Assignment 3: Type Inference

Due: March 4 2016 1AM
Support code:https://www.cs.umass.edu/~arjun/courses/cmpsci631-spring2016//hw/typeinf.zip

In this assignment, you will implement type inference for a language that is almost identical to the earlier type-checker assignment. The support code has a parser for the implicitly-typed syntax and a pretty-printer for the explicitly-typed syntax. Your will write all the intermediate stages and put them together. To help you verify your code, you should be able to drop in the type-checker that you wrote for the earlier assignment and run it with almost no changes.

1 Setup

The support code for this assignment is distributed using *OPAM* (the OCaml Package Manager), which you installed to do the first assignment. From the command line, run:

```
opam repository add plasma git@github.com:plasma-umass/opam-repository.git
opam install compsci631
```

You should get no errors.

2 Support Code

The documentation for the support code is available online[1].

All the code is in a module called `Typeinf`, which has two sub-modules:

- `Typeinf.Implicit` defines the implicitly-typed syntax and functions to parse implicitly typed programs.
- `Typeinf.Explicit`, which define the explicitly-typed syntax and functions to pretty-print explicitly typed programs.

The `Typeinf` module defines a few types, such as the type of constants, that are shared by both languages.

[1] http://plasma-umass.github.io/compsci631/docs/Typeinf.html

3 Requirements

Your submission must have a file called `Main.ml` with the following function:

```
let typeinf (exp : Typeinf.Implicit.exp) : Typeinf.Explicit.exp = ...
```

4 Directions

4.1 Placeholder Type Identifiers

Write a function to create an explicit-typed AST with fresh metavariables:

```
let rec add_metavars (exp : Typeinf.Implicit.exp) : Typeinf.Explicit.exp = ...
```

The OCaml type for types in this language, `Typeinf.Explicit.typ` has a constructor called `TMetavar`, which you can use to store metavariables. (Remember that metavariables are not types, but placeholders that will get filled with types.)

A metavariable is represented as pair of an integer and a string, `(n, str)` and is printed as `"n-x"`. The string helps with debugging and error reporting. The integer should be incremented each time a new metavariable is created.

Tip: After this step, type-inference works only with explicitly-typed ASTs. Therefore, you may find it convenient to import the explicitly-typed AST and use a module-alias to refer to implicitly-typed expressions:

```
open Typeinf
open Explicit

module I = Typeinf.Implicit

let rec add_metavars (exp : I.exp) : exp = match exp with
  | I.Const c -> Const c
  ...
```

4.2 Constraint Generation

Write a function to calculate types and generate constraints. A constraint equates two types, so you can easily represent a constraint as a pair of types:

```
type constr = typ * typ
```

Here are some examples of the kinds of constraints that some simple expressions generate:

Expression	Constraints
n	$[\![n]\!] = \mathsf{Int}$
b	$[\![b]\!] = \mathsf{Bool}$
$e_1 + e_2$	$[\![e_1]\!] = \mathsf{Int}, [\![e_2]\!] = \mathsf{Int}, [\![e_1 + e_2]\!] = \mathsf{Int}$

The easiest way to implement constraint generation is to write a recursive function that consumes expressions and produces their type and a list of sub-constraints. For example:

```
let rec cgen (exp : exp) : typ * constr = match exp with
  | Const (Int _) -> (TInt, [])
  | Const (Bool _) -> (TBool, [])
  | Op2 (Add, e1, e2) ->
    let (t1, lst1) = cgen e1 in
    let (t2, lst2) = cgen e2 in
    (TInt, [(t1, TInt); (t2, TInt)] @ lst1 @ lst2)
  ...
```

Therefore, `cgen e = (t, lst)` implies the constraint $[\![e]\!] = t$ and `lst` are all the other constraints.

Explicitly threading the list of constraints in this way can get annoying. You may find it easier to just store them all in a global variable, which makes the type of `cgen` simpler:

```
let constraints : (constr list) ref = ref []

let add_constraint (lhs : typ) (rhs : typ) : unit =
  constraints := (lhs, rhs) :: !constraints

let rec cgen (exp : exp) : typ = match exp with
  | Const (Int _) -> TInt
  | Const (Bool _) -> TBool
  | Op2 (Add, e1, e2) ->
    add_constraint (cgen e1) TInt;
    add_constraint (cgen e2) TInt;
    TInt
  ...
```

Finally, it is important to constraint the type of variables correctly. All occurrences of a variable `x` must be constrained to the same type. Moreover, if the program has several binders that name a variable `x`, the constraint generator should not mix them up. The simplest way to tackle this is to add a *type environment* to the constraint generator.

For example, we stated that functions produce a single constraint:

Expression	Constraints
$\lambda x.e$	$[\![\lambda x.e]\!] = [\![x]\!] \rightarrow [\![e]\!]$

However, since the first step was to annotate binders with fresh metavariables, we can instead generate this constraint:

Expression	Constraints
$\lambda x : \alpha.e$	$[\![\lambda x : \alpha.e]\!] = \alpha \rightarrow [\![e]\!]$

Furthermore, we can constraint occurrences of `x` in `e` to α, using a type environment.

The following fragment of code uses this technique to generate constraints for functions and identifiers:

```
type env = (id * typ) list (* Other representations are possible too *)

let rec cgen (env : env) (exp : exp) : constr = match exp with
  | Fun (x, t1, e) ->
    let t2 = cgen ((x, t1) :: env) e in
    TFun (t1, t2)
  | Id x -> List.assoc x env
  ...
```

You can use this code unchanged and tackle `let` and `fix` in a similar way.

4.3 Substitution

The key to solving constraints by unification is the substitution data structure, which maps type identifiers to types. You need to carefully define substitution composition, substitution application, and the substitution constructors. To ensure you use substitutions correctly, we recommend creating an abstract data type for substitutions with the following signature:

```
module type SUBST = sig
  type t

  val empty : t
  val singleton : metavar -> typ -> t
  val apply : t -> typ -> typ
  val compose : t -> t -> t
  val to_list : t -> (metavar * typ) list (* for debugging *)
end
```

Given this signature, you can implement a substitution module as follows:

```
(* The SUBST constraint ensures that t is opaque outside the module *)
module Subst : SUBST = struct
  type t = ...
  let empty = ...
  let singleton x typ = ...
  let apply subst typ = ...
  let compose subst1 subst2 = ...
  let to_list subst = ...
end
```

You may represent a substitution in several ways. For example, you could use a list:

```
(* Substitutions as an association list *)
module Subst : SUBST = struct
  type t = (metavar * typ) list
  ...
end
```

If you do so, you may find the standard library functions for manipulating association lists helpful.

You may find it convenient to use a finite map instead:

```
(* Substitutions as a finite map *)
module Subst : SUBST = struct
  module IdMap = Map.Make (String)
  type t = typ IdMap.t
  ...
end
```

In the code above, the IdMap module has the following signature Map.S.

To help you get started, here are some tests that demonstrate simple properties of substitutions:

```
(* Some examples of operations on substitutions *)
let x : metavar = (0, "x")
let y : metavar = (1, "y")

TEST "Subst.apply should replace x with TInt" =
  let s = Subst.singleton x TInt in
  Subst.apply s (TMetavar x) = TInt

TEST "Subst.apply should recur into type constructors" =
  let s = Subst.singleton x TInt in
  Subst.apply s (TFun (TMetavar x, TBool)) = (TFun (TInt, TBool))

TEST "Subst.compose should distribute over Subst.apply (1)" =
  let s1 = Subst.singleton x TInt in
  let s2 = Subst.singleton y TBool in
  Subst.apply (Subst.compose s1 s2) (TFun (TMetavar x, TMetavar y)) =
  Subst.apply s1 (Subst.apply s2 (TFun (TMetavar x, TMetavar y)))
```

```
TEST "Subst.compose should distribute over Subst.apply (2)" =
  let s1 = Subst.singleton x TBool in
  let s2 = Subst.singleton y (TMetavar x) in
  Subst.apply (Subst.compose s1 s2) (TFun (TMetavar x, TMetavar y)) =
  Subst.apply s1 (Subst.apply s2 (TFun (TMetavar x, TMetavar y)))
```

4.4 Unification

Unification is a function that takes two types as arguments and produces a substitution that maps type identifiers to types:

```
let unify (t1 : typ) (t2 : typ) : Subst.t = ...
```

To help you get started, here is a small test suite that tests some key features of unification.

```
(* An incomplete suite of tests for unification *)
TEST "unifying identical base types should return the empty substitution" =
  Subst.to_list (unify TInt TInt) = []

TEST "unifying distinct base types should fail" =
  try let _ = unify TInt TBool in false
  with Failure "unification failed" -> true

TEST "unifying with a variable should produce a singleton substitution" =
  let x = (0, "myvar") in
  Subst.to_list (unify TInt (TId x)) = [(x, TInt)]

TEST "unification should recur into type constructors" =
  let x = (0, "myvar") in
  Subst.to_list (unify (TFun (TInt, TInt))
                       (TFun (TId x, TInt))) =
  [(x, TInt)]

TEST "unification failures may occur across recursive cases" =
  try
    let x = (0, "myvar") in
    let _ = unify (TFun (TInt, TId x))
                  (TFun (TId x, TBool)) in
    false
  with Failure "unification failed" -> true

TEST "unification should produce a substitution that is transitively closed" =
  let x = (0, "myvar1") in
  let y = (1, "myvar2") in
  let z = (2, "myvar3") in
  let subst = unify (TFun (TFun (TInt, TId x), TId y))
                    (TFun (TFun (TId x, TId y), TId z)) in
  Subst.to_list subst = [ (z, TInt); (y, TInt); (x, TInt) ]

TEST "unification should detect constraint violations that require transitive
      closure" =
  try
    let x = (0, "myvar1") in
    let y = (1, "myvar2") in
    let _ = unify (TFun (TFun (TInt, TId x), TId y))
                      (TFun (TFun (TId x, TId y), TBool)) in
    false
  with Failure "unification failed" -> true

TEST "unification should implement the occurs check (to avoid infinite loops)" =
  try
    let x = (0, "myvar") in
    let _ = unify (TFun (TInt, TId x)) (TId x) in
    false (* a bug is likely to cause an infinite loop *)
  with Failure "occurs check failed" -> true
```

4.5 Constraint Solving

Using the `unify` function you wrote above, write a function to solve a list of constraints by repeatedly applying unification to the pair of types in each constraint. Remember to apply the substitution you produce at each step to the constraints that have yet to be unified.

4.6 Type Annotation

Write a function that substitutes the metavariables in the explicitly-typed AST with concrete types that you calculated in the last step:

```
let annotate_exp (subst : Subst.t) (exp : exp) : exp = ...
```

4.7 Type Checking

This step isn't strictly required, but we strongly recommend you type-check the programs produced by the previous step. With some light modifications, you can reuse the type checker you wrote earlier. Checking the annotated code will help you catch bugs.

4.8 Finish

Finally, put the pieces above together to build the inference function. This function shouldn't do anything more than apply the functions above in the right order.