Homework: Type Inference

In this assignment, you will implement type inference for the language that you type-checked earlier. You system will read a program in the language for the *Extended Interpreter* assignment (i.e., a program with no type annotations) and transform it into an abstract syntax tree for the language of the *Type Checker* assignment. Your will write all the intermediate stages and put them together. To help you verify your code, you should be able run the type-checker that you've already written with no changes.

1 Restrictions

None.

2 Requirements

Write a program that takes one argument:

`./typeinf.d.byte` *program*

The argument *program* should be the name of a file that contains a program written using the grammar of the *Extended Interpreter* assignment (in fig. 8.1). If the program is type-correct, you should exit normally with exit code 0. If there is a type error, it should exit with a non-zero exit code.

You do not have to implement let-polymorphism, but feel free to do so. (Let me know if you do.)

3 Support Code

There isn't any new support code for this assignment. Instead, you can use the *Tc_util* and *Xinterp_util* modules from the two earlier assignments. We will refer to these as the *explicitly typed* syntax and the *implicitly typed* syntax respectively:

```
module Explicit = Tc_util
module Implicit = Xinterp_util
```

4 Directions

4.1 Placeholder Type Identifiers

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

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

The OCaml type for types in this language, *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.)

Each metavariable that you generate should be unique. The easiest way to do this step is to have a global counter that you increment each time you need a fresh metavariable and use the builtin function *string_of_int* to turn that counter into a string.

After you complete this step, type-inference works only with explicitly-typed ASTs.

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	$[\![n]\!] = int$
b	$[\![b]\!] = bool$
$e_1 + e_2$	$[\![e_1]\!] = int, [\![e_2]\!] = int, [\![e_1 + e_2]\!] = int$

In the lecture notes, fig. 12.2 has constraint generation rules for a significant subset of this assignment. You should use these rules as a starting point.

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. To resolve this problem, you can either (1) rename identifiers in the program so that all identifiers are unique or (2) add a *type environment* to the constraint generator, which is the approach taken in fig. 12.2.

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

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

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

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

Furthermore, we can constrain 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 = "__x"
let y : metavar = "__y"

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

let%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))

let%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)))

let%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 = ...
```

Do not forget to implement the *occurs check*, or unification may run forever.
To help you get started, the support code includes a small number of tests for unification.

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. 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.