
MP 5 – A Unification-Based Type Inferencer

CS 421 – Fall 2009

Revision 1.0

Assigned Oct 6, 2009

Due Oct 12, 2009 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 interpreter¹. 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.

¹A 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 PicoML, is mainly a simplification of OCaml. In this assignment we shall 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
  | ConstExp of const
  | BinOpExp of string
  | IfExp of exp * exp * exp
  | AppExp of exp * exp
  | FunExp of string * exp
  | LetInExp of string * exp * exp
  | LetRecInExp of string * exp * exp
  | RaiseExp of exp
  | TryWithExp of exp * (int option * exp) * ((int option * exp) list)
```

This type makes use of the auxiliary type:

```
type const =
  BoolConst of bool
  | IntConst of int
  | FloatConst of float
  | StringConst of string
  | NilConst
```

for representing the constants in our language. This type 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. With the exception of `BinOp`, the constructors that take

string arguments (`VarExp`, `Fun`, `LetIn`, and `LetRecIn`) use the string to represent names of variables that they bind. `BinOp` takes the name of the binary operator. We have added in `Raise` and `TryWith` for raising and handling exceptions, but have limited exceptions to integers, rather in the style of Unix.

There is a companion function `print_exp` that prints expressions 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. As a language, the types of PicoML are quite simple: type variables, type constants, and type constructors applied to a sequence of types. To make types even more uniform, we will consider type constants as type constructors applied to an empty sequence of types. Thus we may use the OCaml type:

```
| type expType = TyVar of int | TyConst of (constTy * expType list)
```

We will represent type constructors generically by a name and an arity:

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

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

When inferring types, you will need to generate fresh type-variable names. For this, you may use the side-effecting function `fresh` that 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`.

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 (types):

```
| type env = (string * expType) 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 -> expType -> env = <fun>
val lookup_env : env -> string -> expType option = <fun>
val sum_env : env -> env -> env = <fun>
val ins_env : env -> string -> expType -> 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 binary operators in PicoML. Unlike the environment, the signature will be fixed throughout type inference, and is provided here as a pair of functions `const_signature : const -> expType` and `bin_op_signature` taking respectively a `const` or `bin_op` and returning a `expType`.

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 $\alpha \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list}$ and $\beta \text{ list}$ (where α and β are type variables, written in OCaml as `TyVar 0` and `TyVar 1`); the apparent constraints that need to be gathered here are $\alpha \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list} = \text{bool} \rightarrow \text{bool list} \rightarrow \text{bool list}$ and $\beta \text{ list} = \text{bool list}$, which will eventually lead, by solving them, to $\alpha = \beta = \text{bool}$;
2. Similarly, from typing `(0 :: nil)`, we get $\alpha = \beta = \text{int}$.

The two constraints, $\alpha = \text{bool}$ and $\alpha = \text{int}$, 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 an 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: $\alpha_1 \text{ list} = \text{bool list}$ and $\alpha_1 \rightarrow \alpha_1 \text{ list} \rightarrow \alpha_1 \text{ list} = \text{bool} \rightarrow \text{bool list} \rightarrow \text{bool list}$ from typing `(true :: nil)`, and $\alpha_2 \text{ list} = \text{int list}$ and $\alpha_2 \rightarrow \alpha_2 \text{ list} \rightarrow \alpha_2 \text{ list} = \text{int} \rightarrow \text{int list} \rightarrow \text{int list}$ from typing `(0 :: nil)`, yielding the solution $\alpha_1 = \text{bool}$ and $\alpha_2 = \text{int}$.

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 `Cons`, 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({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 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])

let const_signature = function
  BoolConst b -> bool_ty
  | IntConst n -> int_ty
  | FloatConst f -> float_ty
  | StringConst s -> string_ty
  | NilConst -> mk_list_ty (fresh())

let int_op_ty = mk_fun_ty int_ty (mk_fun_ty int_ty int_ty)
let float_op_ty = mk_fun_ty float_ty (mk_fun_ty float_ty float_ty)
let string_op_ty = mk_fun_ty string_ty (mk_fun_ty string_ty string_ty)

let bin_op_signature = function
  "+" -> int_op_ty
```

```

| "-"  -> int_op_ty
| "*"  -> int_op_ty
| "+."  -> float_op_ty
| "-."  -> float_op_ty
| "*."  -> float_op_ty
| "^"  -> string_op_ty
| "::" ->
    let alpha = fresh() in
      mk_fun_ty alpha (mk_fun_ty (mk_list_ty alpha) (mk_list_ty alpha))
| ", " ->
    let alpha = fresh() in
    let beta = fresh() in
      mk_fun_ty alpha (mk_fun_ty beta (mk_pair_ty alpha beta))
| "=" -> mk_fun_ty int_ty (mk_fun_ty int_ty bool_ty)
| ">" -> let alpha = fresh() in mk_fun_ty alpha (mk_fun_ty alpha bool_ty)

```

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 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; expType:expType }
type proof = { antecedents : proof list; conclusion : judgment }

```

The pre-defined functions `print_jexp` 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:

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

The “|” is just some notation to separate the constraint from the expression. This rule says that the constraints sufficient to guarantee 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 the $\{\tau = \text{int}\}$.

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

1. Let $\Gamma = \{\}$.
2. We start with a “guess” that `fun x -> x + 2` has type τ .
3. Next we examine `fun x -> x + 2` and see that it is a `fun`. We don't know what x will be, so we let it have type τ . Add that to Γ and try to infer the type of the body is τ ...
 - (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 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`.

6.1 Pre-defined Functions

Some important functions are pre-defined: The function `infer`, takes in a function `gather_ty_constraints` and an `exp` and returns an `(expType * 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` works by calling the function `gather_ty_constraints` and gets back a (generic) proof tree and a set of constraints. If `gather_ty_constraints` returns `None`, then `infer` returns `None`. Otherwise, `infer` 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 :
  (judgment -> (proof * (expType * expType) list) option) ->
  env -> exp -> (expType * proof) option = <fun>
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 * (expType * expType) list) option) ->
  env -> exp -> unit = <fun>
```

that prints out details about the constraint that is gathered, and the results of applying the substitution created from the constraints to the original proof tree. 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`: `gather_ty_constraints : judgment -> (proof * (expType * expType) list) 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 set of constraints to be unified.

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

```

let rec gather_ty_constraints judgment =
  let {gamma = gamma; exp = exp; expType = expType} = judgment in
  match exp
  with ConstExp c ->
    let c_ty = const_signature c in
    Some ({antecedents = []; conclusion = judgment}, [(expType, c_ty)])

```

This implements the rule

$\overline{\Gamma \vdash c : \tau \mid \{\tau = \tau'\}}$ 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(get_proof(infer gather_ty_constraints [] (ConstExp (BoolConst true))));;

{} |= true : bool
- : unit = ()

```

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

```

# niceInfer gather_ty_constraints [] (ConstExp (BoolConst true));;

{} |= true : 'b
Constraints: ['b --> bool; ]
Unifying...Unifying substitution: ['b --> bool; ]
Substituting...

{} |= 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 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:

$\overline{\Gamma \vdash x : \tau \mid \{\tau = \tau'\}}$ where x 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.

A sample execution is

```

# niceInfer gather_ty_constraints
  (make_env "f" (mk_fun_ty bool_ty (fresh())))
  (VarExp "f");;

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

```

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

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

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

2. (5 pts) Implement the rule for built-in binary operators:

$$\frac{}{\Gamma \vdash op : \tau \mid \{\tau = \tau'\}}$$

where op is a built-in binary operator, and τ' is an instance of the type assigned by the signature for built-in operator.

A sample execution would be:

```
# niceInfer gather_ty_constraints [] (BinOpExp "::");;
```

```
{ } |= ( :: ) : 'b
```

```
Constraints: ['b --> 'c -> 'c list -> 'c list; ]
Unifying...Unifying substitution: ['b --> 'c -> 'c list -> 'c list; ]
Substituting...
```

```
{ } |= ( :: ) : 'c -> 'c list -> 'c list
```

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

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

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

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:

```
# niceInfer gather_ty_constraints
  (make_env "x" (fresh()))
  (IfExp (ConstExp (BoolConst true), ConstExp (IntConst 1), VarExp "x"));;
```

```
{x : 'b} |= if true then 1 else x : 'c
```

```
|--{x : 'b} |= true : bool
```

```
|--{x : 'b} |= 1 : 'c
```

```
|--{x : 'b} |= x : 'c
```

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

```

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

- : unit = ()

```

4. (10 pts) Implement the rule for application:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash e_1 e_2 : \tau \mid \{\tau_1 = \tau_2 \rightarrow \tau\} \cup C_1 \cup C_2}$$

Here is a sample execution:

```

# niceInfer gather_ty_constraints [] (AppExp (BinOpExp "*", ConstExp (IntConst 3))));

{} |= (( * )) (3) : 'b
|--{} |= ( * ) : 'd
|--{} |= 3 : 'c

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

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

- : unit = ()

```

5. (10pts) Implement the function rule:

$$\frac{\{x \rightarrow \tau_1\} + \Gamma \vdash e : \tau_2 \mid C}{\Gamma \vdash \text{fun } x \rightarrow e : \tau \mid \{\tau = \tau_1 \rightarrow \tau_2\} \cup C}$$

Here is a sample execution:

```

# niceInfer gather_ty_constraints []
  (FunExp ("x", AppExp (AppExp (BinOpExp "+", VarExp "x"), VarExp "x")));

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

Constraints: ['b --> 'd -> 'c; 'f --> 'e -> 'c; 'h --> 'g -> 'f; 'h

```

```

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

```

```

{} |= (fun 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 = ()
```

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \{x \rightarrow \tau_1\} + \Gamma \vdash e : \tau \mid C_2}{\Gamma \vdash \text{let } x=e_1 \text{ in } e_2 : \tau \mid C_1 \cup C_2}$$

Here is a sample execution:

```

# niceInfer gather_ty_constraints []
(LetInExp("y", ConstExp(IntConst 5),
  AppExp(AppExp(BinOpExp "+", VarExp "y"), VarExp "y")));;

```

```

{} |= let y = 5 in (y + y) : 'b
|--{} |= 5 : 'c
|--{y : 'c} |= (y + y) : 'b
  |--{y : 'c} |= (( + ))(y) : 'e
  | |--{y : 'c} |= ( + ) : 'g
  | |--{y : 'c} |= y : 'f
  |--{y : 'c} |= y : 'd

```

```

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

```

```

{} |= let y = 5 in (y + y) : int
|--{} |= 5 : int
|--{y : int} |= (y + y) : int
  |--{y : int} |= (( + ))(y) : int -> int
  | |--{y : int} |= ( + ) : int -> int -> int
  | |--{y : int} |= y : int
  |--{y : int} |= y : int

```

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

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

$$\frac{\{x \rightarrow \tau_1\} + \Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \{x \rightarrow \tau_1\} + \Gamma \vdash e : \tau \mid C_2}{\Gamma \vdash \text{let rec } x=e_1 \text{ in } e_2 : \tau \mid C_1 \cup C_2}$$

Here is a sample execution:

```
# niceInfer gather_ty_constraints []
  (LetRecInExp ("ones",
    AppExp (AppExp (BinOpExp ":", ConstExp (IntConst 1)), VarExp "ones"),
    VarExp "ones"));

{} |= let rec ones = (1 :: ones) in ones : 'b
|--{ones : 'c} |= (1 :: ones) : 'c
| |--{ones : 'c} |= (( :: ))(1) : 'e
| | |--{ones : 'c} |= ( :: ) : 'g
| | |--{ones : 'c} |= 1 : 'f
| |--{ones : 'c} |= ones : 'd
|--{ones : 'c} |= ones : 'b

Constraints: ['e --> 'd -> 'c; 'g --> 'f -> 'e; 'g --> 'h -> 'h list -> 'h list; 'f -->
Unifying...Unifying substitution: ['e --> int list -> int list; 'g --> int -> int list -
Substituting...

{} |= let rec ones = (1 :: ones) in 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
|--{ones : int list} |= ones : int list

- : unit = ()
```

8. (7 pts) Implement the rule for `raise`

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

Here is a sample execution:

```
# niceInfer gather_ty_constraints []
  (RaiseExp (IfExp (ConstExp (BoolConst true), ConstExp (IntConst 3),
    ConstExp (IntConst 4))));

{} |= raise if true then 3 else 4 : 'b
|--{} |= if true then 3 else 4 : int
  |--{} |= true : bool
  |--{} |= 3 : int
  |--{} |= 4 : int

Constraints: [bool --> bool; int --> int; int --> int; ]
Unifying...Unifying substitution: []
```

Substituting...

```
{ } |= raise if true then 3 else 4 : 'b
|--{ } |= if true then 3 else 4 : int
  |--{ } |= true : bool
  |--{ } |= 3 : int
  |--{ } |= 4 : int

- : unit = ()
```

7.1 Extra Credit

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

$$\frac{\Gamma \vdash e : \tau \mid C \quad \Gamma \vdash e_i : \tau \mid C_i \text{ for all } i = 1 \dots m}{\Gamma \vdash (\text{try } e \text{ with } n_1 \rightarrow e_1 \mid \dots \mid n_m \rightarrow e_m) : \tau \mid C \cup \bigcup_{i=1}^m C_i}$$

Here is a sample execution:

```
# niceInfer gather_ty_constraints []
  (TryWithExp (AppExp (AppExp (BinOpExp "^",
                              ConstExp (StringConst "What")),
                              RaiseExp (ConstExp (IntConst 3))),
              (Some 0, ConstExp (StringConst " do you mean?")),
              [(None, ConstExp (StringConst " the heck?"))]));;

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

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

{ } |= try ("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 = ()
```