MP 6 – A Unification-based Type Inferencer

CS 421

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Objectives

Your objectives are:

- Understand the details of the basic algorithm for first order unification.
- Understand the type-inference algorithm.

3 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 / compiler. 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.

One of the most important pre-processing steps performed in statically typed languages is the type checking / type inferencing phase, checking whether the program has been typed correctly / can be typed correctly.

In this MP, you will be implementing most of the functions required to perform type inferencing using unification.

4 Given Code

We will be performing type inferencing on a language called PicoML, which is a small subset / simplification of OCaml. The subset of the syntax we'll be implementing is interesting enough to cover most interesting aspects of Hindley-Milner type inferencing, eg polymorphism, generalization, type instantiation, etc.

4.1 AST

The abstract syntax tree for PicoML expressions is given as follows within Common.hs.

```
data Exp =
  ConstExp Const
  | VarExp String
  | MonOpExp Monop Exp
  | BinOpExp Binop Exp Exp
  | IfExp Exp Exp Exp
  | AppExp Exp Exp
  | FunExp String Exp
  | LetExp String Exp Exp
  | LetRecExp String String Exp Exp
```

Notice that within the AST we use strings to represent variable dereferences and variable bindings. (Our subset of OCaml does not have any form of pattern matching.)

It makes use of the following auxiliary types to represent different types of constants, unary (monop) expressions, and binary expressions.

```
data Const =
  IntConst Int
  | BoolConst Bool
  | StringConst String
  | NilConst
  | UnitConst
```

```
data Monop = HdOp | TlOp | PrintOp | IntNegOp | FstOp | SndOp | NotOp
```

```
data Binop = IntPlusOp | IntMinusOp | IntTimesOp | IntDivOp
  | ConcatOp | ConsOp | CommaOp | EqOp | GreaterOp
```

4.2 Types

The goal of a type inferencing algorithm is to determine if an expression can be correctly assigned a type. So we also have a data structure that represents the types of our language.

```
data MonoTy = TyVar VarId | TyConst String [MonoTy] deriving Eq
type PolyTy = ([VarId], MonoTy)
```

A monomorphic type is represented as either a type variable, or as a type constructor, with a string indicating which type constructor is being used, and a list of types representing the arguments to the type constructor.

In this representation, ”basic” types are represented as type constructors that take no arguments. For example, the common file contains definitions for int, bool, string, and unit types respectively.

```
intTy = TyConst ”int” []
boolTy = TyConst ”bool” []
stringTy = TyConst ”string” []
unitTy = TyConst ”unit” []
```

It also can represent types that take arguments, for instance, list, pair, and function types. Our subset of ML does not utilize user-defined types, so these will be the only other types that we have. We've implemented functions that can be used to construct instances of these types for type inferencing as follows.

```
listTy tau = TyConst ”list” [tau]
pairTy t1 t2 = TyConst ”pair” [t1, t2]
funTy t1 t2 = TyConst ”fun” [t1, t2]
```

The definition for a polymorphic type is straightforward. It is simply a monomorphic type with a list of type variables that it is quantified over.

When we perform type inferencing, we will also have a notion of a type environment, which will map variable bindings to types. We use a Haskell map to represent these.

```
type TypeEnv = H.Map String PolyTy
```

Notice that a type environment maps variables bindings to *quantified* (polymorphic) types.

Lastly, we have a definition of type errors which ought to be thrown in the case of errors during the type inferencing or unification process. We will elaborate in more detail about when these errors need to be thrown in the problem definitions.

```
data TypeError =
  UnifError Exp
  | LookupError String
```

4.3 Fresh Variable Monad

In past MPs, we've heavily leaned on usage of the State monad + Error monad in order to carry information about the environment within our various interpreters. For this MP, we have many different things to keep track of. Type inferencing is performed relative to a type environment. We also need to continuously keep track of substitution constraints, which will be generated from our unification algorithm.

However, because we want to understand the exact process by which our type inferencing algorithm is gathering constraints, but we don't want to have to make you write a monad, we'll actually be excluding this state from our monad, and requiring you to explicitly carry it through the correct parts of the algorithm.

There are still a few reasons we need to use a monad. We still use a monad to propagate errors throughout the code. We also use it to generate fresh type variables, since there are many points in the type inferencing algorithm when we will need new types.

Our monad is declared in our common file as

```
data FVState a = ...
```

The functions you will be interested in are

```
freshTVar :: FVState VarId
freshTVar = ...

freshTau :: FVState MonoTy
freshTau = ...

throwError :: TypeError -> FVState a
throwError e = ...
```

The first function is used to generate fresh type variables. The second does the same thing, but also wraps it into a monomorphic type (for convenience, since the specific id is only likely to be of interest when implementing the fresh instance function). The third function is used to throw errors.

4.4 Substitutions

As we mentioned, we'll be manipulating both types in type environments, as well as substitutions, which map type variables to other monomorphic types (including possibly other type variables). We define a substitution using Haskell maps again.

```
type SubstEnv = H.Map VarId MonoTy
```

Various lifting functions are given which apply substitutions to types and type environments. To "apply" a substitution means to replaced all type variables with the type they are mapped to within the substitution (if there is no mapping for a type variable, it will not be changed).

```
liftMonoTy :: SubstEnv -> MonoTy -> MonoTy
liftMonoTy sEnv tau = ...

liftPolyTy :: SubstEnv -> PolyTy -> PolyTy
liftPolyTy sEnv (qVars, tau) = ...

liftEnv :: SubstEnv -> TypeEnv -> TypeEnv
liftEnv sEnv tEnv = ...
```

We also have the following functions for manipulating substitutions themselves. (Which are not essential, since they are essentially just wrappers over Haskell functions).

```
substEmpty :: SubstEnv
substEmpty = H.empty

substInit :: VarId -> MonoTy -> SubstEnv
substInit i tau = H.singleton i tau

substCompose :: SubstEnv -> SubstEnv -> SubstEnv
substCompose sEnv1 sEnv2 = H.union sEnv1 (scRec (toList sEnv2))
```

4.5 Auxiliary Functions

Our type inferencing rules utilize certain auxiliary functions which you will need to use. Some of them have been implemented for you, however you will have to implemente a few of them yourself.

The following functions will provide the types for all the constants, unary, and binary operators that will be used by the program. Note that these are polymorphic types, and they will have to be instantiated in order to get monomorphic types from them.

```
constTySig :: Const -> FVState PolyTy
constTySig c = ...

monopTySig :: Monop -> FVState PolyTy
monopTySig m = ...

binopTySig :: Binop -> FVState PolyTy
binopTySig b = ...
```

This MP deals with both monomorphic and polymorphic types. When typing expressions, we want to end up with a monomorphic type, but when adding a type to the type environment, we want a polymorphic type. Then, we require certain functions to convert between monomorphic types and polymorphic types. You will have to implement conversion of a polymorphic type into a monomorphic type ("instantiation")

However conversion from a monomorphic type into a polymorphic type will be done for you by a process known as "generalization". The intuition is somewhat complicated. The implementation is simply that it quantifies a monomorphic type by every free type variable except for type variables that are still free in the type environment.

```
gen :: TypeEnv -> MonoTy -> PolyTy
gen env tau = ...
```

5 Handing In / Testing

You must modify app/Infer.hs, as that is the only file we will be grading. You are allowed to (and in fact, are encouraged to) write your own tests. You can do so in test/Tests.hs. We have also provided you with a test suite that you can run via stack test.

There is also a file, Main.hs which you can compile if you want to run an interpreter. The interpreter will allow you to type in expressions + top-level declarations and test the results of your type inferencing on the given input code (provided that it parses properly).

Like previous MPs, you are not allowed to import other modules. You are allowed to use functions given to you in Prelude. You are encouraged to write helper functions, and are allowed to use functions written in earlier parts of this MP in later parts of the MP.

6 Problems

Your goal, as we stated, is to implement the type inferencing algorithm.

6.1 Fresh Instance Function

(0 pts, but necessary for the rest of the MP) The first step is to define the instantiation function that we mentioned earlier, otherwise known as the fresh instance function.

Given a polymorphic type, it finds all the quantified type variables in the type, replaces them with fresh type variables, and then returns the result as a monomorphic type.

The idea is that a use of a polymorphic value will have to be consistent within a single use, however we want different uses of the value to be able to instantiate the type variables differently. Therefore, we need copies of the type variables each time we use them.

The fresh instance function is actually defined using the monad, because it requires us to be able to generate fresh type variables.

```
freshInst :: PolyTy -> FVState MonoTy
freshInst (qVars, tau) = undefined
```

For instance, a fresh instance of ∀ 'a. 'a -> 'b might be something like 'c -> 'b, where 'c would be taken from a call to freshTVar and/or freshTau.

We also define a helper function to make it easier to compose uses of the fresh instance function in the context of the do-notation.

```
freshInstFV :: FVState PolyTy -> FVState MonoTy
freshInstFV s = ...
```

6.2 Occurs Function

(0 pts, but necessary for the rest of the MP) Next you will have to implement the unification algorithm. One minor auxiliary function you will need is the "occurs" check, which simply checks if a given type variable exists within a given monomorphic type.

```
occurs :: VarId -> MonoTy -> Bool
occurs i tau = undefined
```

For instance, the type variable 'a would be said to occur within 'a -> 'b, but not within int -> int or 'b -> 'c -> 'd.

6.3 Unification Algorithm

(20 pts) Now we move onto the unification algorithm. Unification is itself only an auxiliary function for type inferencing, but the algorithm is somewhat novel. In some ways it functions as the primary engine of the type inferencing algorithm.

The unification algorithm acts over a list of equations and returns a substitution. (This is another function which requires the use of the monad because we have to be able to throw errors.)

```
unify :: Exp -> [(MonoTy, MonoTy)] -> FVState SubstEnv
unify e eqList = undefined
```

An "equation" in this context is given as a pair of monomorphic types. The idea is that these are types which we require to be equal to each other. The unification algorithm will come up with a substitution that satisfies these equalities.

You'll notice that the unification algorithm takes in a list of pairs of types, but it also takes in an expression. This expression is returned with unification errors for debugging purposes. (You need not worry about putting in the "correct" expression, they're just to be useful when using the interpreter.)

Given an equation set C, here is a basic outline of the unification algorithm.

1. If C is empty, return the identity substitution.
2. If C is not empty, pick an equation $(s, t) \in C$. Let C' be $C \setminus \{(s, t)\}$.
   (a) Delete rule: If s and t are are equal, discard the pair, and unify C'.
   (b) Orient rule: If t is a variable, and s is not, then discard (s, t), and unify $\{(t, s)\} \cup C'$.
   (c) Decompose rule: If s is a PairTy, FnTy, or ListTy, $s = \text{Con } s_1 s_2$ and $t = \text{Con } t_1 t_2$ (where Con is the corresponding constructor), then discard (s, t), and unify $C' \cup \{(s_1, t_1), (s_2, t_2)\}$
   (d) Eliminate rule: If s is a variable, and s does not occur in t, substitute s with t in C' to get C''. Let ϕ be the substitution resulting from unifying C''. Return ϕ updated with $s \mapsto \phi(t)$.
   (e) If none of the above cases apply, it is a unification error (your unify function should return Nothing in this case).

The unification algorithm will fail in two cases. The first is if it fails the occurs check. The second is if during decomposition something fails. In both cases it suffices to simply throw a UnifError.

6.4 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:

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

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 from σ_1 is applied to our assumptions), and the substitution generated from the 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 'a.
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 '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.

Problem 1. (5 pts) Implement the rule for constants:

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

Note that this makes use of both the freshInst and constTySig auxiliary functions. (You may want to use freshInstFV in here and in other locations, since constTySig and the other type signature auxiliary functions require use of the monad.)

A sample execution would be

```
Enter an expression:
46
Parsed as:
  46
Inferred type:
  int
```

Problem 2. (5 pts) Implement the rule for variables:

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

Note that $\Gamma(x)$ represents looking up the value of x in Γ. (Ie performing lookup in the Hashmap.) This is the only other part of our code that can throw an error (a LookupError).

Also note that you will not be able to test this in the interpreter unless you at least partially implement let expressions as well.

A sample execution is

```
let f = 0
Parsed as:
  let f = 0
Inferred type:
  int
Enter an expression:
f
Parsed as:
  f
Inferred type:
  int
```

Problem 3. (5 pts) Implement the rule for LetInExp.

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

This rule is somewhat complicated to understand, but it is relatively important for being able to use the interpreter, since it gets used for top-level declarations. However it's relatively simple to implement since we've already given you the gen function.

Here is the sample execution:

```
Enter an expression:
let x = true in x
Parsed as:
  (let x = true in x)
Inferred type:
  bool
```

Problem 4. (10 pts) Implement the rule for built-in binary and unary operators:

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

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

where $\otimes$ is a built-in binary or unary operator.

A sample execution would be:

```
Enter an expression:
3 + 4
Parsed as:
  (3 + 4)
Inferred type:
  int
Enter an expression:
~ 7
Parsed as:
  (~ 7)
Inferred type:
  int
```

Problem 5. (10 pts) Implement the rule for if_then_else:

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

Here is a sample execution:

```
Enter an expression:
if true then 62 else 252
Parsed as:
  (if true then 62 else 252)
Inferred type:
  int
```

Problem 6. (10pts) Implement the function rule:

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

Here is a sample execution:

```
Enter an expression:
fun x -> x + 1
Parsed as:
  (fun x -> (x + 1))
Inferred type:
  (int -> int)
```

Problem 7. (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:

```
Enter an expression:
let f = fun x -> x + x in f 3
Parsed as:
  (let f = (fun x -> (x + x)) in (f 3))
Inferred type:
  int
```

Problem 8. (5 pts) Implement the rule for LetRecInExp.

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

Here is the sample execution:

```
Enter an expression:
let rec f x = f (x - 1) in f
Parsed as:
  (let rec f x = (f (x - 1)) in f)
Inferred type:
  (int -> 'a)
```