Project 5: Simply Typed Lambda Calculus with Type Reconstruction

Deadline: December 3, 23:59 **Submission:** Moodle **Code:** 5-inference.zip

1 Assignment

In this assignment you will write a Hindley-Milner type inferencer for simply typed lambda calculus.

You will be given a working parser and this time we're not interested in evaluation, but only type checking. Of course, you can reuse an evaluator from earlier projects, for 'fun', but you will be evaluated (forgive the pun) only for the type checker. This type checker, however, is more interesting than the previous one: instead of requiring the user to give types to variables in lambda abstractions (and `let`), we want it to be smart enough to *infer* them. In a way, the source language can be regarded as the untyped lambda calculus, in which the user might give types in lambda abstractions.

2 Source Language

We are interested in a simple version of simply typed lambda calculus, with booleans, numbers and `let`. Figure 1 shows the syntax for our language. memory:

The only difference from exercise 3 is that the two type annotations are optional. We also keep `let` in the abstract syntax tree, instead of treating it as a derived form. This will prove useful when we implement let-polymorphism. The evaluation rules are the same as before.

3 Constraint-based Typing

The basic idea about type inference is to treat each type judgment as a constraint on the types it involves, rather than a check on them. This way, typing will collect a set of such constraints, and never fail at this point. Afterwards, it tries to solve the constraint list, and if it succeeds the program is well-typed. Before we delve into the constraint typing rules, we need to detail a bit what constraints operate upon, and what is a *solution* to the constraint list.

3.1 Type Variables

When we talk about types, but we don't want to make them concrete, to name them, we say "a type T..." and then we use this new name to refer to all types. We *abstract* over types. Type variables are just that: types that could be substituted by any concrete type: $Nat, Bool \to Bool$, etc. It is important to note that type

$t ::=$ "$true$"	true value
| "$false$"	false value
| "if" t_1 "$then$" t_2 "$else$" t_3	if
| $numericLit$	integer
| "$pred$" t	predecessor
| "$succ$" t	successor
| "$iszero$" t	iszero
| x	variable
| "\" x [" : " T] "." t	abstraction
| $t\ t$	application (left associative)
| "let" x [" : " T] " = " t_1 "in" t_2	let
| "(" t ")"	
$v ::=$	application (values)
| "$true$"	
| "$false$"	
| nv	numeric value
| "\" x " : " T "." t	abstraction value
$nv ::=$	numeric values
| "0"	
| "$succ$" nv	

Fig. 1. Source Language

variables can appear in other types, for instance $a \to a$, where a is a type variable, is also a type.

We will add another alternative in our types, although this will not be available to users writing programs (but it will be used during type reconstruction):

$T ::= "Bool"$	boolean type
$\mid "Nat"$	numeric type
$\mid T "->" T$	function types (right associative)
$\mid id$	type variables
$\mid "(" T ")"$	

3.2 Substitutions

A substitution σ is a mapping from type variables to types. For example, $[X \to Y, Y \to Nat]$. There are two operations that are defined on substitution:

- *applying* a substitution to a type T giving σT: it means replacing all occurrences of type variables in T for which σ is defined with their mapping.
- *extending* a substitution means adding a new mapping from a type variable to a type.

Application can be straight-forwardly extended for typing contexts or constraints, by applying the substitution to each type in the context or constraint.

As you have seen in the lecture, solving a constraint system gives back a substitution. It means each constraint in the system can be made `true` (or satisfied) if we apply the given substitution to the two types.

The easiest way to model substitutions in Scala is to make the `Substitution` class inherit from a function type (`Type ⇒ Type`, the type of functions from `Type` to `Type`). This gives us convenient syntax for application, we can simply write a `s(t)` for applying substitution `s` to type `t`. You will also need to add methods for extending a substitution. It might be nice to have methods for composing two substitutions, giving back a new substitution which acts as if the two substitutions have been applied one after the other.

Constraints can be modeled by a simple `Pair` of types.

3.3 Unification

The unification algorithm works by inspecting at each step one constraint, and extending the substitution with a new mapping, if necessary. Suppose the current constraint is T = S:

- if `T == S` (the two types are the same), proceed to next constraint, (this one is trivially satisfied)
- if one of them is a type variable `X`, and `X` does not appear in the other one, extend the resulting substitution with `[X -> T]` and proceed to next constraints. The check is necessary to avoid recursive constraints, which would lead to non termination. **Note:** The fact that `X` is bound to some type `T`

has to be taken into account for the constraints not yet examined. One way to do this is to substitute all occurrences of `X` with type `T` in the remaining constraints.
- if both are function types, unify the argument and result types, respectively.
- else fail. In this case we can't find a substitution that satisfies the current constraint. It will translate to a type error.

3.4 Constraint-based Typing

In constraint-based typing, the result of the typing function is not just its type, but also a list of constraints on types. The type returned by the type checker at this point will probably contain many type variables, and it's actual type is found by solving the constraints and applying the resulting substitution to that type. If the unifier fails, it means the term is not type-correct.

For example:

```
(\b.if b then false else true)
```

has a type `X -> Bool` and a constraint list `[X = Bool]`. Only after solving this (trivial) constraint, and applying the resulting substitution, we get the correct type of this term: `Bool -> Bool`.

Here are the typing rules, augmented with constraints. If you compare these rules with the ones given in the book (page 322), you will notice that we have ignored "freshness" conditions on type variables: it is necessary that type variables, whenever created, are different from all other type variables in the system. This is easy to achieve when programming the type checker, and including them here would only obfuscate the typing rules. However, a formally correct type system needs to include them.

We read a typing judgment (`iszero t`) like this: "suppose typing `t` returns a type `T` and a set of constraints `C`; the type of `iszero t` is `Bool` and the constraints are `C` augmented with the constraint that `T` has to be `Nat`".

Most type judgments are straight forward, the only two which require some thinking are the ones for abstraction and application. Let's start with application: As before, we type the two subterms, and we get back two sets of constraints and two types. Next, we need to make sure the type t_1 is a function type, so we add a constraint. But what should the result of this function be (the argument type is clear that it is T_2)? We don't know, so we invent a new type variable, and introduce it in the constraint system as `T1 = T2 -> X`. Hopefully, by the end of type checking, other constraints will make this new type variable to be some concrete type. The same argument goes for abstractions: what should be the type of `x`?

Note: The parser we provided makes optional the type in lambda abstractions. If there is one, we take it into account, otherwise we invent a fresh type variable and introduce it in the environment.

$$\Gamma \vdash true : Bool \mid \{\}$$

$$\Gamma \vdash false : Bool \mid \{\}$$

$$\Gamma \vdash 0 : Nat \mid \{\}$$

$$\frac{\Gamma \vdash t : T \mid C \quad C' = C \cup \{T = Nat\}}{\Gamma \vdash pred\ t : Nat \mid C'}$$

$$\frac{\Gamma \vdash t : T \mid C \quad C' = C \cup \{T = Nat\}}{\Gamma \vdash succ\ t : Nat \mid C'}$$

$$\frac{\Gamma \vdash t : T \mid C \quad C' = C \cup \{T = Nat\}}{\Gamma \vdash iszero\ t : Bool \mid C'}$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \quad \Gamma \vdash t_2 : T_2 \mid C_2 \quad \Gamma \vdash t_3 : T_3 \mid C_3 \quad C = C_1 \cup C_2 \cup C_3 \cup \{T_1 = Bool, T_2 = T_3\}}{\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3 : T_2 \mid C}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid \{\}}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2 \mid C}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2 \mid C}$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \quad \Gamma \vdash t_2 : T_2 \mid C_2 \quad X \text{ is fresh}, C = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}}{\Gamma \vdash t_1\ t_2 : X \mid C}$$

Fig. 2. Type rules

3.5 Let-polymorphism

You may have noticed the above typing rules miss one important case: `let`. It deserves special treatment. We would like to implement let-polymorphism, that is, to allow a piece of code defined by `let` be run with different types. Consider a `double` function, which applies its first argument twice to its second argument:

```
let double = \f.\x.f(f(x)) in
  if (double (\x:Bool. if x then false else true) false)
    then double (\x:Nat.succ x) 0
    else 0
```

We'd like double to be able to work for both `Nat` and `Bool` functions. Notice that if we treat it simply by inventing a type variable which participates in constraints, it would not work as expected: constraints in its first use would deem that its a `(Bool -> Bool) -> Bool` while constraints in the second use would require `(Nat -> Nat) -> Nat`.

3.6 Type Schemes

The solution is to generalize the type of double by making it a *type scheme*. A type scheme is a "recipe" for creating types, and it works by abstracting over its type variables. Each time double is used, we instantiate the type scheme to yield a new type, which can participate in type constraints.

A *type scheme* is a type and a list of type variables (used in that type) which can be instantiated. Instantiation means inventing fresh type variables for each

of the arguments of a type scheme, and substituting them with the fresh ones. For example:

```
TypeScheme(List("a"), a -> a) instantiates to a1 -> a1
```

3.7 Type-checking *let*

Here's a sketch of the type checking procedure for `let x = v in t`:

1. we type the right hand side v obtaining a type `S` and a set of constraints `C`.
2. We use unification on `C` and apply the result to `S` to find its first approximation as type. At this point, the substitution we found should be applied to the current environment too, since we have committed to a set of bindings between type variables and types! Let's call this new type `T`
3. We generalize some type variables inside T and obtain a type scheme. **Important:** We need to be careful about what type variables we generalize. We should not generalize any type variables appearing in the environment, because they appear in constraints that need to be satisfied. For example:

   ```
   (\f.\x. let g = f in g(0)) (\x.if x then false else true) true
   ```

   While typing `let`, the environment will contain bindings such as `[f -> $X_1$, g -> $X_2$]` and some constraints on them, like { `$X_1$ = Bool -> Bool` }. Notice inside `let` we use `g` as a function on `Nat` and that should be a constraint on `f` too, which is an argument of the function. Applying this function to `Bool` should fail. If we wrongly generalize, we would get a `TypeScheme($X_1$, $X_1$)` for `g` which would be instantiated at use to X_3, which would be constrained to be a function on `Nat`. Notice how this constraint will not involve the X_1 anymore, and the type checker would miss the type error! The bottom line is, generalizing type `T` to a type scheme should only abstract type variables that don't appear in the current environment.
4. We extend the environment with a binding from `x` to its type scheme. Environments will not contain bindings to types, but to type schemes. Each time we lookup a variable in the environment, we need to instantiate its type scheme. In trivial cases, type schemes have an empty argument list, and their instantiation is always the type they contain. We type check `t` with the new environment.
5. Each time `x` appears in `t`, its type scheme will be instantiated and used as a type for `x`. In the previous point we proposed to unify treatment of variables found in the environment, so all lookups actually instantiate type schemes, and this point is implicitly followed.

4 Implementation Hints

This is probably the most challenging project so far, so we have given you a bit more framework (notably, the parser and the abstract syntax trees). Here are

some more implementation hints, which might be useful to you (but it does not mean they have to be followed, or that they are really the best way to solve this exercise):

- The hierarchy of types is different from the hierarchy of syntax trees that appear in the source code. By convention, we use "Type" as a suffix for trees appearing in the syntax, and "Type" as a prefix (as in `TypeVar`) in the type hierarchy. The two have no common superclass.
- `TypeScheme` is not a type, so it should not subclass `Type`. It can be turned into a `Type` by instantiation.
- Environments can be list of pairs of a name (variable name, not to be mistaken for a type variable) and a type scheme.
- Constraints are pairs of types. They could be made into full fledged classes, and carry a position with them (the position of the tree which caused this constraint). This would make type errors more user-friendly.
- You will need to organize functionality related to types (like generalization, collecting type variables, or fresh type variable construction). You can use an `object Type` which acts in many ways as the static part of a class, in Java terms. You can later `import` this functionality where you need it (as with packages).
- You might define objects for the empty substitution and the empty list of constraints, since they might appear in more than one place.
- Type scheme instantiation can be implemented using substitution.