MP 6 – A Unification-Based Type Inferencer

CS 421 – Spring 2014
Revision 1.0

Assigned Feb 27, 2014
Due Mar 16, 2014 23:59

1 Change Log

1.0 Initial Release.

2 Caution

This assignment can appear quite complicated at first. It is essential that you understand how all the code you will write will eventually work together. Please read through all of the instructions and the given code thoroughly before you start, so you have some idea of the big picture.

3 Objectives

Your objectives are:

- Become comfortable using record types and variant types, particularly as used in giving Abstract Syntax Trees.
- Become comfortable with the notation for semantic specifications.
- Understand the type-inference algorithm.

4 Background

One of the major objectives of this course is to provide you with the skills necessary to implement a language. There are three major components to a language implementation: the parser, the internal representation, and the interpreter[1]. In this MP you will work on the middle piece, the internal representation.

A language implementation represents an expression in a language with an *abstract syntax tree* (AST), usually implemented by means of a user-defined type. Functions can be written that use this type to perform evaluations, preprocessing, and anything that can or should be done with a language. In this MP, you will write some functions that perform type inferencing using unification. This type-inferencer will appear again as a component in several future MPs. You will be given a collection of code to support your work including types for abstract syntax trees, environments and a unification procedure in `Mp6common`. You will be asked to implement the unification procedure in MP7.

4.1 Type Inferencing Overview

The pattern for type inferencing is similar to the procedure used to verify an expression has a type. The catch is that you are not told the type ahead of time; you have to figure it out as you go. The procedure is as follows:

[1] A language implementation may instead/also come with a compiler. In this course, however, we will be implementing an interpreter.

1. Infer the types of all the subexpressions. For each subexpression, you will get back a proof tree and a substitution. You will incrementally apply the substitution for the proof of the judgment for one subexpression to all subsequent judgments for the remaining subexpressions.

2. Create a new proof tree from the proof trees for the subexpressions.

3. Create a new substitution by composing the substitutions of the subexpressions, together with a substitution representing any additional information learned from the application of the rule in question, if there is any.

4. Return the new proof tree and the new substitution.

5 Given Code

This semester the language for which we shall build an interpreter, which we call MicroML, is mainly a simplification of SML. In this assignment we shall build a type inferencer for expressions in MicroML. The file `mp6common.cmo` contains compiled code to support your construction of this type inferencer. Its contents are described here.

5.1 OCaml Types for MicroML AST

Expressions in MicroML are quite similar to expressions in OCaml. The Abstract Syntax Trees for MicroML expressions are given by the following OCaml type:

```
type dec =  (* This type will be expanded in later MPs *)
     Val of string * exp             (* val x = exp *)
   | Rec of string * string * exp    (* val rec f x = exp *)
   | Seq of dec * dec                (* dec1 dec2 *)
   | Local of dec * dec              (* local dec1 in dec2 end *)

and exp =
   | VarExp of string                    (* variables *)
   | ConstExp of const                   (* constants *)
   | MonOpAppExp of mon_op * exp         (* % exp1
                    where % is a builtin monadic operator *)
   | BinOpAppExp of bin_op * exp * exp   (* exp1 % exp2
                    where % is a builtin binary operator *)
   | IfExp of exp * exp * exp            (* if exp1 then exp2 else exp3 *)
   | AppExp of exp * exp                 (* exp1 exp2 *)
   | FnExp of string * exp               (* fn x => x *)
   | LetExp of dec * exp                 (* let dec in exp end *)
   | RaiseExp of exp                     (* raise e *)
   | HandleExp of (exp * int option * exp * (int option * exp) list)
                          (* e handle i => e0 | j => e1 | ...| k => en *)
```

This type makes use of the auxiliary types:

```
type const =
     BoolConst of bool        (* for true and false *)
   | IntConst of int          (* 0,1,2, ... *)
   | RealConst of float       (* 2.1, 3.0, 5.975, ... *)
   | StringConst of string    (* "a", "hi there", ... *)
   | NilConst                 (* [ ] *)
   | UnitConst                (* ( ) *)
```

```
type bin_op =
     IntPlusOp         (* _ + _ *)
   | IntMinusOp        (* _ - _ *)
   | IntTimesOp        (* _ * _ *)
   | IntDivOp          (* _ / _ *)
   | RealPlusOp        (* _ +. _ *)
   | RealMinusOp       (* _ -. _ *)
   | RealTimesOp       (* _ *. _ *)
   | RealDivOp         (* _ /. _ *)
   | ConcatOp          (* _ ^ _ *)
   | ConsOp            (* _ :: _ *)
   | CommaOp           (* _ , _ *)
   | EqOp              (* _ = _ *)
   | GreaterOp         (* _ > _ *)

type mon_op =
     IntNegOp      (* integer negation *)
   | HdOp          (* hd *)
   | TlOp          (* tl *)
   | FstOp         (* fst *)
   | SndOp         (* snd *)
   | PrintStringOp (* print_string *)
```

for representing the constants, and binary and unary operations in our language. Any of these types may be expanded in future MPs in order to enrich the language.

Some of the constructors of `exp` should be self-explanatory. Names of constants are represented by the type `const`. Names of variables are represented by strings. `BinOpAppExp` takes the binary operator, represented by the type `bin_op`, together with two operands. Similarly, `MonOpAppExp` takes the unary operator of the `mon_op` type and an operand. The constructors that take `string` arguments (`VarExp`, `FunExp`, `Val`, and `Rec`) use strings to represent names of variables that they bind. We have added in `RaiseExp` and `HandleExp` for raising and handling exceptions, but have limited exceptions to integers, rather in the style of Unix. We use `int option` in place of `int` to allow for a catchall pattern. We have completed declarations with declarations local to other declarations.

There are companion functions `string_of_exp` and `string_of_dec` that convert expressions, and declarations respectively, into a string using a more readable form, similar to OCaml or SML concrete syntax.

5.2 OCaml Types for MicroML Types

In addition to having abstract syntax trees for the expressions of MicroML, we need to have abstract syntax trees for the types of MicroML. The types of MicroML can be categorized into two kinds: monomorphic types and polymorphic types. Monomorphic types are simple: type variables, and type constructors applied to a sequence of types. To make types this uniform, we will consider type constants as type constructors applied to an empty sequence of types. Thus we may use the following OCaml type to represent the monomorphic types of MicroML:

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

Type variables will just be represented by integers:

```
type typeVar = int
```

Again, there is a companion function `string_of_monoTy` that generates a string containing a more readable form of the `monoTy`, similar to OCaml concrete syntax for types.

Polymorphic types in MicroML are universally quantified monomorphic types. We will represent the quantified (and thus bound) type variables by a list of `typeVar`:

```
type polyTy = typeVar list * monoTy  (* the list is for quantified variables *)
```

Again, there is a function `string_of_polyTy` that gives a more readable form of a given `polyTy`.

A monomorphic type can be considered as a `polyTy` where the list of quantified type variables is the empty list.

When inferring types, you will need to generate fresh type-variable names. For this, you may use the side-effecting function `fresh` that takes unit and returns a fresh type variable. The index stored by `fresh` (initially set to 0) will keep on growing as you use `fresh`.

5.3 Environments

We need an environment to store the types of the variables. Later, we will need environments to store values for the variables for use during execution. The idea of an environemt and the operations we perform over them is independent of the informention staored for each variable. An environment in gnereal will be represented by a list of pairs mapping variables (strings) to information (types here, values later):

```
type 'a env = (string * 'a) list
```

One interacts with environments using the following functions, pre-defined in `mp6common.ml`:

```
(*environment operations*)
let make_env x y = ...           (*create env with single pair*)
let rec lookup_env gamma x = ... (*look up x in gamma*)
let sum_env delta gamma = ...    (*update gamma with all mappings in delta*)
let ins_env gamma x y = ...      (*insert x->y into gamma*)

val make_env : string -> 'a -> 'a env = <fun>
val lookup_env : 'a env -> string -> 'a option = <fun>
val sum_env : 'a env -> 'a env -> 'a env = <fun>
val ins_env : 'a env -> string -> 'a -> 'a env = <fun>
```

For convenience, since we will only use environments supplying polymorphic types we supply an abbreviation for that specialized type:

```
type type_env = polyTy env
```

5.4 Signatures

In addition to the environment, which will change over the course of executing programs, we need a way to store the types of constants and built-in unary and binary operators in MicroML. Unlike the environment, the signature will be fixed throughout type inference, and is provided here as three functions,

```
val const_signature : const -> polyTy = <fun>
val binop_signature : binop -> polyTy = <fun>
val monop_signature : monop -> polyTy = <fun>
```

taking respectively a `const`, a `binop`, or `monop`, and returning a `polyTy`.

Some constants and operators, like `NilConst`, `ConsOp`, and `CommaOp` have true polymorphic types, with quantified type variables. During type inference, it will be necessary to create distinct monomorphic instances of their polymorphic types, replacing all instances of the quantified variables with fresh variables. For instance, consider the expression

```
((true :: []), (0 :: []))
```

(We use the more familiar OCaml-like notation rather than abstract syntax that we have implemented to represent it.) When typing this expression, we shall eventually get to type both its immediate subexpressions, `(true :: [])` and `(0 :: [])`, separately:

1. When typing `(true :: [])`, we discover that the operator `::` and constant `NilConst` are used with the types `bool -> bool list -> bool list` and `bool list`, respectively; this should be consistent with the types of `::` and `[]` stored in our signature, namely with respectively $\forall\alpha.\ \alpha$ `-> `α `list -> `α `list` and $\forall\alpha.\ \alpha$ `list` (where α is a type variable, written in OCaml as `TyVar 0`). In order for `::` to be applied to `true` (and `NilConst`), we need to replace all instances of α (in both types) to `bool`.

2. Similarly, from typing `(0 :: [])`, we get we need to replace all instances of α by `int`.

The two constraints, α `= bool` and α `= int`, are inconsistent (i.e., have no solution), since they would lead to `bool = int`. Thus, we have done something wrong above, because we will certainly want to allow the use of `::` and `NilConst` polymorphically, dealing with lists of arbitrary types, in particular with lists of booleans and with lists of integers. The above problem comes from binding the same type variable, α to both `bool` and `int` - this is *not* a proper use of the polymorphic variable α, since within a fixed problem the type varaible α should be instantiated everywhere with the *same* type. The solution is to replace α with a fresh type variable each time we type the operator `::` and the constant `[]`; this way, the constraints are: α_1 `list = bool list` and α_2 `-> `α_2 `list -> `α_2 `list = bool -> bool list -> bool list` from typing `(true :: [])`, and α_3 `list = int list` and α_4 `-> `α_4 `list -> `α_4 `list = int -> int list -> int list` from typing `(0 :: []`), yielding the solution $\alpha_1 = \alpha_2 =$`bool` and $\alpha_3 = \alpha_4 =$`int`.

With the signatures for constants and built-in unary and binary operators, we provide polymorphism for the built in constants and binary operators. For those constants and operators like `NilConst` and `ConsOp`, every time their type is requested from the appropriate signature, a new type with fresh type variables is given. The code for these signatures is as follows:

```
let bool_ty = TyConst("bool",[])
let int_ty = TyConst ("int", [])
let real_ty = TyConst ("real",[])
let string_ty = TyConst ("string",[])
let unit_ty = TyConst("unit", [])
let mk_pair_ty ty1 ty2 = TyConst("*",[ty1;ty2])
let mk_fun_ty ty1 ty2 = TyConst("->",[ty1;ty2])
let mk_list_ty ty = TyConst("list",[ty])

let polyTy_of_monoTy mty = (([],mty):polyTy)

let int_op_ty = polyTy_of_monoTy(mk_fun_ty int_ty (mk_fun_ty int_ty int_ty))
let real_op_ty =
    polyTy_of_monoTy(mk_fun_ty real_ty (mk_fun_ty real_ty real_ty))
let string_op_ty =
    polyTy_of_monoTy(mk_fun_ty string_ty (mk_fun_ty string_ty string_ty))

(* fixed signatures *)
let const_signature const = match const with
   BoolConst b -> polyTy_of_monoTy bool_ty
 | IntConst n -> ([], int_ty)
 | RealConst f -> ([], real_ty)
 | StringConst s -> ([], string_ty)
 | NilConst -> ([0],mk_list_ty (TyVar 0))
 | UnitConst -> ([], unit_ty)

let binop_signature binop = match binop with
     IntPlusOp   -> int_op_ty
   | IntMinusOp   -> int_op_ty
   | IntTimesOp   -> int_op_ty
```

```
    | IntDivOp   -> int_op_ty
    | RealPlusOp   -> real_op_ty
    | RealMinusOp   -> real_op_ty
    | RealTimesOp   -> real_op_ty
    | RealDivOp   -> real_op_ty
    | ConcatOp -> string_op_ty
    | ConsOp ->
        let alpha = TyVar 0
        in ([0],
               mk_fun_ty alpha (mk_fun_ty (mk_list_ty alpha) (mk_list_ty alpha)))
    | CommaOp ->
        let alpha = TyVar 0 in
        let beta = TyVar 1 in
            ([0;1],
             mk_fun_ty alpha (mk_fun_ty beta (mk_pair_ty alpha beta)))
    | EqOp ->
      let alpha = TyVar 0 in ([0],mk_fun_ty alpha (mk_fun_ty alpha bool_ty))
    | GreaterOp ->
      let alpha = TyVar 0 in ([0],mk_fun_ty alpha (mk_fun_ty alpha bool_ty))

let monop_signature monop = match monop with
     | HdOp -> let alpha = TyVar 0 in([0], mk_fun_ty (mk_list_ty alpha) alpha)
     | TlOp -> let alpha = TyVar 0 in
                   ([0], mk_fun_ty (mk_list_ty alpha) (mk_list_ty alpha))
     | PrintStringOp -> (([], mk_fun_ty string_ty unit_ty):polyTy)
     | IntNegOp -> ([], mk_fun_ty int_ty int_ty)
     | FstOp ->
        let alpha = TyVar 0 in
        let beta = TyVar 1 in
            ([0;1], mk_fun_ty (mk_pair_ty alpha beta) alpha)
     | SndOp ->
        let alpha = TyVar 0 in
        let beta = TyVar 1 in
            ([0;1], mk_fun_ty (mk_pair_ty alpha beta) beta)
```

To be able to use different instances of these polymorphic types, as well as those occurring in our typing environments, we have provided you with a function `freshInstance : polyTy -> monoTy` that returns an instance where all the quantified variables have been replaced by fresh variables.

5.5 Type Judgments and Proofs

From the lectures, you know that an *expression* type judgment has the form $\Gamma \vdash e : \tau$. This says that in the environment Γ, the expression e has type τ. A *declaration* type judgment has the form $\Gamma \vdash d : \Delta$. This says that, in environment Γ, the declaration d generates bindings for the variables in the environment Δ with each such variable having the type given by Δ.

A proof is recursively defined as a (possibly empty) sequence of proofs together with the judgment being proved. Judgments and proofs are represented by the following data types:

```
type judgment =
   ExpJudgment of type_env * exp * monoTy
 | DecJudgment of type_env * dec * type_env

type proof = Proof of proof list * judgment
```

The pre-defined functions `string_of_jexp` and `string_of_proof` generate more readable forms of these typing judgments and proofs. The function `string_of_proof` generates string for the proof, which when printed is in a tree-like form, with the root at the top, upside-down from the way we are used to seeing proofs.

5.6 Substitutions

In this MP, you are asked to write code returning, among other things, a substitution. A substitution, in it most basic form is a partial mapping from variables to expressions. In our case, we will use substitutions from type variables to monomorphic types. In a similar manner to what we did with environments, we represent substitutions as lists:

```
type substitution = (typeVar * monoTy) list
```

Just as with environments, we need a collection of functions to operate with them. To apply a substitution to each of monomoprhic types, polymporphic types and environments, you are given the following functions:

```
val monoTy_lift_subst : substitution -> monoTy -> monoTy
val polyTy_lift_subst : substitution -> polyTy -> polyTy
val env_lift_subst : substitution -> env -> env
```

You also need to be able to create the substitution that represents the composition of two substitutions. If `s1` and `s2` are two substitutions, then

```
val subst_compose : substitution -> substitution -> substitution
```

can be used to generate their composition.

$$\texttt{subst\_compose s1 s2} = \texttt{s1} \circ \texttt{s2}$$

That is, it creates a substitution that has the same effect as first applying the substitution `s2` and then applying `s1`.

You will start generating the substitutions to be composed by applying unification to a list of constraints implied by the rule you are implementing. A constraint is represented by a pair of monomorphic types, with the intention being that you need to make them equal by filling in their variables with a substitution. The function

```
val unify : (monoTy * monoTy) list -> substitution option
```

returns `Some` of a substitution simultaneously satisfying all the constraints in the input list, if there is one, and `None` if no solution exists. You will be implementing this function in the next MP.

The last function you will need that we supply is one for generalizing a monomorphic type with respect to a typing environment to a polymorphic type where all the variables that do not occur (free) in the range of the environment are universally quantified in the polymorphic type created.

```
val gen : type_env -> monoTy -> polyTy
```

6 Type Inferencing

The rules used for a type-inferencer are derived from the rules for the type system shown in class. The additional complication is that we assume at each step that we do not fully know the types to be checked for each expression. Therefore, as we progress we must accumulate our knowledge in the form of a substitution telling us what we have learned so far about our type variables in both the type we wish to verify and the typing environment. To do so, we supplement our typing judgments with one extra component, a typing substitution. Here's an example:

$$\frac{\Gamma \vdash e_1 \ : \texttt{int} \ \mid \sigma_1 \quad \sigma_1(\Gamma) \vdash e_2 \ : \texttt{int} \ \mid \sigma_2}{\Gamma \vdash e_1 + e_2 : \tau \mid \textit{unify}\{(\sigma_2 \circ \sigma_1(\tau), \texttt{int} \ )\} \circ \sigma_2 \circ \sigma_1}$$

The "|" is just some notation to separate the substitution from the expression. You can pronounce it as "subject to". This rule says that the substitution sufficient to guarantee that the result of adding two expressions e_1 and e_2 will have

type τ is the composition of the substitution σ_1 guaranteeing that e_1 has type `int` , the substitution σ_2 guaranteeing that e_2 has type `int` (when the knowledge form σ_1 is applied to our assumptions), and substitution generated from the constraint the $\{\tau =$ `int` $\}$.

For example, suppose you want to infer the type of `fn x => x + 2`. In English, the reasoning would go like this.

1. Let $\Gamma = \{\}$.
2. We start with a "guess" that `fn x => x + 2` has type `'a`.
3. Next we examine `fn x => x + 2` and see that it is a `fn`. We don't know what `x` will be, so we let it have type `'b`. Add that to Γ and try to infer the type of the body is `'c` ...
   (a) Examine `x + 2`. We apply the above rule, so we need to infer the subtypes.
      i. Examine `x`. Γ says that `x` has type `'b`. We are trying to show it has type `int` . We generate the substitution solving the constraint $\{$`'b` $=$ `int` $\}$.
      ii. Examine `2`. This is an integer, as needed. (To be really thorough we would add the substitution solving the constraint $\{$`int` $=$ `int` $\}$.)
   (b) We combine these inferences together to make a new proof-tree, and add to the combined substitutions the substitution solving the constraint that says the result of applying the combination of the two substitutions known so far to `'c` must be the same as `int`. We need to compose the substitution solving this constraint to the substitutions making `'b` be type `int`, and `int` be type `int`. (Yes, that last one was trivial, but the rule says we have to do it. It amounts to composing wth the identity function.)
4. Now we're ready to come back to the type of the whole expression. The variable `x` has type `'b`, and the output has type `'c`, but the whole expression has type `'a`, so `'a` must also be `'b -> 'c`. But, from above we have already learned that `'b = int` and that `'c = int`, and recorded this in our substitutions. Applying the substitutions accumulated so far to the constraint that `'a = 'b -> 'c`, we generate the substitution that solves `'a = int -> int`.
5. The result of our combined substitutions tell us that we need to rewrite `'b` and `'c` to `int` everywhere, and rewrite `'a` to `int -> int` everywhere. In particular, we get a final type of `int -> int`.

6.1 Pre-defined Testing Functions

Some important functions for testing your code are pre-defined: The function `infer_exp`, takes in a function `gather_exp_ty_substitution`, an `env` and an `exp` and returns an `(monoTy * proof) option`. The first part of the result type is the type of the entire expression and the second part is a proof (assuming success).

`infer_exp` works by generating a fresh type variable τ and calling the function `gather_exp_ty_substitution` and gets back a (generic) proof tree and a substitution. If `gather_exp_ty_substitution` returns `None`, then `infer_exp` returns `None`. Otherwise, `infer_exp` applies the substitution to τ to obtain the ultimate type. This ultimate type as well as the proof are then returned in a `Some` of a pair. There exists an analogous function `gather_exp_ty_substitution` that works on `dec`s.

The functions `get_proof` and `get_ty` extract the proof and type parts, respectively (or raise an exception on `None`).

```
val infer_exp :
  (type_env -> exp -> monoTy -> (proof * substitution) option) ->
  type_env -> exp -> (monoTy * proof) option
val infer_dec :
  (type_env -> dec -> (proof * substitution) option) ->
  type_env -> dec -> (type_env * proof) option
val get_ty : ('a * 'b) option -> 'a
val get_proof : ('a * 'b) option -> 'b
```

There are also verbose forms of `infer_exp` and `infer_dec`:

```
val niceInfer_exp :
  (type_env -> exp -> monoTy -> (proof * substitution) option) ->
  type_env -> exp -> string
val niceInfer_dec :
  (type_env -> dec -> (proof * type_env * substitution) option) ->
  type_env -> dec -> string
```

that prints out details about the substitution that is gathered, and the results of applying the substitution to the original fresh type. You will see these functions used in examples below.

7 Problems: Your task

The bodies of the main type inferencing functions, `infer_exp` and `infer_dec`, are already implemented. Your task is to finish the implementation of the main functions needed by `infer_exp` and `infer_dec`: `gather_exp_ty_substitution : type_env -> exp -> monoTy -> (proof * substitution) option` takes in a type environment, an expression, and a type and returns `None` (on failure), or `Some` of a pair of a generic proof tree containing type variables, and a substitution. Similarly, `gather_dec_ty_substitution : type_env -> dec -> (proof * type_env * substitution) option` takes in a type environment and a declaration, and again returns `None` (on failure), or `Some` of a triple of a generic proof tree containing type variables, a type environment, and a substitution. The type environment is already in the proof tree, but it will be most convenient for our algorithm to hand it back directly as well, since we will always immediately need it. In each case, when the substitution returned is applied to the proof tree returned, the result is a fully valid type derivation.

To help you get started, we will give you the clause for `gather_exp_ty_substitution` for a constant expression:

```
match exp
with ConstExp c ->
     let tau' = const_signature c in
     (match unify [(tau, freshInstance tau')]
      with None       -> None
         | Some sigma -> Some(Proof([],judgment), sigma))
| _ -> raise (Failure "Not implemented yet")
```

This implements the rule

$$\frac{}{\Gamma \vdash c : \tau \mid \mathit{unify}\{(\tau, \mathit{freshInstance}(\tau'))\}}$$

where c is a special constant, and τ' is an instance of the type assigned by the constants signature.

A sample execution would be:

```
# print_string(string_of_proof(get_proof(
       infer_exp gather_exp_ty_substitution [] (ConstExp (BoolConst true)))));;

  {} |= true : bool
```

To see what happened in greater details, we may do:

```
# print_string
 (niceInfer_exp gather_exp_ty_substitution [] (ConstExp(BoolConst true)));;

  {} |= true : 'b
```

```
Unifying substitution: ['b --> bool]
Substituting...

  {} |= true : bool
```

It is not necessary for your work to generate exactly the same substitution that our solution gives. What is required is that the type you get for an expression must be an instance of the type the standard solution gets, and the type given by the standard solution must be an instance of the type you give. As a result, running `niceInfer` on the standard solution will give one way that the type inference could proceed, but it likely is not the only way.

1. (5 pts) Implement the rule for variables:

$$\frac{}{\Gamma \vdash x : \tau \mid \mathit{unify}\{(\tau, \mathit{freshInstance}(\Gamma(x)))\}} \text{ where } x \text{ is a program variable}$$

   Note that $\Gamma(x)$ represents looking up the value of x in Γ. In OCaml, one writes `Mp6common.lookup_env gamma x` where `x` is the string naming the variable.

   A sample execution is

```
# print_string (niceInfer_exp gather_exp_ty_substitution
                (make_env "f" ([0], mk_fun_ty bool_ty (TyVar 0)))
                (VarExp "f"));;

  {f : Forall 'a. bool -> 'a} |= f : 'b

Unifying substitution: ['b --> bool -> 'c]
Substituting...

  {f : Forall 'a. bool -> 'a} |= f : bool -> 'c
```

2. (10 pts) Implement the rule for built-in binary and unary operators:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \sigma_1 \quad \sigma_1(\Gamma) \vdash e_2 : \tau_2 \mid \sigma_2}{\Gamma \vdash e_1 \otimes e_2 : \tau \mid \mathit{unify}\{(\sigma_2 \circ \sigma_1(\tau_1 \to \tau_2 \to \tau), \mathit{freshInstance}(\tau'))\} \circ \sigma_2 \circ \sigma_1}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \sigma}{\Gamma \vdash \otimes e_1 : \tau \mid \mathit{unify}\{(\sigma(\tau_1 \to \tau), \mathit{freshInstance}(\tau'))\} \circ \sigma}$$

   where $\otimes$ is a built-in binary or unary operator, and τ' is an instance of the type assigned by the signature for the built-in operator.

   A sample execution would be:

```
# print_string (niceInfer_exp gather_exp_ty_substitution []
                (BinOpAppExp(ConsOp, ConstExp (IntConst 62), ConstExp NilConst)));;

  {} |= 62 :: [] : 'b
  |--{} |= 62 : 'c
  |--{} |= [] : 'd
```

```
Unifying substitution: [’b --> int list; ’e --> int; ’f --> int; ’d --> int list;
 ’c --> int]
Substituting...

  {} |= 62 :: [] : int list
  |--{} |= 62 : int
  |--{} |= [] : int list

- : unit = ()
# print_string (niceInfer_exp gather_exp_ty_substitution []
                (MonOpAppExp(PrintStringOp, ConstExp(StringConst "hi"))));;

  {} |= print_string "hi" : ’b
  |--{} |= "hi" : ’c

Unifying substitution: [’b --> unit; ’c --> string]
Substituting...

  {} |= print_string "hi" : unit
  |--{} |= "hi" : string

- : unit = ()
```

Please note that your results on these and all other problems may look different and still be correct. For example, the order of the terms in substitutions is not fixed. Moreover, if you choose fresh variables for subterms in a different order than we did, your answer would still be correct while assigning the type variables differently.

3. (10 pts) Implement the rule for `if_then_else`:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \mid \sigma_1 \quad \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau) \mid \sigma_2 \quad \sigma_2 \circ \sigma_1(\Gamma) \vdash e_3 : \sigma_2 \circ \sigma_1(\tau) \mid \sigma_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \mid \sigma_3 \circ \sigma_2 \circ \sigma_1}$$

For this problem, you will have to recursively construct proofs with constraints for each of the subexpressions, and then use these results to build the final proof and constraints.

Here is a sample execution:

```
# print_string (niceInfer_exp gather_exp_ty_substitution []
    (IfExp(ConstExp(BoolConst true),
           ConstExp(IntConst 62), ConstExp(IntConst 252))));;

  {} |= if true then 62 else 252 : ’b
  |--{} |= true : bool
  |--{} |= 62 : ’b
  |--{} |= 252 : int

Unifying substitution: [’b --> int]
Substituting...

  {} |= if true then 62 else 252 : int
```

```
  |--{} |= true : bool
  |--{} |= 62 : int
  |--{} |= 252 : int

- : unit = ()
```

4. (10pts) Implement the function rule:

$$\frac{[x : \tau_1] + \Gamma \vdash e : \tau_2 \mid \sigma}{\Gamma \vdash \texttt{fn } x \texttt{ => } e : \tau \mid \textit{unify}\{(\sigma(\tau),\, \sigma(\tau_1 \to \tau_2))\} \circ \sigma}$$

Here is a sample execution:

```
# print_string (niceInfer_exp gather_exp_ty_substitution []
    (FnExp("x", BinOpAppExp(IntPlusOp, VarExp "x", VarExp "x"))));;

  {} |= fn x => x + x : 'b
  |--{x : 'c} |= x + x : 'd
    |--{x : 'c} |= x : 'e
    |--{x : 'c} |= x : 'f

Unifying substitution: ['b --> int -> int; 'd --> int; 'c --> int; 'f --> int;
 'e --> int]
Substituting...

  {} |= fn x => x + x : int -> int
  |--{x : int} |= x + x : int
    |--{x : int} |= x : int
    |--{x : int} |= x : int

- : unit = ()
```

5. (10 pts) Implement the rule for application:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \mid \sigma_1 \quad \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau_1) \mid \sigma_2}{\Gamma \vdash e_1 e_2 : \tau \mid \sigma_2 \circ \sigma_1}$$

Here is a sample execution:

```
# print_string (niceInfer_exp gather_exp_ty_substitution []
  (AppExp(FnExp("x", BinOpAppExp(IntPlusOp, VarExp "x", VarExp "x")),
          ConstExp(IntConst 62))));;

  {} |= (fn x => x + x) 62 : 'b
  |--{} |= fn x => x + x : 'c -> 'b
  | |--{x : 'd} |= x + x : 'e
  |   |--{x : 'd} |= x : 'f
  |   |--{x : 'd} |= x : 'g
```

```
  |--{} |= 62 : int

Unifying substitution: ['b --> int; 'c --> int; 'e --> int; 'd --> int;
 'g --> int; 'f --> int]
Substituting...

  {} |= (fn x => x + x) 62 : int
  |--{} |= fn x => x + x : int -> int
  | |--{x : int} |= x + x : int
  |   |--{x : int} |= x : int
  |   |--{x : int} |= x : int
  |--{} |= 62 : int

- : unit = ()
```

6. (7 pts) Implement the rule for `raise`

$$\frac{\Gamma \vdash e : \texttt{int} \mid \sigma}{\Gamma \vdash \texttt{raise}\ e : \tau \mid \sigma}$$

Here is a sample execution:

```
# print_string (niceInfer_exp gather_exp_ty_substitution []
  (RaiseExp(IfExp(ConstExp(BoolConst true), ConstExp(IntConst 62),
                  ConstExp(IntConst 252)))));;

  {} |= raise if true then 62 else 252 : 'b
  |--{} |= if true then 62 else 252 : int
    |--{} |= true : bool
    |--{} |= 62 : int
    |--{} |= 252 : int

Unifying substitution: []
Substituting...

  {} |= raise if true then 62 else 252 : 'b
  |--{} |= if true then 62 else 252 : int
    |--{} |= true : bool
    |--{} |= 62 : int
    |--{} |= 252 : int
```

Before we can conclude all out cases for inferring types for expressions, we need to also consider how to infer type information for declarations, and even what information to collect. For a single expressions, what you are trying it infer is its type. In a declaration, we have a binding of a collection of expressions to a corresponding collection of identifiers. The type information we need to infer then is a collection of bindings of the types of those expression to those corresponding identifiers. Thus, declarations need to have type environments as their types. The type environments will be increments to be added to the given environment.

The type of `gather_dec_ty_substitution` is `type_env -> dec -> (proof * type_env * substitution) option`.

7. (5 pts) For a simple declaration, a single expression is typed in the given type environment, and the incremental type environment associating the given identifier with the type inferred for the expression. The substitution generated

by inferring the type of the expression is the substitution needed to infer the type environment for the declaration. Implement the following rule for simple `val` declarations:

$$\frac{\Gamma \vdash e : \tau \mid \sigma}{\Gamma \vdash \texttt{val}\ x = e : [x : \mathit{GEN}(\sigma(\Gamma), \sigma(\tau))] \mid \sigma}$$

Here is a sample execution:

```
# print_string(niceInfer_dec gather_dec_ty_substitution []
                  (Val("x", BinOpAppExp(ConsOp, ConstExp NilConst,
                                        ConstExp NilConst))));;

  {} |= val x = [] :: [] : {x : Forall 'd. 'd list list}
  |--{} |= [] :: [] : 'b
    |--{} |= [] : 'c
    |--{} |= [] : 'e

Unifying substitution: ['b --> 'd list list; 'f --> 'd list; 'g --> 'd list;
 'e --> 'd list list; 'c --> 'd list]
Substituting...

  {} |= val x = [] :: [] : {x : Forall 'a. 'a list list}
  |--{} |= [] :: [] : 'd list list
    |--{} |= [] : 'd list
    |--{} |= [] : 'd list list

- : unit = ()
```

8. (10 pts) Implement the following `val rec` rule for recursive declarations:

$$\frac{[f : \tau_1 \to \tau_2,\ x : \tau_1] + \Gamma \vdash e : \tau_2 \mid \sigma}{\Gamma \vdash \texttt{val rec}\ f\ x = e : [f : \mathit{GEN}(\sigma(\Gamma), \sigma(\tau_1 \to \tau_2))] \mid \sigma}$$

Here is a sample execution:

```
# print_string
  (niceInfer_dec gather_dec_ty_substitution []
    (Rec("length",
     "list",
     IfExp(BinOpAppExp(EqOp, VarExp "list", ConstExp NilConst),
           ConstExp (IntConst 0),
           BinOpAppExp(IntPlusOp, ConstExp (IntConst 1),
                       (AppExp(VarExp "length",
                               MonOpAppExp(TlOp,VarExp "list")))))))));;


  {} |= val rec length list = if list = [] then 0 else 1 + (length (tl list)) : {length : Forall 'l.
                                                                                   'l list -> int}
  |--{length : 'b -> 'c, list : 'b} |= if list = [] then 0 else 1 + (length (tl list)) : 'c
    |--{length : 'b -> 'c, list : 'b} |= list = [] : bool
    | |--{length : 'b -> 'c, list : 'b} |= list : 'd
    | |--{length : 'b -> 'c, list : 'b} |= [] : 'e
    |--{length : 'f list -> 'c, list : 'f list} |= 0 : 'c
    |--{length : 'f list -> int, list : 'f list} |= 1 + (length (tl list)) : int
```

```
      |--{length : 'f list -> int, list : 'f list} |= 1 : 'h
      |--{length : 'f list -> int, list : 'f list} |= length (tl list) : 'i
        |--{length : 'f list -> int, list : 'f list} |= length : 'j -> 'i
        |--{length : 'f list -> int, list : 'f list} |= tl list : 'f list
          |--{length : 'f list -> int, list : 'f list} |= list : 'k

Unifying substitution: ['f --> 'l; 'k --> 'l list; 'i --> int; 'j --> 'l list; 'h --> int; 'c --> int;
                        'g --> 'l list; 'b --> 'l list; 'e --> 'l list; 'd --> 'l list]
Substituting...

  {} |= val rec length list = if list = [] then 0 else 1 + (length (tl list)) : {length : Forall 'a.
                                                                                   'a list -> int}
  |--{length : 'l list -> int, list : 'l list} |= if list = [] then 0 else 1 + (length (tl list)) : int
    |--{length : 'l list -> int, list : 'l list} |= list = [] : bool
    | |--{length : 'l list -> int, list : 'l list} |= list : 'l list
    | |--{length : 'l list -> int, list : 'l list} |= [] : 'l list
    |--{length : 'l list -> int, list : 'l list} |= 0 : int
    |--{length : 'l list -> int, list : 'l list} |= 1 + (length (tl list)) : int
      |--{length : 'l list -> int, list : 'l list} |= 1 : int
      |--{length : 'l list -> int, list : 'l list} |= length (tl list) : int
        |--{length : 'l list -> int, list : 'l list} |= length : 'l list -> int
        |--{length : 'l list -> int, list : 'l list} |= tl list : 'l list
          |--{length : 'l list -> int, list : 'l list} |= list : 'l list

- : unit = ()
```

9. (10 pts) Implement the rule for sequences of declarations:

$$\frac{\Gamma \vdash dec_1 : \Delta_1 \mid \sigma_1 \quad \sigma_1(\Delta_1 + \Gamma) \vdash dec_2 : \Delta_2 \mid \sigma_2}{\Gamma \vdash dec_1 \;\; dec_2 : \sigma_2 \circ \sigma_1(\Delta_2 + \Delta_1) \mid \sigma_2 \circ \sigma_1}$$

Caution: The updating of one environment by another, written with the infix $+$, is NOT commutative. Be sure to do them in the order given.

Here is a sample execution:

```
# print_string
  (niceInfer_dec gather_dec_ty_substitution []
   (Seq(Val("x", ConstExp (BoolConst true)), Val("y", ConstExp (IntConst 3)))));;

  {} |= val x = true
val y = 3 : {y : int,x : bool}
  |--{} |= val x = true : {x : bool}
  | |--{} |= true : 'b
  |--{x : bool} |= val y = 3 : {y : int}
    |--{x : bool} |= 3 : 'c

Unifying substitution: ['c --> int; 'b --> bool]
Substituting...

  {} |= val x = true
val y = 3 : {y : int,x : bool}
  |--{} |= val x = true : {x : bool}
  | |--{} |= true : bool
  |--{x : bool} |= val y = 3 : {y : int}
    |--{x : bool} |= 3 : int
```

```
- : unit = ()
```

10. (10 pts) Implement the rule for `let_in_end` expressions:

$$\frac{\Gamma \vdash dec : \Delta \mid \sigma_1 \quad \Delta + \sigma_1(\Gamma) \vdash e : \sigma_1(\tau) \mid \sigma_2}{\Gamma \vdash \texttt{let}\ dec\ \texttt{in}\ e\ \texttt{end} : \tau \mid \sigma_2 \circ \sigma_1}$$

Here is a sample execution:

```
# print_string (niceInfer_exp gather_exp_ty_substitution []
    (LetExp(Val("y", ConstExp(IntConst 5)),
            BinOpAppExp(IntPlusOp, VarExp "y", VarExp "y"))));;

  {} |= let val y = 5 in y + y end : 'b
  |--{} |= val y = 5 : {y : int}
  | |--{} |= 5 : 'c
  |--{y : int} |= y + y : 'b
    |--{y : int} |= y : 'd
    |--{y : int} |= y : 'e

Unifying substitution: ['b --> int; 'e --> int; 'd --> int; 'c --> int]
Substituting...

  {} |= let val y = 5 in y + y end : int
  |--{} |= val y = 5 : {y : int}
  | |--{} |= 5 : int
  |--{y : int} |= y + y : int
    |--{y : int} |= y : int
    |--{y : int} |= y : int

- : unit = ()
```

11. (5 pts) Implement the following rule for declarations local to other declarations:

$$\frac{\Gamma \vdash dec_1 : \Delta_1 \mid \sigma_1 \quad \sigma_1(\Delta_1 + \Gamma) \vdash dec_2 : \Delta_2 \mid \sigma_2}{\Gamma \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} : \sigma_2 \circ \sigma_1(\Delta_2) \mid \sigma_2 \circ \sigma_1}$$

Caution: The updating of one environment by another, written with the infix $+$, is NOT commutative. Be sure to do them in the order given.

Here is a sample execution:

```
# print_string
  (niceInfer_dec gather_dec_ty_substitution []
   (Local(Val("x", ConstExp (BoolConst true)),
          Val("y", IfExp(VarExp "x", ConstExp (IntConst 3),
                                     ConstExp(IntConst 4))))));;
```

```
  {} |= local val x = true
in val y = if x then 3 else 4 end : {y : int}
  |--{} |= val x = true : {x : bool}
  | |--{} |= true : 'b
  |--{x : bool} |= val y = if x then 3 else 4 : {y : int}
    |--{x : bool} |= if x then 3 else 4 : 'c
      |--{x : bool} |= x : bool
      |--{x : bool} |= 3 : 'c
      |--{x : bool} |= 4 : 'c

Unifying substitution: ['c --> int; 'b --> bool]
Substituting...

  {} |= local val x = true
in val y = if x then 3 else 4 end : {y : int}
  |--{} |= val x = true : {x : bool}
  | |--{} |= true : bool
  |--{x : bool} |= val y = if x then 3 else 4 : {y : int}
    |--{x : bool} |= if x then 3 else 4 : int
      |--{x : bool} |= x : bool
      |--{x : bool} |= 3 : int
      |--{x : bool} |= 4 : int

- : unit = ()
```

7.1 Extra Credit

12. (7 pts) Implement the rule for handling exceptions with `handle`:

$$\frac{\Gamma \vdash e : \tau \mid \sigma \quad \sigma_{i-1} \circ \ldots \circ \sigma_1 \circ \sigma(\Gamma) \vdash e_i : \sigma_{i-1} \circ \ldots \circ \sigma_1 \circ \sigma(\tau) \mid \sigma_i \text{ for all } i = 1 \ldots m}{\Gamma \vdash e \texttt{ handle } n_1 \texttt{ -> } e_1 \texttt{ | } \ldots \texttt{ | } n_m \texttt{ -> } e_m) : \tau \mid \sigma_m \circ \ldots \circ \sigma_1 \circ \sigma}$$

Here is a sample execution:

```
# print_string (niceInfer_exp gather_exp_ty_substitution []
   (HandleExp(BinOpAppExp(ConcatOp, ConstExp(StringConst "What"),
                          RaiseExp(ConstExp(IntConst 3))),
              Some 0, ConstExp(StringConst " do you mean?"),
              [(None, ConstExp(StringConst " the heck?")) ])));;

  {} |= ("What" ^ (raise 3)) handle 0 => " do you mean?" | _ => " the heck?" : 'b
  |--{} |= "What" ^ (raise 3) : 'b
  | |--{} |= "What" : 'c
  | |--{} |= raise 3 : 'd
  |   |--{} |= 3 : int
  |--{} |= " do you mean?" : string
  |--{} |= " the heck?" : string

Unifying substitution: ['b --> string; 'd --> string; 'c --> string]
Substituting...
```

```
  {} |= ("What" ^ (raise 3)) handle 0 => " do you mean?" | _ => " the heck?" : string
  |--{} |= "What" ^ (raise 3) : string
  | |--{} |= "What" : string
  | |--{} |= raise 3 : string
  |   |--{} |= 3 : int
  |--{} |= " do you mean?" : string
  |--{} |= " the heck?" : string

- : unit = ()
```