Tutorial 4

The goal of this tutorial is to provide students with more experience in writing interpreters and also introduce them to type-checking and abstract interpretation.

Tutorial Bundle

For this tutorial there is a bundle of files that can be downloaded from Moodle. Look for Bundle for Tutorial 4 under section *Tutorials and Assignments*.

Type-Checking and Abstract Interpretation

So far we have been focused on writing interpreters for small languages. An interpreter is a program that given expressions in a certain language, it outputs the value denoted by that expression. The value of an expression is a piece of information that is as precise as it can be regarding the result of executing that program. In other words, when we evaluate an expression like the following:

```
evaluate (3 + 5) ==> 8
```

we cannot be more precise about the result of this particular program: the expression $3 + 5$ evaluates only to the (concrete) number 8. The evaluate function implements what is called a **concrete** interpreter.

However it is possible to write interpreters that return **abstract** values. Those interpreters, called **abstract interpreters**, return some abstraction of the result of executing a program.

The most common and familiar example of abstract interpretation is type-checking or type-inference. A type-checker analyses a program in a language, checks whether the types of all sub-expressions are compatible and returns the corresponding type of the program. For example:

```
tcheck (3 + 5) ==> Int
```

A type-checker works in a similar way to a **concrete interpreter**. The difference is that instead of returning a (concrete) value, it returns a **type**. A type is an **abstraction of values**. When we see that the type of an expression is `Int`, we do not know exactly which concrete number will that expression evaluate to, but we do know that it will evaluate to an integer number and not to a Boolean value.

Type-checking is not the only example of abstract interpretation. In fact abstract interpretation is a huge area of research in programming languages because various forms of abstract interpretation are useful to prove certain properties about programs.

A Language with Boolean Values

Before we embark on writing a type-checker, we will first augment our small language from last tutorial to have Boolean values (`true` and `false`). Boolean values alone don't seem much interesting. We will also add logical operators (e.g., `&&`, `||`, `!`), comparison operators (e.g., >, >=, <, <=, ==) and conditional execution (`if-then-else` expression).

The augmented language we are going to use is:

```
data BinaryOp = Add | Sub | Mult | Div | Power
              | And | Or  | GT   | LT  | LE
              | GE  | EQ

data UnaryOp = Neg | Not

data Value = IntV  Int   -- Integers
           | BoolV Bool  -- Boolean values

data Exp = Literal Value
         | Unary UnaryOp Exp
         | Binary BinaryOp Exp Exp
         | If Exp Exp Exp
         | Var String
         | Decl String Exp Exp
```

You may notice that in this tutorial the representation of operators is slightly different from before. We separate unary operators from binary operators and pull them out into different datatypes. These changes are for the benefit of keeping `Exp` data type small and less clustered as more features added to our language.

The abstract syntax of the language now has 9 new operators in 3 categories:

- Logical operators: `&&`(`And`), `||`(`Or`), `!`(`Not`)
- Comparison operators: >(`GT`), <(`LT`), <=(`LE`), >=(`GE`), ==(`EQ`)
- Conditional execution: `if-then-else` expression

Also there are two kinds of values in the language: integers (e.g., `IntV 3`) and Boolean values (e.g., `BoolV True`).

With these changes, our language now seems more like what we would expect from a programming language. Now you can write programs like:

```
var x = 3; var y = 4; if (x >= y) x + 1; else y + 1 // return 5

var x = true; var y = false; (x || y) && (!x || !y) // x XOR y
```

Note that the concrete syntax for an `if` expression is:

```
if <condition> <exp>; else <exp>
```

Question 1: The *set of free variables* of an expression corresponds to the variables that are not bound by a variable declaration. For example, for the expression:

```
x * x
```

`{x}` is the set of free variables. For the expression:

```
var x = 3; x * y * z
```

`{y,z}` is the set of free variables. Here `x` is not free because there is a declaration of `x`.

Finally, for the expression:

```
var x = y; var y = 3; x + y
```

`{y}` is the set of free variables. In this case, even though there is a variable `y` declared inside the expression, the first occurrence of `y` is not bound anywhere.

Now you are asked to define the function `fv` that calculates the set of free variables of an expression. Go to the file *Interp.hs*, you will find the following type:

```
fv :: Exp -> [String]
```

Note: You can use the following two auxiliary Haskell functions in the definition of `fv`:

```
union :: [String] -> [String] -> [String]
delete :: String -> [String] -> [String]
```

The `union` function takes two lists of strings and returns a list that contains all strings in the two list parameters without repeated elements. In other words `union` is similar to `++`, except that removes duplicated elements. For example:

```
> ["a","b"] ++ ["b","c"]
["a","b","b","c"]
> union ["a","b"] ["b","c"]
["a","b","c"]
```

The `delete` function deletes a value from a list. For example:

```
> delete "x" ["a","x","y"]
["a","y"]
```

As before, whenever we add new features to our language, we'll have to rethink about how evaluation works in the language. This time, it is slightly more complicated than before, due to the fact that we changed the representation of operators. Also we have Boolean values in the language, so evaluating an expression can either return an integer or a Boolean value.

To bridge the gap between different operators and values, we will define two auxiliary functions as follows:

```
unary :: UnaryOp -> Value -> Value
binary :: BinaryOp -> Value -> Value -> Value
```

Given the type signatures, the above two functions should be self-explanatory.

Question2: Complete the definition of `unary` and `binary` in the file *Interp.hs*.

Note: You should be cautious about exhausting all possible combinations of operators and values. For example, the binary operator `Add` expects its two operands to be integers. Therefore you can take for granted that `Add` is undefined for Boolean values.

Now we are ready to implement the evaluation function again. Same as the last tutorial, we will need an *enviroment* as a collection of *bindings*. Go to the file *Interp.hs*, you will find the following:

```
evaluate :: Exp -> Value
evaluate e = eval e [] -- starts with an empty environment
  where
    eval :: Exp -> Env -> Value
    eval = "TODO: Question 3"
```

Question 3: Define the missing cases for the interpreter function `eval`. Remember to use `unary` and `binary` functions in the cases of `Unary` and `Binary` constructors, respectively.

If you play around the new evaluation function for a while, chances are that you will encounter something as follows when you mistakenly enter some invalid expressions:

```
*Interp> calc "1 + true"
*** Exception: Non-exhaustive patterns in function binary
```

It is not surprising, isn't it? We are setting ourselves on fire by adding integer to a Boolean value. It would be better if our interpreter can spot that and reject such programs without ever running. This is where type-checking comes onto stage.

Types Do Matter

For this language types are represented as:

```
data Type = TInt | TBool
```

This data type accounts for the two possible types (i.e., integer and Boolean) in the language.

So what do we mean by type-checking? In the tutorial, when we say type-checking, we refer to *static type-checking*. In other words, we verify the type safety of a program based on analysis of a program's text (source code). If a program passes the static type-checking, then the program is guaranteed to satisfy some set of type-safety properties [1]. For example, type-checking can fail when the types of sub-expressions are incompatible:

```
1 + true
```

should fail to pass type-checking, because addition is an operation that expects two integer values. In the above, the second argument is not an integer, but a Boolean.

In a language with variables, a type-checker needs to track all the types of bound variables. To do this we can use what is called a **type environmentnt**:

```
type TEnv = [(String, Type)]
```

We are going to use the following type for the type-checker:

```
tcheck :: Exp -> TEnv -> Maybe Type
```

Note how similar this type is to the type of an environment-based interpreter except for two differences:

[1] See Wikipedia

1. Where in the concrete interpreter we used `Value`, we now use `Type`.
2. The return type is now `Maybe Type`

This is because 1) if we look at `tcheck` as an abstract interpreter, then types play the role of abstract values; 2) use `Maybe` type make the code more robust when tracking for type-checking errors.

To begin with, we also need two auxiliary functions, just like `unary` and `binary`.

```
tunary :: UnaryOp -> Type -> Maybe Type
tbinary :: BinaryOp -> Type -> Type -> Maybe Type
```

Question 4: Go to the file *TypeCheck.hs*, implement the above two functions.

Typing Rules for Expressions

To type-check expressions in our language, we need a set of typing rules:

- **Typing Arithmetic Expressions**: Type-checking arithmetic expressions is fairly obvious. For example, to type-check an expression of the form:

  ```
  e1 + e2
  ```

  we proceed as follows:

  1. check whether the type of `e1` is `TInt`
  2. check whether the type of `e2` is `TInt`
  3. If both types are `TInt` return `TInt` as the result type (`Just TInt`); otherwise fail to type-check (`Nothing`)

  You should be able to handle other cases in a similar way.

- **Typing Declare Expressions**: To type-check a declare expression of the form:

  ```
  var x = e; body
  ```

  we proceed as follows:

  1. type-check the expression `e`
  2. if `e` has a valid type, then type-check the `body` expression with the type environment extended with a pair `(x, typeof e)`.

  For expression with unbound variables, you should return `Nothing`. In other words, the type-checker works only for closed (i.e., no free variables) expression.

- **Typing If Expressions**: The only slightly tricky expression for type-checking is an `if` expression. The typing rule for an `if` expression of the form:

  `if e1 e2; else e3`

  is

  1. check whether the type of `e1` is `TBool`
  2. compute the type of `e2`
  3. compute the type of `e3`
  4. check whether the types of `e2` and `e3` are the same. If they are the same return that type; otherwise fail.

  Note that if type-checking any sub-expressions fails with a type-error (`Nothing`), then the type-checking of the `if` expression will also fail.

Question 5: Implement the function `tcheck`.

Putting All Together

Now let's combine the type-checker with the evaluate function: given an expression, we will first type-check it, if it passes, then we evaluate it; otherwise we issue an exception with an error message.

```
tcalc :: String -> Value
```

`tcalc` has the same type with `calc`. However, it has the ability to reject ill-typed programs!

Here are some examples:

```
*TypeCheck> tcalc "3 == 3"
true

*TypeCheck> tcalc "if (3 == 4) true; else false"
false

*TypeCheck> tcalc "var x = 3; x + true"
*** Exception: You have a type-error in your program!
```

Question 6: Complete the definition of `tcalc`.

Assignment 3

Your third assignment is to try to complete as many questions in the tutorial as you can. All questions are **graded**. You should submit your solutions to Moodle. Look for Submission of Tutorial 4 under section *Submissions and Score Reporting*.

The deadline for this assignment is **Oct 22, 11:55 PM**.

Please submit your answers using the modified Haskell files. **Try to ensure that the files you submit compile. If you submit files that do not compile, points will be removed from your grade. Try to comment the code explaining what you intended to achieve. If you cannot create code that compiles, it is especially important to write some comments explaining what you intended to achieve.**