MP 5 – A Unification-Based Type Inferencer

CS 421 – Summer 2013

Revision 1.1

Assigned Jun 20, 2013
Due Jun 28, 2013, 11:59 PM
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Added description of type-constructing helper functions.

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 typing rules.
- 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`.

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

4.1 Type Inference Overview

Recall the four main steps of type inference:

1. Recursively infer the types of all 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 will build an interpreter, which we call PicoML, is mainly a simplification of OCaml. In this assignment we will build a type inferencer for expressions in PicoML. The file `mp5common.cmo` contains compiled code to support your construction of this type inferencer. Its contents are described here.

5.1 OCaml Types for PicoML AST

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

```
type exp =
   | VarExp of string               (* variables *)
   | ConExp of const                (* constants *)
   | IfExp of exp * exp * exp       (* if exp1 then exp2 else exp3 *)
   | AppExp of exp * exp            (* exp1 exp2 *)
   | BinExp of binop * exp * exp    (* exp1 % exp2
                                       where % is a builtin binary operator *)
   | MonExp of monop * exp          (* % exp1
                                       where % is a builtin monadic operator *)
   | FunExp of string * exp         (* fun x -> exp *)
   | LetExp of string * exp * exp   (* let x = exp1 in exp2 *)
   | RecExp of string * string * exp * exp    (* let rec f x = exp1 in exp2 *)
   | RaiseExp of exp                (* raise exp *)
   | TryWithExp of exp * (int option * exp) * ((int option * exp) list)
                                 (* try exp with n1 -> exp1 | ... | nm -> expm *)
```

This type makes use of the auxiliary types:

```
type const = Bool of bool | Int of int | Float of float | String of string
           | Nil | Unit

type binop =
     Add | Sub | Mul | FAdd | FSub | FMul | Concat | Cons | Comma | Eq | Less

type monop = Head | Tail | Print | Neg
```

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

The constructors of `exp` should be self-explanatory. Names of constants are represented by the type `const`. Names of variables are represented by strings. `BinExp` takes the binary operator, represented by the type `binop`, together with two operands. Similarly, `MonExp` takes the unary operator of the `monop` type and an operand. The constructors that take `string` arguments (`VarExp`, `FunExp`, `LetExp`, and `RecExp`) use the string to represent names of variables that they bind. We have added in `RaiseExp` and `TryWithExp` for raising and handling exceptions, but have limited exceptions to integers, rather in the style of Unix.

There is a companion function `string_of_exp` that converts expressions into strings in a more readable form, similar to OCaml concrete syntax.

5.2 OCaml Types for PicoML Types

In addition to having abstract syntax trees for the expressions of PicoML, we need to have abstract syntax trees for the types of PicoML. The types of PicoML can be categorized into two kinds: monomorphic types and polymorphic types. Monomorphic types are either type variables, type constants, or type constructors applied to a sequence of types. To make types more 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 PicoML:

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

We will represent type constructors generically by a name and an arity (number of arguments):

```
type constTy = {name : string; arity : int}
```

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. We've provided some functions for making the most common kinds of types:

```
let bool_ty = TyConst({name="bool"; arity = 0},[])
let int_ty = TyConst ({name="int"; arity = 0}, [])
let float_ty = TyConst ({name="float"; arity = 0},[])
let string_ty = TyConst ({name="string"; arity = 0},[])
let unit_ty = TyConst({name="unit"; arity = 0}, [])
let mk_pair_ty ty1 ty2 = TyConst({name="*"; arity = 2},[ty1;ty2])
let mk_fun_ty ty1 ty2 = TyConst({name="->"; arity = 2},[ty1;ty2])
let mk_list_ty ty = TyConst({name="list"; arity = 1},[ty])
```

Polymorphic types in PicoML 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. An environment is a list of pairs mapping variables (strings) to values (polymorphic types):

```
type env = (string * polyTy) list
```

One interacts with environments using the following functions, pre-defined in `mp5common.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 -> polyTy -> env = <fun>
val lookup_env : env -> string -> polyTy option = <fun>
val sum_env : env -> env -> env = <fun>
val ins_env : env -> string -> polyTy -> env = <fun>
```

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 PicoML. Unlike the environment, these signatures will be fixed throughout type inference, and are 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 `Nil`, `Cons`, and `Comma` 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 for the quantified variables with fresh ones. 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 will eventually have to type both its immediate subexpressions, `(true :: [])` and `(0 :: [])`, separately:

1. When typing `(true :: [])`, 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 `[]` stored in our signature, which are $\forall\alpha.$ α `->` α `list ->` α `list` and $\forall\alpha.$ α `list` respectively (where α is a type variable, written in our system as `TyVar 0`). In order for `::` to be applied to `0` (and `Nil`), we need to replace all instances of α (in both types) with `bool`.

2. Similarly, from typing `(0 :: [])`, we find that we need to replace all instances of α with `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 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 universal type that can take *different values in different contexts*. 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`. We wrote this process as a function `freshInstance` in the presentation of the type inference algorithm in lecture.

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 `Nil` and `Cons`, every time their type is requested from the appropriate signature, a new type with fresh type variables is given. To be able to use different instance of these polymorphic types, as well as those occurring in out 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 a type judgment has the form $\Gamma \vdash e : \tau$. This says that in the environment Γ, the expression e has type τ. 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 judgment = { gamma:env; exp:exp; monoTy:monoTy }

type proof = {antecedents : proof list; conclusion : 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 a string for a proof tree, which displays similarly to the proof trees shown in class, although upside-down, with the root at the top.

5.6 Substitutions

In this MP, you are asked to write code returning, among other things, a substitution. A substitution is a partial mapping from variables to expressions. In our case, we will use substitutions from type variables to monomorphic types. As with environments, we represent substitutions as lists:

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

Just as with environments, substitutions provide a collection of functions to manipulate them. To apply a substitution to each of monomorphic types, polymorphic 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 unifying a list of constraints implied by the rule you are implementing. A constraint is represented by a pair of monomorphic types, indicating that you need to make the two types 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, as we expect from a unification algorithm.

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 : env -> monoTy -> polyTy
```

6 Type Inference

The rules used for type inference 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. For 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 "|" separates the substitution from the expression. 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` , and the substitution generated from the constraint $\{\tau = \texttt{int} \ \}$.

6.1 Pre-defined Functions

Some important functions for testing your code are pre-defined. The function `infer` takes in a function `gather_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 (as long as type inference succeeds).

`infer` works by generating a fresh type variable τ and calling the function `gather_ty_substitution` and gets back a (generic) proof tree and a substitution. If `gather_ty_substitution` returns `None`, then `infer` returns `None`. Otherwise, `infer` 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.

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

```
val infer :
  (judgment -> (proof * substitution) option) ->
  env -> exp -> (monoTy * proof) option
val get_ty : ('a * 'b) option -> 'a = <fun>
val get_proof : ('a * 'b) option -> 'b = <fun>
```

There is also a verbose form of `infer`

```
val niceInfer :
  (judgment -> (proof * substitution) option) -> env -> exp -> 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 body of the main type inferencing function, `infer`, is already implemented. Your task is to finish the implementation of the main function needed by `infer`. The function `gather_ty_substitution : judgment -> (proof * substitution) option` takes in a judgment and returns `None` (on failure), or `Some` of a pair of a generic proof tree containing type variables, and a substitution.

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

```
let rec gather_ty_substitution judgment =
    let {gamma = gamma; exp = exp; monoTy = tau} = judgment in
    match exp
    with ConExp c ->
      let tau' = const_signature c in
      (match unify [(tau, freshInstance(tau'))] with None -> None
       | Some sigma ->
         Some ({antecedents = []; conclusion = judgment}, sigma))
```

This implements the rule

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

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 gather_ty_substitution [] (ConExp (Bool true))
  )));;

  {} |= true : bool
```

To see what happened in greater detail, we may run:

```
# print_string (niceInfer gather_ty_substitution [] (ConExp(Bool 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 is probably not the only way.

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

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

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

   A sample execution is

```
# print_string (niceInfer gather_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. (5 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 gather_ty_substitution []
                (BinExp(Cons, ConExp (Int 62), ConExp Nil)));;

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

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

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

# print_string (niceInfer gather_ty_substitution []
                (MonExp(Print, ConExp(Int 62))));;

  {} |= print_int 62 : 'b
  |--{} |= 62 : 'c

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

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

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 gather_ty_substitution []
    (IfExp(ConExp(Bool true), ConExp(Int 62), ConExp(Int 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
```

4. (10pts) Implement the function rule:

$$\frac{[x:\tau_1]+\Gamma \vdash e:\tau_2 \mid \sigma}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e:\tau \mid \mathit{unify}(\sigma(\tau),\ \sigma(\tau_1 \rightarrow \tau_2)) \circ \sigma}$$

Here is a sample execution:

```
# print_string (niceInfer gather_ty_substitution []
    (FunExp("x", BinExp(Add, VarExp "x", VarExp "x"))));;

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

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

  {} |= fun x -> x  +  x : int -> int
  |--{x : int} |= x  +  x : int
    |--{x : int} |= x : int
    |--{x : int} |= x : int
```

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

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

Here is a sample execution:

```
# print_string (niceInfer gather_ty_substitution []
  (AppExp(FunExp("x", BinExp(Add, VarExp "x", VarExp "x")),
          ConExp(Int 62))));;

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

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

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

6. (10 pts) Implement the `let_in_` rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \sigma_1 \quad [x : \mathit{GEN}(\sigma_1(\Gamma), \sigma_1(\tau_1))] + \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau) \mid \sigma_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau \mid \sigma_2 \circ \sigma_1}$$

Here is a sample execution:

```
# print_string (niceInfer gather_ty_substitution []
    (LetExp("y", ConExp(Int 5), BinExp(Add, VarExp "y", VarExp "y"))));;

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

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

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

7. (10 pts) Implement the `let_rec_in_` rule:

$$\frac{[f:\tau_1 \rightarrow \tau_2] + ([x:\tau_1] + \Gamma) \vdash e_1 : \tau_2 \mid \sigma_1 \quad [f: \mathit{GEN}(\sigma_1(\Gamma), \sigma_1(\tau_1 \rightarrow \tau_2))] + \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau) \mid \sigma_2}{\Gamma \vdash \texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2 : \tau \mid \sigma_2 \circ \sigma_1}$$

Here is a sample execution:

```
# print_string (niceInfer gather_ty_substitution [] (
   RecExp("fact", "n",
      IfExp(
         BinExp(Eq, VarExp "n", ConExp(Int 0)),
         ConExp(Int 1),
         BinExp(Mul,
            VarExp "n",
            AppExp(VarExp "fact", BinExp(Sub, VarExp "n", ConExp(Int 1))))),
      AppExp(VarExp "fact", ConExp(Int 5)))));;

  {} |= let rec fact n = if n  =  0 then 1 else n  *  (fact (n  -  1)) in fact 5 : 'b
  |--{fact : 'd -> 'c,n : 'd} |= if n  =  0 then 1 else n  *  (fact (n  -  1)) : 'c
  | |--{fact : 'd -> 'c,n : 'd} |= n  =  0 : bool
  | | |--{fact : 'd -> 'c,n : 'd} |= n : 'f
  | | |--{fact : 'd -> 'c,n : 'd} |= 0 : 'e
  | |--{fact : int -> 'c,n : int} |= 1 : 'c
  | |--{fact : int -> int,n : int} |= n  *  (fact (n  -  1)) : int
  |   |--{fact : int -> int,n : int} |= n : 'h
  |   |--{fact : int -> int,n : int} |= fact (n  -  1) : 'g
  |     |--{fact : int -> int,n : int} |= fact : 'i -> 'g
  |     |--{fact : int -> int,n : int} |= n  -  1 : int
  |       |--{fact : int -> int,n : int} |= n : 'k
  |       |--{fact : int -> int,n : int} |= 1 : 'j
  |--{fact : int -> int} |= fact 5 : 'b
    |--{fact : int -> int} |= fact : 'l -> 'b
    |--{fact : int -> int} |= 5 : int

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

  {} |= let rec fact n = if n  =  0 then 1 else n  *  (fact (n  -  1)) in fact 5 : int
  |--{fact : int -> int,n : int} |= if n  =  0 then 1 else n  *  (fact (n  -  1)) : int
  | |--{fact : int -> int,n : int} |= n  =  0 : bool
  | | |--{fact : int -> int,n : int} |= n : int
  | | |--{fact : int -> int,n : int} |= 0 : int
  | |--{fact : int -> int,n : int} |= 1 : int
  | |--{fact : int -> int,n : int} |= n  *  (fact (n  -  1)) : int
  |   |--{fact : int -> int,n : int} |= n : int
  |   |--{fact : int -> int,n : int} |= fact (n  -  1) : int
  |     |--{fact : int -> int,n : int} |= fact : int -> int
  |     |--{fact : int -> int,n : int} |= n  -  1 : int
  |       |--{fact : int -> int,n : int} |= n : int
  |       |--{fact : int -> int,n : int} |= 1 : int
```

```
  |--{fact : int -> int} |= fact 5 : int
    |--{fact : int -> int} |= fact : int -> int
    |--{fact : int -> int} |= 5 : int
```

8. (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 gather_ty_substitution []
  (RaiseExp(IfExp(ConExp(Bool true), ConExp(Int 62), ConExp(Int 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
```

7.1 Extra Credit

9. (7 pts) Implement the rule for handling exceptions with `try_with`:

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

Here is a sample execution:

```
# print_string (niceInfer gather_ty_substitution [] (
   TryWithExp(
      BinExp(Concat, ConExp(String "What"), RaiseExp(ConExp(Int 3))),
      (Some 0, ConExp(String " do you mean?")),
      [
         (None, ConExp(String " in the world?"))
      ])));;

  {} |= (try "What"  ^  (raise 3) with  -> " do you mean?"|  -> " the heck?") : 'b
  |--{} |= "What"  ^  (raise 3) : 'b
  | |--{} |= "What" : 'd
  | |--{} |= raise 3 : 'c
```

```
  |   |--{} |= 3 : int
  |--{} |= " do you mean?" : string
  |--{} |= " in the world?" : string

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

  {} |= (try "What"  ^  (raise 3) with  -> " do you mean?"|  -> " in the world?")
        : string
  |--{} |= "What"  ^  (raise 3) : string
  | |--{} |= "What" : string
  | |--{} |= raise 3 : string
  |   |--{} |= 3 : int
  |--{} |= " do you mean?" : string
  |--{} |= " in the world?" : string
```