Assignment 2: A Type Checker

Due date: Wednesday March 28 before 10pm

Updated: rough marking scheme posted here

In dynamically-typed languages like Python and Racket, basic type-checking is pretty straightforward to implement. This is because values are stored in memory together with a tag representing their type (intuitively, as an extra "type" instance attribute) that can be accessed at runtime. For example, the type of the expression `(if x 3 "hello")` is determined first by evaluating `x`, then choosing and evaluating one of the branch expressions, and then checking its type. The type of the `if` expression as a whole can only be calculated at runtime, since it depends on the *value* of `x`.

In statically-typed languages, types of `if` expressions, and every other kind of expression, are determined just by analyzing the source code—without evaluating any expression at all! This is more work, as the type checker must be able to calculate the type of an expression *without* knowing its value. To support this, statically-typed languages usually enforce additional restrictions as part of their type systems; for example, Haskell requires that both branches of an `if` have the same type. On this assignment, you'll implement a static type checker that does a (small) part of what Haskell's type checker can do, including supporting basic generic polymorphism and even some *type inference*, which is becoming more and more common in mainstream languages.

Type system description

In Assignment 1, we specified a grammar that you used to break down a Racket program and interpret it. Because we are operating in Haskell, we don't have the convenient `quote` to turn a program into a datum (nested list), so won't describe the language's surface-level syntax, but instead the different types of expressions that are represented using the `Expr` type in our cde. Moreover, because we're doing type-checking, we need to specify the various *rules* governing type constraints and the handling of polymorphism.

Program

A program consists of either a single expression or a sequence of name-value bindings followed by a single expression. The *type* of a program is the type of its single top-level expression. Each binding introduces an identifier that can be used in subsequent program expressions; the *type* of an identifier is equal to the type of its corresponding expression.

Literals

There are two kinds of literals, integers and booleans. They have *type* `Int_` and `Bool_`, respectively.

If expressions

An if expression must have three parts (condition, "then" expression, "else" expression). The *type* of an if expression is equal to the type of its "then" expression, as long as the following two conditions hold (we label them with error message variables given in the starter code):

- The condition of an `If` must have type `Bool_`. (`errorIfCondition`)
- The two branches must have the same type. (`errorIfBranches`)

Function call

A function call consists of an expression (the "function" expression) followed by zero or more expressions (the "argument" expressions). The *type* of the function call is equal to the *return type* of the function expression, as long as the following conditions hold:

- The type of the first expression of the call is a `Function` type. (`errorCallNotAFunction`)
- The number of arguments in the call is equal to the number of parameters in the function type. (`errorCallWrongArgNumber`)
- The type of each argument can be "unified" with the corresponding parameter type. (`errorCallWrongArgType`)

Functions can be generically-polymorphic, indicated by a type that contains one or more type variables. See Part 1 below for details about type unification and handling polymorphic function calls. (Note: on Exercise 9, you just need to check whether corresponding types are *equal*.)

Lambda expression

A lambda expression consists of a list of parameter names and a single "body" expression. The *type* of the lambda expression is a `Function` whose parameter types are *inferred* through static analysis of the lambda's body (see Part 2 below), and whose return type is the type of the body expression.

The type checker reports ~~a `errorTypeUnification`~~ (**updated**) *any* error using `Left` if type-checking the body produces a type, but generates constraints on the parameter types that cannot be satisfied (e.g., a parameter must be both an `Int_` and a `Bool_` at the same time). See the last test in the provided sample tests in the starter code.

To make this assignment a bit easier, we have the following restrictions on allowed lambda expressions (you may assume these always hold in your implementation):

- When used in top-level bindings, the lambda is non-recursive.
- The inferred type of every parameter and return value is `Int_`, `Bool_`, or a type variable. In other words, *you do not need to handle the case of inferring a function type explicitly* (this is harder to do, why?).

  In terms of language semantics, we could say that the only *user-defined* functions that are higher-order (take and/or return functions) are maximally polymorphic in the corresponding parameter/return type. For example, your type checker should be able to infer the type of `id x = x` to be `id :: a -> a`, but does not need to handle `apply f x = f x`, which has inferred type `apply :: (a -> b) -> a -> b`, since the latter has a parameter type whose "outer shape" is a `Function`.

Restrictions regarding name collisions

[*Note: this subsection is fairly technical and can be safely skipped on first read.*]

Using strings to label type variables exposes us to the problem of name collisions; for example, in the expression `(if (identity #t) (identity 4) (identity 5))`, the same type variable `TypeVar "t2"` takes on a `Bool_` type in the first clause, and `Int_` type in the other two clauses. While more sophisticated type-checkers would handle this by a combination of scoping and renaming, this is beyond the scope of this assignment.

Instead, you may assume the following:

- Parameters names are *globally unique* across all functions. So if a lambda expression has a parameter called `myParam`, *no other lambda expression* may have this parameter name. (This enables deterministic generation of unique type variable names—see `genTVarName` in the starter code.)
- Every expression contains at most one call to each polymorphic function. (This helps avoid name collision for repeated uses of the same function—like `identity`—in a single expression.)

  Note, however, that this restriction is a *local* one. We consider the expressions in each top-level binding, and the single top-level expression, to all be distinct expressions, and so each can make a call to the same polymorphic function.

Type checker behaviour

Finally, we specify exactly what your program needs to do for this assignment. Your program is a *type checker* that takes as input a program and calculates the type of the program, or returns an error message if the program contains an error.

The *first* time the type-checker detects an error, it should stop type-checking the rest of the program and simply return that error message. This is described on the Exercise 9 handout as "propagating the error message upwards." For programs with multiple errors, you may choose which error to report (this may be implementation-specific, determined by the order in which you check certain types).

Starter code

- `TypeChecker.hs`. This starter code is a modified version of the starter code for Exercise 9.
  - **Updated** March 16: fixed some typos in definitions of `runTypeCheck` and `TypeConstraints`. You may have fixed these already, but just in case please compare those two definitions with the updated starter code.
- (**new**) (English) test descriptions from Exercise 9
- (**new**) `TypeCheckerTest.hs`

As usual, we'll post some more information about testing and a sample marking scheme closer to the deadline. Please note that you're welcome to reuse your tests from Exercise 9, but we strongly recommend putting them in a separate file (you can just do `import TypeChecker`).

Part 0: Complete Exercise 9

Your first step on working on this assignment is to complete Exercise 9, which gets you to implement a basic type-checker that works for integer and boolean literals, `if` expressions, and non-polymorphic function calls.

Once you've completed that exercise, you can transfer your code directly to this assignment's starter code. You'll be able to re-use almost everything, though you'll likely need to make modifications to extend the capabilities of your type-checker for this assignment.

Note about group work: while Exercise 9 must still be submitted individually, we'll allow you to work closely with a partner so that if you choose to work in a group for this assignment, you can complete "Part 0" together as well.

Part 1: Calling polymorphic functions

Your first new task here is to support type-checking of function calls where the function being called is polymorphic. We've provided a few such functions in the starter code, like `identity` and `apply`.

Your first instinct might be to just treat a `TypeVar` as a wildcard, similar to `_` in pattern-matching. While this works for the first function, the problem is when the same type variable is re-used in multiple places in the function signature, e.g. `identity :: t2 -> t2`. When we match argument types to a function's parameter types, we need not just to check whether the types are equal, but in the case of type parameter, we need to save what type(s) the type variable should actually be bound to. For example, in the function call `id 3`, the type variable `t2` in `t2 -> t2` is constrained to `Int_`, and the type-checker should be able to determine that `identity 3` returns an `Int_`.

Type unification

The key idea to implement this is *type unification*, a generalization of type equality (i.e., comparing types using `(==)`, which you probably did on Exercise 9). Whereas type equality can be represented by the function type `Type -> Type -> Bool`, the key function `unify` will have type `Type -> Type -> Maybe TypeConstraints`. Let's break down that return type:

- `Maybe` is used to capture the idea that a unification may succeed or fail. We avoid using `Either` here because `unify` is a helper; it's up to the calling code to take the result convert it to an `Either`, choosing an appropriate error message based on the context.
- In the case of a success, the returned value is a set of constraints, where intuitively each constraint represents a restriction on a type variable. The starter code represents these as a tuple `(Type, Type)`, where the first type must be a type variable, and the second is some sort of restriction on that type variable. For example, we might represent the constraint "`a` must be an integer" with the value `(TypeVar "a", Int_)`.

Example. For the function call `identity 3`, the following happens (note that this is fairly abstract):

1. The type of `identity` is looked up, yielding `[Function [TypeVar "t2"] (TypeVar "t2")]`.
2. The type of `3` is resolved as `Int_`.
3. The `Int_` is *unified* with the type variable `TypeVar "t2"`, yielding the constraint set `{(TypeVar "t2", Int_)}`.
4. The return type `TypeVar "t2"` is calculated to be `Int_` using the generated constraint. We say that the type variable `t2` has been *resolved* as `Int_`.

Type representation and catching unification errors

The tuple-based representation of constraints on type variables has two main benefits: it's easy to explain, and easy to accumulate these constraints in code (just use `Set.union`). However, this representation alone is not sufficient for supporting type unification in general: it does not (directly) encode the *transitivity* that's essential to unifying types. For example, a set `{(TypeVar "a", Int_), (TypeVar "b", TypeVar "a")}` doesn't just encode the information that `a` is `Int_`, but that `b` is actually `Int_` as well. The set `{(TypeVar "a", Int_), (TypeVar "a", Bool_)}` is a *type error*: `a` cannot be both `Int_` and `Bool_` simultaneously.

Your major challenge on this assignment is to determine how to tackle these problems to both collect constraints on type variables and then use them to both detect errors and determine the types of variables. The starter code outlines one approach: accumulate the tuple-based constraints from calling `unify`, then after the constraints have all been collected, `consolidate` them into a new data structure (detecting errors along the way), and then use the new data structure to `resolve` type variables in the return type of the function being called. If you choose to follow this approach, it'll be up to you to decide on the data structure you use to `consolidate` the type

constraints—make sure to document this carefully, as this will be a part your TAs take a careful look at.

That said, here are two alternate approaches which you are free to use:

- Stick with the given data types, and write functions to both check for errors and resolve type variables.
- Design and implement a new data structure, and change the given type of `unify` to this completely new data structure instead, ignoring the original tuple-based `TypeConstraint`.

It's up to you whether you go with these, as we aren't testing any of your internal helpers (and data types) directly. But note that our expectations around good code design and documentation are, of course, still in effect.

Part 2: Function definitions and type inference

The second part of this assignment is extending your type checker to enable *function expressions* (i.e., lambdas) that can be used both in top-level definitions and as standalone expressions. The `Expr` constructor for lambda expressions should look very straight-forward, using strings to name parameters and taking a single body expression.

Far more interesting is how lambda expressions are type-checked. In the many mainstream statically-typed languages like Java, function definitions require explicit type annotations for all function parameters and return value (though Java's lambda expressions do not). In our exploration of Haskell to date, we've also written type annotations for all functions, as is good practice for documenting our code intent. However, it turns out that in Haskell type annotations are almost never necessary; when a type annotation is omitted, the Haskell compiler is still able to type-check the function, and in fact computes the function's type for later use in type-checking when the function is called.

The ability of a type checker to statically compute the types of expressions is known as **type inference**, and is one of the Haskell type system's major features. (As an aside, modern iterations of Java and other statically-typed languages are adopting at least partial type inference in their type systems.) In this part of the assignment, you'll add some limited type inference functionality to your type checker to support the *static typing of lambda expressions*, without requiring any type annotations!

Implementing this requires the following modifications and additions to your code:

1. When doing type checking, it's no longer enough to simply return a type; instead, both types and type constraints must be returned, so that type constraints can be accumulated recursively through an expression. Create a new type called `TypeCheckResult'` (or a different name) that captures this, and modify the type of `typeCheck` to return a `TypeCheckResult'` instead. This will require modifications to your existing code.

   Warning: for testing purposes, we'll only be looking at the return value of `runTypeCheck`, so *don't change the original `TypeCheckResult` type definition, or type signature of `runTypeCheck`.* But of course you can (and should) change its implementation to account for the change in `typeCheck`.

   Since this is an implementation change, before moving on we strongly encourage you to test your code thoroughly to ensure your change hasn't broken any of the existing functionality.

2. When type-checking a `Lambda` expression, first generate fresh type variables for each of the function parameters, and update the type environment to store these. Then type-check the body of the function with this updated environment, accumulating type constraints on these

new type variables. Finally, if the body type-checks, use the accumulated constraints to determine a (function) type for the `Lambda` expression as a whole.

For example, suppose we have the following expression:

```
\x -> x < 3
```

It would be represented by our `Expr` data structure as

```
Lambda ["x"] (Call (Identifier "<") [Identifier "x", IntLiteral 3])
```

First, your type checker should generate a fresh type variable for the `x`, say `TypeVar "tvar_x"`. It should then type-check the body `x < 3`, yielding the type `Bool_` (for the type of the entire body, since that's what `<` returns), and the constraint `(TypeVar "tvar_x", Int_)`, generated because `<` expects two integers as arguments. Finally, your type checker should use this information to deduce a final type signature of `Function [Int_] Bool_` for this function.

We've deliberately given you only a high-level view of this idea because we want you to work out the details, including deciding exactly how to implement this. Getting this to work together with Part 1 is almost certainly the most algorithmically-complex task of the course, and a worthy challenge for your extended work in Haskell!

Notes:

1. This is a form of *local* type inference: only the body of the function is used to determine its type. One consequence of this is that you shouldn't accumulate type constraints or type variables between top-level definitions.
2. While we strongly recommend starting with inferring the types of non-polymorphic function first (for arbitrary numbers of parameters), eventually your code should be able to detect and infer the types of generically-polymorphic functions as well.
3. Working on this might prompt you to modify your implementation for the "type unification" functions/types. That's fine, just don't forget to test your code thoroughly for Part 1 if you make changes!

For course-related questions, please contact **david at cs.toronto.edu**.