the list
                                *)

  | (Eq(t,Var(x))::rest) -> unify (Eq(Var(x), t)::rest)
  | (Eq (Arr(a1,a2),Arr(b1,b2))::rest)
    -> (unify [Eq(a1,b1);Eq(a2,b2)]) $ (unify rest)
  |_ -> raise FAIL;;


(* decorate:term -> tylam
   the first step in type inference.
   FINISH
   USE make_new_var() *)
let rec decorate tm =
  x:=0;
  match tm with
    |(TmVar y) ->
    |(TmApp (t1,t2)) ->
    |(TmAbs (y,t)) ->


(* THE BIG DADDY: the top level type inference function *)

(* FINISH!  what do do:
 * let type_inf ctx str = [[followed by 5 let terms]]
 * 1: parse the string. Creates an untyped lambda term t1
 * 2: decorate the result. Creates a typed lambda term t2
 * 3: let u  =  make_constraints [] t2 (Var("A")) in
 * 4: unify u. Call the resulting subst u'
 * 5: Apply this subst to t2 (so the types are now correct) to get t'
 * 6: show your work! use show_judgment using ctx and t' and at the
 *      end the actual type of the whole term using typof
 *)

let type_inf ctx str =  (* FINISH IT *)
```

I included the following tests:

```
(* tests *)
print_string "\n****************** TESTS ************\n";;
print_string "(* A QUICK TEST *)";;
print_string "qtest 1 \n";;
let t1 = (parse "\\x.\\y.x y") in
    let t2 = decorate t1 in
    let u = make_constraints [] t2 (Var("A")) in
    let u' = unify u in
    let t' = apply2term u' t2 in
    (showtylam t',showtype (typeof [] t'));;

print_string "\n******** type inference test problems********\n ";;
print_string "#1 : INFERTYPE \\x.\\y.\\z.(x z) (y z)\n";;
type_inf [] "\\x.\\y.\\z.(x z) (y z)";;
print_string "#2 : INFERTYPE \\x.x x\n";;
  type_inf [] "\\x.x x";;
print_string "#3 : INFERTYPE \\x.\\y.y(x y)\n";;
    type_inf [] "\\x.\\y.y(x y)";;
print_string "#4 : INFERTYPE \\x.\\y.(y x) y\n";;
      type_inf [] "\\x.\\y.(y x) y";;
print_string "#5 : INFERTYPE \\x.\\y.x (x y)\n";;
type_inf [] "\\x.\\y.x (x y)";;

print_string "#6 : INFERTYPE \\x.\\y.x y x\n";;
type_inf [] "\\x.\\y.x y x";;
```

Notice that there is **very little** to complete! The hard work is reading and understanding the code that's there. You may not think it's worth the trouble to understand all the utility and helper functions I provided. **In that case:** forget the supplied code and just implement the whole darn thing yourself, stealing the parts of my code that are useful to you (e.g. the lexer and parser). This is the most fun option! You **must** include my tests at the end, however. Good luck!