MP 5 – A Unification-Based Type Inferencer

CS 421 – su 2011

Assigned July 3, 2011
Due July 10, 2011 23:59
Extension 48 hours (20% penalty)

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 `Mp5common`. You will be asked to implement the unification procedure in MP6.

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

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. Infer the types of all the subexpressions. For each subexpression, you will get back a proof tree and a list of constraints.
2. Create a new proof tree from the subexpressions.
3. Create a new set of constraints by taking the union of the constraints of the subexpressions. Add any new constraints to this.
4. Return the new proof tree and new set of constraints.

In a separate phase, using functions supplied by `Mp5common` we apply the set of constraints to the proof to finish inferring a type.

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 `mp5common.cmo` contains compiled code to support your construction of this type inferencer. Its contents are described here.

5.1 Preliminaries

Some data types and functions have been created for your convenience in order to complete this MP. One such data type is `nelist`. The `nelist` type is for holding polymorphic non-empty lists (lists of one or more elements).

```
type 'a nelist = {first : 'a; rest : 'a list}
```

We also provide you with mapping and folding functions to work over `nelist` data types.

```
val nelist_map : ('a -> 'b) -> 'a nelist -> 'b nelist = <fun>
val nelist_fold_right : ('a -> 'b -> 'b) -> 'b -> 'a nelist -> 'b = <fun>
val nelist_fold_left : ('a -> 'b -> 'a) -> 'a -> 'b nelist -> 'a = <fun>
val nelist_app : ('a -> 'b) -> ('a -> unit) -> 'a nelist -> unit = <fun>
val list_of_nelist : 'a nelist -> 'a list = <fun>
```

This assignment will also make use of the option type by returning values with `Some` or returning failures with `None`. You might find it helpful to apply a function to the value stored in an option's `Some` type. We provide you with `option_map` for this purpose.

```
val option_map : ('a -> 'b) -> 'a option -> 'b option = <fun>
```

`option_map` will take in a function and apply it to an option's `Some` value and return it in an option type. `None` is returned if the input is `None`.

5.2 OCaml Types for MicroML AST

Expressions and declarations in MicroML are almost identical to expressions and declarations in OCaml. The Abstract Syntax Trees for MicroML expressions and declarations are given by the following OCaml type:

```
type dec =
     Val of string * exp
   | Rec of (string * exp) nelist
   | Seq of dec * dec
   | Local of dec * dec

and exp =
   | VarExp of string
   | ConstExp of const
   | IfExp of exp * exp * exp
   | AppExp of exp * exp
   | FnExp of string pattern
   | LetExp of dec * exp
   | RaiseExp of exp
   | Handle of exp * (int option pattern nelist)

and 'a pattern = 'a * exp
```

The `exp` type makes use of the auxiliary type:

```
type const =
     BoolConst of bool
   | IntConst of int
   | RealConst of float
   | StringConst of string
   | NilConst
   | UnitConst
   | PrimOp of prim_op
```

for representing the constants in our language. This type may be expanded in future MPs in order to enrich the language.

The prim_op type represents primitive operators in SML. The only unary operator given is integer negation.

```
type prim_op = BinOp of bin_op | IntNeg
```

The primitive binary operators are given by the Ocaml data type bin_op.

```
type bin_op =
     IntPlusOp
   | IntMinusOp
   | IntTimesOp
   | RealPlusOp
   | RealMinusOp
   | RealTimesOp
   | ConcatOp
   | ConsOp
   | CommaOp
   | EqOp
   | GreaterOp
```

Some of the constructors of `exp` and `dec` should be self-explanatory. Names of constants are represented by the type `const`. Names of variables are represented by strings. Constructors that take `string` arguments (`VarExp`, `FnExp`, `Val`, and `Rec`) use the string to represent the names of variables that they bind. For instance, the `Val`

declaration constructor binds a string representing a variable to an expression. `Rec` is similar to `Val`, but it allows for recursive and mutually recursive bindings. A `Local` declaration introduces a scoped collection of bindings from the first declaration for use only in the second declaration. We have added in `RaiseExp` and `Handle` for raising and handling exceptions, but have limited exceptions to integers, rather in the style of Unix. Note that `LetExp` works in a similar way as the combination of Ocaml's `let ... in` and `let rec ... in`, but in a more general way since it makes use of declarations.

There are companion functions `print_exp` and `print_dec` that prints expression and declarations in a more readable form.

5.3 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. As a language, the types of MicroML are quite simple: type variables, type constants, and type constructors.

```
type tyExp =
     VarType of int
   | BoolType
   | IntType
   | RealType
   | StringType
   | UnitType
   | FunType of tyExp * tyExp
   | PairType of tyExp * tyExp
   | ListType of tyExp
   | OptionType of tyExp
```

Again, there is a companion function `print_tyExp` that prints expressions in a more readable form, similar to OCaml concrete syntax.

When inferring types, you will need to generate fresh type variable. For this, you may use the side-effecting function `fresh`, which takes a unit and returns a fresh type variable. The index stored by `fresh` (initially set to 0) will keep on growing as you use `fresh`. There is a companion function, `reset:unit -> unit`, that can be used to reset the internal index to 0. It should not be used as a part of any program in this assignment, but it may be useful in cleaning up when you are testing your code.

5.4 Environments

We need an environment to store the types of the variables. An environment is a list of pairs mapping variables to values. The environment is left polymorphic to handle variable mappings in future assignments. However, this assignment will require our environments to map variables to types.

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

You can interact with environments using the following functions, pre-defined in `mp5common.ml`:

```
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 insert_env : 'a env -> string -> 'a -> 'a env = <fun>
```

`make_env` takes in a variable as a string and a value and returns a new environment that contains the binding of the value to the string. `lookup_env` takes in an environment and a variable as a string and returns an option of `Some` of the variable's binding or `None` if the variable is not in the environment. `sum_env` takes in two environments and returns an environment with the second updated by the first. `insert_env` takes in an environment, a variable as a string, and a value and returns the given environment with the variable (re)bound to the given value.

5.5 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 primitive operators in MicroML. Unlike the environment, the signature will be fixed throughout type inference, and is provided by the following function: `const_signature : const -> tyExp`.

Some constants and operators, like `nil`, `cons`, and `,` have type variables in their associated type. This means that they are *polymorphic* constants, and during type inference their type variable α needs to be replaced with a fresh type variable - this would correspond to a use of these constants in a new context using their polymorphic nature. For instance, consider the expression

```
((true :: nil), (0 ::nil))
```

(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 :: nil)` and `(0 :: nil)`, separately:

1. When typing `(true :: nil)`, we discover that the operator `::` and constant `nil` are used with the types `bool -> bool list -> bool list` and `bool list`, respectively; this should be consistent with the types of `::` and `nil` stored in our signature, namely with respectively α `-> `α `list -> `α `list` and β `list` (where α and β are type variables, written in OCaml as `VarType 0` and `VarType 1`); the apparent constraints that need to be gathered here are α `-> `α `list -> `α `list = bool -> bool list -> bool list` and β `list = bool list`, which will eventually lead, by solving them, to α `= `β `= bool`;
2. Similarly, from typing `(0 :: nil)`, we get α `= `β `= 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 `nil` 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 (and β), to both `bool` and `int` - this is *not* a proper use of the polymorphic variable α, since α does not indicate a yet unknown, but fixed type; rather, α indicates a truly variable type, that can take *different values in different contexts*. The solution is to replace α and β with fresh type variables each time we type the operator `::` and the constant `nil`; this way, the constraints are: α_1 `list = bool list` and α_1 `-> `α_1 `list -> `α_1 `list = bool -> bool list -> bool list` from typing `(true :: nil)`, and α_2 `list = int list` and α_2 `-> `α_2 `list -> `α_2 `list = int -> int list -> int list` from typing `(0 ::nil)`, yielding the solution α_1 `= bool` and α_2 `= int`.

With the signatures for constants and built-in binary operators, we provide (a weak form of) 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.

5.6 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 structures:

```
type 'exp judgment_exp = { gamma_exp:'exp env; exp:exp; result: 'exp }
type 'exp judgment_dec = { gamma_dec:'exp env; dec:dec; result_env:'exp env }
type 'exp judgment =
    ExpJudgment of 'exp judgment_exp
  | DecJudgment of 'exp judgment_dec
```

```
type 'exp proof = {antecedents : 'exp proof list; conclusion : 'exp judgment}
```

The pre-defined functions `print_jexp`, `print_jdec`, and `print_proof` print readable forms of these typing judgments and proofs. The function `print_proof` prints a proof in a tree-like form, with the root at the top, upside-down from the way we are used to seeing proofs.

6 Type Inferencing

The rules used for a type-inferencer are very similar to the ones used in class. They have one extra component, a field for the constraints. Here's an example:

$$\frac{\Gamma \vdash e_1 \ : \texttt{int} \mid C_1 \quad \Gamma \vdash e_2 \ : \texttt{int} \mid C_2}{\Gamma \vdash e_1 + e_2 : \tau \mid \{\tau = \texttt{int}\} \cup C_1 \cup C_2}$$

The "|" is just some notation to separate the constraint from the expression. This rule says that the constraints sufficient to guarentee that the result of adding two expressions e_1 and e_2 will have type τ are the constraints C_1 sufficient to guarantee that e_1 has type `int` together with the constraints C_2 to guarantee that e_2 has type `int` , together with the additional constraint that $\{\tau = \texttt{int}\}$.

For example, suppose you want to infer the type of `fn x -> x + 2`. (We will use `fn` in MicroML, instead of `fun` as in OCaml.) 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 types of the subexpressions.
      i. Examine `x`. Γ says that `x` has type `'b`. We are trying to show it has type `int` . We generate the constraint $\{$`'b` $=$ `int` $\}$.
      ii. Examine `2`. This is an integer, as needed. (To be really thorough we would add the constraint $\{$`int` $=$ `int` $\}$.)
   (b) We combine these inferences together to make a new proof-tree, and add to the constraints that `'c` is `int`. We need to add this to the constraints that `'b` is type `int`, and `int` is type `int`. (Yes, that last one was trivial, but the rule says we have to do it. It will be removed later.)
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`, so the whole expression has type `'a` which must also be `'b -> 'c`. We must add this constraint as well.
5. Our constraints say to rewrite `'a` to `'b -> 'c` and rewrite `'b` and `'c` to `int` everywhere. We do that, and get a final type of `int -> int`.

Rules for types-inference of declarations are very similar. Instead of assigning a type for the rule, we have an environment that binds a type to the declared variable name:

$$\frac{\Gamma \vdash e : \tau \mid C}{\Gamma \vdash \texttt{val}\ \ x\ \ \texttt{=}\ \ e : \{x : \tau\} \mid C}$$

6.1 Pre-defined Functions

Some important functions are pre-defined: The function `infer_exp`, takes in a function `gather_exp_ty_constraints`, a type environment, and an expression and returns a `(tyExp * tyExp proof) option`. The first part of the result option type is the type of the entire expression and the second part is the proof (assuming success). The function `infer_dec`, takes in a function `gather_dec_ty_constraints`, a type environment, and a declaration and returns `(tyExp env * tyExp proof) option`. The first part of the result is environment giving the types of the declared values and the second part is the proof (assuming success).

`infer_exp` works by calling the function `gather_exp_ty_constraints` and gets back a (generic) proof tree and a set of constraints. If `gather_exp_ty_constraints` returns `None`, then `infer_exp` returns `None`. Otherwise, `infer_exp` attempts to unify the constraints. If this is successful, it gets a substitution. This substitution is applied all over the proof tree to obtain a concrete proof. The ultimate type as well as the proof are then returned in a `Some` of a pair. `None` is returned if unification fails.

Similarly, `infer_dec` works by calling the function `gather_dec_ty_constraints` and gets back a proof tree, a type environment, and a set of constraints. If `gather_dec_ty_constraints` returns `None`, then `infer_dec` returns `None`. Otherwise, `infer_dec` attempts to unify the constraints. If this is successful, it gets a substitution. This substitution is applied all over the proof tree to obtain a concrete proof. The ultimate type as well as the proof are then returned in a `Some` of a pair. `None` is returned if unification fails.

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

```
val infer_exp :
  (tyExp judgment_exp -> (tyExp proof * (tyExp * tyExp) list) option) ->
  tyExp env -> exp -> (tyExp * tyExp proof) option = <fun>
val infer_dec :
 (tyExp env -> dec ->
   (tyExp proof * tyExp env * (tyExp * tyExp) list) option) ->
  tyExp env -> dec -> (tyExp env * tyExp proof) option = <fun>

val get_ty : ('a * 'b) option -> 'a = <fun>
val get_proof : ('a * 'b) option -> 'b = <fun>
```

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

```
val niceInfer_exp :
  (tyExp judgment_exp -> (tyExp proof * (tyExp * tyExp) list) option) ->
  tyExp env -> exp -> unit = <fun>
val niceInfer_dec :
  (tyExp env -> dec ->
    (tyExp proof * tyExp env * (tyExp * tyExp) list) option) ->
  tyExp env -> dec -> unit = <fun>
```

that print out details about the constraints that are gathered, and the results of applying the substitutions created from the constraints to the original proof trees. 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`. The function you need to finish to gather expression type constraints is: `gather_exp_ty_constraints : tyExp judgment_exp -> (tyExp proof * (tyExp * tyExp) list) option`. The function `gather_exp_ty_constraints` takes in an expression judgment and returns `None` (on failure), or `Some` of a pair of a generic proof tree containing type variables, and a set of constraints to be unified. The function you need to finish to gather declaration type

constraints is: `gather_dec_ty_constraints : tyExp env -> dec -> (tyExp proof * tyExp env * (tyExp * tyExp) list) option`. The function `gather_dec_ty_constraints` takes in an environment and a declaration and returns `None` (on failure), or `Some` of a generic proof tree containing type variables, a type environment, and a set of constraints to be unified.

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

```
let rec gather_exp_ty_constraints judgment =
    let {gamma_exp = gamma; exp = exp; result = tau} = judgment in
    match exp
    with ConstExp c ->
      let tau' = const_signature c in
         Some ({antecedents = []; conclusion = ExpJudgment judgment}, [(tau, tau')])
```

This implements the rule

$\frac{}{\Gamma \vdash c : \tau \mid \{\tau = \tau'\}}$ 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_proof print_tyExp
              (get_proof(infer_exp gather_exp_ty_constraints
              []
              (ConstExp (BoolConst true))));;

  {  } |= true : bool

- : unit = ()
```

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

```
# niceInfer_exp gather_exp_ty_constraints [] (ConstExp (BoolConst true));;

  {  } |= true : 'b

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

  {  } |= true : bool

- : unit = ()
```

To help you with declarations, we will give you the clause for `gather_dec_ty_constraints` for a `Val` declaration:

```
and gather_dec_ty_constraints gamma dec =
    match dec with Val (v,e) ->
       let tau = fresh() in
       let env_inc = make_env v tau in
       option_map
       (fun (e_pf, c) ->
        ({antecedents=[e_pf];
         conclusion =
          DecJudgment {gamma_dec = gamma; dec = dec; result_env = env_inc}},
        env_inc,
        c))
        (gather_exp_ty_constraints {gamma_exp = gamma; exp = e; result = tau})
```

This implements the rule

$$\frac{\Gamma \vdash e : \tau \mid C}{\Gamma \vdash \texttt{val}\ x\ \texttt{=}\ e : \{x : \tau\} \mid C}$$

The second argument to `option_map`, `gather_exp_ty_constraints {gamma_exp = gamma; exp = e; result =` calculates the hypothesis $\Gamma \vdash e : \tau \mid C$. The first argument

```
(fun (e_pf, c) ->
 ({antecedents=[e_pf];
   conclusion =
   DecJudgment {gamma_dec = gamma; dec = dec; result_env = env_inc}},
  env_inc,
  c))
```

is a function that builds the full proof tree, the incremental environment, and the full constraints from the hypothesis and its constraints. The function `option_map` helps us focus on just what needs to be done in the case where the proof for the hypothesis actually can be built.

A sample execution would be:

```
# print_proof print_tyExp
              (get_proof(infer_dec gather_dec_ty_constraints
              []
              (Val ("x", (ConstExp (BoolConst true))))));;

  {  } |= val x = true :: { x : bool }
  |--{  } |= true : bool

- : unit = ()
```

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

```
# niceInfer_dec gather_dec_ty_constraints []
     (Val ("x", (ConstExp (BoolConst true))));;

  {  } |= val x = true :: { x : 'b }
  |--{  } |= true : 'b

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

  {  } |= val x = true :: { x : bool }
  |--{  } |= true : bool

- : unit = ()
```

It is not necessary for your work to generate exactly the same set of constraints that our solution gives. You may choose to examine your antecedents in a different order than the one we did, for example. What is required is that the type you get for an expressions must be an instance of the type the standard solution gets, and the type given by the standard solution must be and instance of the type you give. As a result, running `niceInfer_exp` or `niceInfer_dec` 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 variable expressions:

$$\frac{}{\Gamma \vdash x : \tau \mid \{\tau = \tau'\}} \text{ where } x \text{ is a program variable and } \Gamma(x) = \tau'$$

Note that $\Gamma(x)$ represents looking up the value of x in Γ. In OCaml, one writes `Mp5common.lookup_env gamma x` where `x` is the string naming the variable.

Here is a sample execution:

```
# print_proof print_tyExp
    (get_proof (infer_exp gather_exp_ty_constraints
                      (make_env "f" (FunType(BoolType, fresh())))
                      (VarExp "f")));;


  { f : bool -> 'b } |= f : bool -> 'b

- : unit = ()
```

2. (10 pts) Implement the rule for `if_then_else` expressions:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \mid C_1 \quad \Gamma \vdash e_2 : \tau \mid C_2 \quad \Gamma \vdash e : \tau \mid C_3}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau \mid C_1 \cup C_2 \cup C_3}$$

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_proof print_tyExp
     (get_proof (infer_exp gather_exp_ty_constraints
                      (make_env "x" (fresh()))
                      (IfExp (ConstExp (BoolConst true), ConstExp (IntConst 1),
                                VarExp "x"))));;

  { x : int } |= if true then 1 else x : int
  |--{ x : int } |= true : bool
  |--{ x : int } |= 1 : int
  |--{ x : int } |= x : int

- : unit = ()
```

3. (10 pts) Implement the application expression rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \mid C_1 \quad \Gamma \vdash e_2 : \tau_1 \mid C_2}{\Gamma \vdash e_1 \;\; e_2 : \tau \mid C_1 \cup C_2}$$

Here is a sample execution:

```
# print_proof print_tyExp
     (get_proof (infer_exp gather_exp_ty_constraints empty_env
                      (AppExp ((ConstExp(PrimOp(BinOp IntTimesOp))),
                               ConstExp (IntConst 3)))));;

  {  } |= (*)3 : int -> int
  |--{  } |= (*) : int -> int -> int
  |--{  } |= 3 : int

- : unit = ()
```

4. (10 pts) Implement the function expression rule:

$$\frac{\{x \to \tau_1\} + \Gamma \vdash e : \tau_2 \mid C}{\Gamma \vdash \texttt{fn}\ x\ \texttt{->}\ e : \tau \mid \{\tau = \tau_1 \to \tau_2\} \cup C}$$

Here is a sample execution:

```
# print_proof print_tyExp
     (get_proof (infer_exp gather_exp_ty_constraints empty_env
    (FnExp ("x", AppExp(AppExp((ConstExp(PrimOp(BinOp IntPlusOp))),
                                                VarExp "x"),
                                    VarExp "x")))));;

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

- : unit = ()
```

5. (5 pts) Implement the Raise expression rule:

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

Here is a sample execution:

```
# print_proof print_tyExp
     (get_proof (infer_exp gather_exp_ty_constraints empty_env
      (RaiseExp (ConstExp (IntConst 3)))));;

  {  } |= raise 3 : 'b
  |--{  } |= 3 : int

- : unit = ()
```

6. (10 pts) Implement the Let expression rule:

$$\frac{\Gamma \vdash dec : \Delta \mid C_1 \quad \Delta + \Gamma \vdash e : \tau \mid C_2}{\Gamma \vdash \texttt{let}\ dec\ \texttt{in}\ e\ \texttt{end} : \tau \mid C_1 \cup C_2}$$

Here is a sample execution:

```
# print_proof print_tyExp
     (get_proof (infer_exp gather_exp_ty_constraints empty_env
      (LetExp(Val("x", ConstExp(IntConst 6)), ConstExp (NilConst)))));;

  {  } |= let val x = 6 in [] end : 'd list
  |--{  } |= val x = 6 :: { x : int }
  | |--{  } |= 6 : int
  |--{ x : int } |= [] : 'd list

- : unit = ()
```

7. (10 pts) Implement the Sequence declaration rule:

$$\frac{\Gamma \vdash dec_1 : \Delta_1 \mid C_1 \quad \Delta_1 + \Gamma \vdash dec_2 : \Delta_2 \mid C_2}{\Gamma \vdash dec_1 \;\; dec_2 : \Delta_1 + \Delta_2 \mid C_1 \cup C_2}$$

Here is a sample execution:

```
# print_proof print_tyExp
              (get_proof(infer_dec gather_dec_ty_constraints
              []
              (Seq ((Val ("x", (ConstExp (BoolConst true)))),
              (Val ("y", (ConstExp (IntConst 2)))))))));;

  {  } |= val x = true
val y = 2 :: { x : bool,y : int }
  |--{  } |= val x = true :: { x : bool }
  | |--{  } |= true : bool
  |--{ x : bool } |= val y = 2 :: { y : int }
    |--{ x : bool } |= 2 : int

- : unit = ()
```

8. (10 pts) Implement the Local declaration rule:

$$\frac{\Gamma \vdash dec_1 : \Delta_1 \mid C_1 \quad \Delta_1 + \Gamma \vdash dec_2 : \Delta_2 \mid C_2}{\Gamma \vdash \texttt{local} \;\; dec_1 \;\; \texttt{in} \;\; dec_2 : \Delta_2 \mid C_1 \cup C_2}$$

Here is a sample execution:

```
# print_proof print_tyExp
              (get_proof(infer_dec gather_dec_ty_constraints
              []
              (Local ((Val ("x", (ConstExp (BoolConst true)))),
              (Val ("y", (VarExp "x")  ))))));;

  {  } |= local val x = true
in val y = x end :: { y : bool }
  |--{  } |= val x = true :: { x : bool }
  | |--{  } |= true : bool
  |--{ x : bool } |= val y = x :: { y : bool }
    |--{ x : bool } |= x : bool

- : unit = ()
```

9. (15 pts) Implement the Recursive declaration rule:

$$\frac{\{x_1 : \tau_1\} + ... + \{x_n : \tau_n\} + \Gamma \vdash e_i : \tau_i \mid C_i \;\text{ for all } i = 1 \ldots n}{\Gamma \vdash \texttt{val rec} \;\; x_1 = e_1 \;\; \texttt{and} \;\; ... \;\; \texttt{and} \;\; x_n = e_n : \{x_1 : \tau_1\} + ... + \{x_n : \tau_n\} \mid C_1 \cup ... \cup C_n}$$

Here is a sample execution:

```
# print_proof print_tyExp
              (get_proof(infer_dec gather_dec_ty_constraints
              []
              (Rec{first=("ones", AppExp(AppExp((ConstExp(PrimOp(BinOp ConsOp))),
              ConstExp(IntConst 1)), VarExp "ones"));rest=[]}) ));;

  {  } |= val rec ones = 1 :: ones :: { ones : int list }
  |--{ ones : int list } |= 1 :: ones : int list
    |--{ ones : int list } |= (::)1 : int list -> int list
    | |--{ ones : int list } |= (::) : int -> int list -> int list
    | |--{ ones : int list } |= 1 : int
    |--{ ones : int list } |= ones : int list

- : unit = ()
```

8 Extra Credit

10. (7 pts) Implement the Handle expression rule:

$$\frac{\Gamma \vdash e : \tau \mid C \quad \Gamma \vdash e_i : \tau \mid C_i \text{ for all } i = 1 \ldots m}{\Gamma \vdash \texttt{(handle } e \texttt{ with } n_1 \texttt{ -> } e_1 \texttt{ | ... | } n_m \texttt{ -> } e_m\texttt{)} : \tau \mid C \cup \bigcup_{i=1}^{m} C_i}$$

Here is a sample execution:

```
# print_proof print_tyExp
     (get_proof (infer_exp gather_exp_ty_constraints empty_env
     (Handle(AppExp (AppExp((ConstExp(PrimOp(BinOp ConcatOp))),
                            ConstExp (StringConst "What")),
                     RaiseExp (ConstExp (IntConst 3))),
               {first = (Some 0, ConstExp (StringConst " do you mean?"));
                rest=[(None, ConstExp (StringConst " the heck?"))]}))));;


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

- : unit = ()
```