MP 6 – A Unification-based Type Inferencer

CS 421

Revision 1.1

1 Change Log

1.0 Initial Release.

1.1 Clean up given code and use constraint generation algorithm

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 names, and also to represent bindings. PicoML does not have 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
data PolyTy = Forall [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 "->" [t1, t2]
```

4.3 Polymorphic Type Explanation

Polymorphic types are derived from monomorphic types. Rather than representing a specific monomorphic type, they actually represent an entire set of monomorphic types, using an underlying monomorphic type as a "template". An example of a polymorphic type would be

$$\forall\alpha.(\alpha \to \texttt{int} \to \alpha\ \texttt{list})$$

This polymorphic type denotes the set of all monomorphic types that take some type α, and an integer, and return a list of values also of type α. The type $\texttt{string} \to \texttt{int} \to \texttt{string list}$ would be included in this set, taking α as $\texttt{string}$. The type $(\texttt{string} \to \texttt{bool}) \to \texttt{int} \to (\texttt{string} \to \texttt{bool})\ \texttt{list}$ would also be within this set, where α is the function type $\texttt{string} \to \texttt{bool}$. We would even have $\beta \to \texttt{int} \to \beta\ \texttt{list}$ in this set, where the α chosen is another type variable β.

On the other hand $\texttt{bool} \to \texttt{int} \to \texttt{int list}$ would not be in this set, since α is assigned both $\texttt{bool}$ and $\texttt{int}$ in different places, which is contradictory.

If a monomorphic type is in the set of types represented by a polymorphic type, we say that it is an *instance* of the polymorphic type. Generally speaking a monomorphic type τ is an "instance" of a polymorphic type σ if the type variables in the template type can be replaced in some way to construct τ. You may see some authors write $\tau \sqsubseteq \sigma$. Note that not all type variables in a polymorphic type template can be replaced. For instance, take the following polymorphic type

$$\forall\beta.(\alpha \to \beta \to (\alpha, \beta))$$

Here, $\alpha \to \texttt{int} \to (\alpha, \texttt{int})$ would be considered an instance of this polymorphic type where β is replaced by $\texttt{int}$, while $\texttt{string} \to \beta \to (\texttt{string}, \beta)$ would not be, even though we can get it by replacing α with $\texttt{string}$. This is because the $\forall$ quantifier given by the polymorphic type only includes β and not α.

In general, the type variables given by the $\forall$ are the only variables which can be replaced to check if a monomorphic type is an instance of a polymorphic type. We call such variables the *quantified* type variables. Note that by this definition, a monomorphic type is just a polymorphic type with no quantified type variables.

Going back to Haskell, the way polymorphic types (`PolyTy`) are defined in our language is as a datatype consisting of a list of the quantified variables (a `VarId` list) and a monomorphic type.

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 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
  = InfiniteType VarId MonoTy
  | Can'tMatch MonoTy MonoTy
  | LookupError String
```

4.4 Unification Constraints

As part of our algorithm, we will generate a list of unification constraints. A constraint of the form $\tau_1 \sim \tau_2$ expresses that the monomorphic types τ_1 and τ_2 must be the same type. In Haskell;

`data Constraint = MonoTy :~: MonoTy`

Like the list constructor `:`, the constraint constructor `:~:` is infix. You can pattern match on a constraint by writing `t1 :~: t2`. You can even use `:~:` as a function directly. The function `zipWith (:~:) :: [MonoTy] -> [MonoTy] -> [Constraint]` might be helpful for this MP.

4.5 Substitutions

Once we have all of the unification constraints, our goal will be to unify those constraints to find a substitution that satisfies them. A substitution maps type variables to other monomorphic types (including possibly other type variables). We define a substitution using Haskell maps again.

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

To "apply" a substitution means to replace all type variables with the type they are mapped to by the substitution (if there is no mapping for a type variable, it will not be changed). We write $\sigma(\tau)$ for the application of substitution σ to type τ. Twice, we will have a set of constraints ϕ which we want to solve and immediately apply the resulting substitution to a type. For this, we write *unify*$(\phi)(\tau)$.

We define a custom typeclass called `Type` to express that a (Haskell) type contains (PicoML) types.

```
class Type t where
  apply :: Substitution -> t -> t

instance Type MonoTy where ...
instance Type PolyTy where ...
instance Type TypeEnv where ...
instance Type Constraint where ...
```

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

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

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

4.6 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 need a supply of fresh type variable names. We need to be able throw errors. We also need to continuously keep track of unification constraints, which will be generated from our inference algorithm.

However, because we want you to understand the exact environment that each inference is performed in, we will exclude that from our monad and require you to explicitly use the correct environment in each step of the algorithm.

There are still a few reasons we need to use a monad. We will use a monad to propagate errors through the code. We'll use it to generate fresh type variables, since there are many points in the type inferencing algorithm when we will need new types. Finally, we'll use it to keep track of all of the unification constraints that we generate, so that we don't need to manually bind and combine all of the constraint sets at every step.

Our monad is declared in our common file as

```
type Infer a = ...
```

The functions you will be interested in are

```
freshTau :: Infer MonoTy
freshTau = ...

throwError :: TypeError -> Infer a
throwError e = ...

constrain :: MonoTy -> MonoTy -> Infer ()
constrain tau1 tau2 = ...
```

The first function is used to generate fresh type variables. The second function is used to throw errors. The third function adds the constraint $\tau_1 \sim \tau_2$ to the generated constraint set.

Here is a dummy example of what the syntax would look like in practice:

```
foo x = do tau1 <- freshTau
           constrain tau1 intTy
           if bar tau1 then throwError ...
           else return tau1
```

4.7 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 implement one 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 -> PolyTy
constTySig c = ...

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

binopTySig :: Binop -> 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 to quantify all of the type variables in the monomorphic type that aren't also shared with another type in the type environment.

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

Sometimes you will need to put a monomorphic type into the type environment without generalizing it. As discussed above, a monotype is a polytype with no quantified variables.

```
polyTyOfMonoTy :: MonoTy -> PolyTy
polyTyOfMonoTy tau = Forall [] tau
```

5 Handing In / Testing

You must modify `src/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 main 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). You can use `stack build && stack exec infer` to run the interpreter.

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.

In 4.3, we described what it means for a monotype to be an instance of a polytype. To make "fresh instance" of a polytype, you replace all of the quantified variables with fresh type variables. The idea is that the monomorphic type we return will be an instance of the polymorphic type.

The greater idea here is that whenever a variable is referenced, there is some monomorphic type that it must have at that particular use. We want to ensure that this monomorphic type is an instance of the actual polymorphic type that the variable was given. We accomplish this by constraining the inferred type to equal a fresh instance of the polymorphic type template. The type must be given "fresh" variables in case some of the quantified type variables are already in use somewhere else in the program. This is how the fresh instance function will eventually be used.

Practically speaking, the fresh instance function can be implemented several different ways. We recommend that you implement it in two steps. First, take the list of quantified type variables and pair each of them together with fresh type variables. Second, create a substitution using this pairing and apply it to the `MonoTy` portion of the `PolyTy`.

For example, to generate a fresh instance of `∀ 'a 'c. 'a -> 'b -> 'c`, we would start by generating fresh type variables for `'a` and `'c`. These can be generated from a call to `freshTau`. Let's say we get type variables `'d` and `'e` respectively. We would then replace them in the original type `'a -> 'b -> 'c` to get type `'d -> 'b -> 'e`.

While the fresh instance function is not graded, a few test cases are included in `Tests.hs` in `freshGroup` which you can use as reference if you want more examples.

In order to generate fresh type variables, you will need to use the `Infer` monad. As a result, the fresh instance function actually has a monadic return type.

```
freshInst :: PolyTy -> Infer MonoTy
freshInst (Forall qVars tau) = undefined
```

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`.

You can write this function by hand, but for your convenience we have also provided a function

```
freeVars :: MonoTy -> [VarId]
freeVars tau = ...
```

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 constraints and returns a substitution that solves them. This is another function which requires the use of the monad because we have to be able to throw errors.

```
unify :: [Constraint] -> Infer Substitution
unify eqList = undefined
```

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

1. If ϕ is empty, return the identity substitution.
2. If ϕ is not empty, pick an equation $(s \sim t) \in \phi$. Let ϕ' be $\phi \setminus \{s \sim t\}$.
   (a) **Delete rule:** If s and t are are equal, discard the pair, and unify ϕ'.
   (b) **Orient rule:** If t is a variable, and s is not, then discard (s, t), and unify $\{t \sim s\} \cup \phi'$.
   (c) **Decompose rule:** If $s = \texttt{TyConst}(c_1, [s_1, \cdots s_n])$ and $t = \texttt{TyConst}(c_2, [t_1, \cdots t_n])$, if $c_1 = c_2$ then the type constructors match, so unify $\{s_1 \sim t_1, \cdots, s_n \sim t_n\} \cup \phi'$. Otherwise, your function should throw the error `Can'tMatch s t`.
   (d) **Eliminate rule:** If s is a variable, and s does not occur in t, substitute all occurences of s in ϕ' with t to get ϕ''. Let σ be the substitution resulting from unifying ϕ''. Return σ updated with $s \mapsto \sigma(t)$.
   If s does occur in t, throw the error `InfiniteType s t`.

The unification algorithm can only fail in two cases. The first is if type constructors don't match in the Decompose rule. The second is if the occurs check fails.

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 set of constraints 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 constraint set. Overall, the type will be `infer :: TypeEnv -> Exp -> Infer MonoTy`. Here's an example:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \phi_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid \phi_2}{\Gamma \vdash e_1 + e_2 : \texttt{int} \mid \{\tau_1 \sim \texttt{int}, \tau_2 \sim \texttt{int}\} \cup \phi_1 \cup \phi_2}$$

The "|" is just some notation to separate the constraint set from the expression. You can pronounce it as "subject to". This rule says that the constraints sufficient to guarantee that the result of adding two expressions e_1 and e_2 will have type `int` is the union of the constraint set ϕ_1 guaranteeing that e_1 has type τ_1, the constraint set ϕ_2 guaranteeing that e_2 has type τ_2, and the constraint set $\{\tau_1 \sim \texttt{int}, \tau_2 \sim \texttt{int}\}$.

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 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 `'a`. Add that to Γ and try to infer the type of the body ...
   (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 `'a`.
      ii. Examine `2`. This is an `int`.
   (b) We return that the type of the body is `int`, along with the constraint set $\{\texttt{'a} \sim \texttt{int}, \texttt{int} \sim \texttt{int}\}$. Yes, the last constraint is trivial, but the rule says we have to do it. Note that in code, we don't need to manually combine the new constraints with ϕ_1 and ϕ_2, because the monad is handling that for us. The new constraints would be generated by two calls to `constrain`.
3. Now we're ready to come back to the type of the whole expression. The variable `x` has type `'a`, and the output has type `int`, so the whole expression has type `'a -> int`, subject to the constraint set $\{\texttt{'a} \sim \texttt{int}, \texttt{int} \sim \texttt{int}\}$.
4. Unify the constraint set to find the substitution that solves it; the substitution is $\{\texttt{'a} \mapsto \texttt{int}\}$.
5. The substitution tells us that we need to rewrite `'a` to `int` everywhere, so we get a final type of `int -> int`.

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

$$\frac{}{\Gamma \vdash c : \mathit{freshInst}(\mathit{constTySig}(c)) \mid \emptyset}\ \textsc{Const}$$

Note that this makes use of both the `freshInst` and `constTySig` auxiliary functions. Remember that `freshInst` returns its result in 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:

$$\frac{}{\Gamma \vdash x : \mathit{freshInst}(\Gamma(x)) \mid \emptyset}\ \textsc{Var}$$

where x is a program variable.

Note that $\Gamma(x)$ represents looking up the value of x in Γ (i.e. performing lookup in the map). This is the only part of the code for `infer` 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`.

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \phi_1 \qquad [x : \mathit{GEN}(\Gamma, \mathit{unify}(\phi_1)(\tau_1))] + \Gamma \vdash e : \tau \mid \phi_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1 \texttt{in}\ e : \tau \mid \phi_1 \cup \phi_2}\ \textsc{Let}$$

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. Compare it to the following incorrect rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \phi_1 \qquad [x : \mathit{GEN}(\Gamma, \tau_1)] + \Gamma \vdash e : \tau \mid \phi_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1 \texttt{in}\ e : \tau \mid \phi_1 \cup \phi_2}\ \textsc{Let (bad!)}$$

What goes wrong? Imagine we are trying to infer the type of the following:

```
fun x -> let y = x + 1 in y
```

After inferring the type of `y`, it will be a type variable `'a` and we will have a constraint `'a` $\sim$ `int`. We would then apply *GEN* immediately, adding $[y : \forall' a.'a]$ to Γ while typechecking e. Then, while typechecking e, we call *freshInst* on the type of `y` and we return the result, `'b`. However we don't have a constraint that `'b` $\sim$ `int`, so after unifying constraints, the type is still `'b`. To get around this problem, we have to unify the constraints generated by inferring the type of e_1 and apply the resulting substitution to τ_1 before generalizing it.

To get the generated constraints, we have provided the function `listen :: Infer a -> Infer (a, [Constraint])`. You can look at the given definition of `inferInit` for an example of how to use it.

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 rules for built-in binary and unary operators:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \phi_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid \phi_2}{\Gamma \vdash e_1 \otimes e_2 : \tau \mid \{\tau_1 \to \tau_2 \to \tau \sim \mathit{freshInst}(\mathit{binopTySig}(\otimes))\} \cup \phi_1 \cup \phi_2} \textsc{Binop}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \phi}{\Gamma \vdash \otimes e_1 : \tau \mid \{\tau_1 \to \tau \sim \mathit{freshInst}(\mathit{monopTySig}(\otimes))\} \cup \phi} \textsc{Monop}$$

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`:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid \phi_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid \phi_2 \qquad \Gamma \vdash e_3 : \tau_3 \mid \phi_3}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : \tau_2 \mid \{\tau_1 \sim \texttt{bool}, \tau_2 \sim \tau_3\} \cup \phi_1 \cup \phi_2 \cup \phi_3} \textsc{If}$$

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:

$$\frac{[x : \tau_1] + \Gamma \vdash e : \tau_2 \mid \phi}{\Gamma \vdash \texttt{fun}\, x \;\texttt{->}\; e : \tau_1 \to \tau_2 \mid \phi} \textsc{Fun}$$

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:

$$\frac{\Gamma \vdash f : \tau_1 \mid \phi_1 \qquad \Gamma \vdash e : \tau_2 \mid \phi_2}{\Gamma \vdash f\, e : \tau \mid \{\tau_1 \sim \tau_2 \to \tau\} \cup \phi_1 \cup \phi_2} \textsc{App}$$

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`.

$$\frac{[x:\tau_1]+[f:\tau_1\to\tau_2]+\Gamma\vdash e_1:\tau_3 \mid \phi_1 \qquad [f:\mathit{GEN}(\Gamma,\mathit{unify}(\{\tau_2\sim\tau_3\}\cup\phi_1)(\tau_1\to\tau_2))]+\Gamma\vdash e:\tau \mid \phi_2}{\Gamma\vdash (\texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e):\tau \mid \phi_1\cup\phi_2}\ \textsc{LetRec}$$

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)
```