The University of Hong Kong

COMP3259: Principles of Programming Languages

Tutorial 4

Tutor
Wan Qianyong `qywan@cs.hku.hk`

Instructor
Bruno Oliveira `bruno@cs.hku.hk`

4 March 2025

Table of Contents

1 Introduction

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.

2 Type-Checking and Abstract Interpretation

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

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

The result of this particular program cannot be more precise: the expression $3 + 5$ just evaluates 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*, output some abstraction of the result of executing a program.

The most common example of abstract interpretation is type-checking or type-inference. A type-checker analyzes an expression, checks whether the types of all sub-expressions are compatible, and returns the corresponding type of the expression. For example:

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

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

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.

3 A Language with Boolean Values

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

The augmented language we are going to use is:

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

data UnaryOp = Neg | Not

data Value = IntV  Int
           | BoolV Bool

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 are added to our language.

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

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

There are also 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  // returns 5
var x = true; var y = false; (x || y) && (!x || !y)  // returns true
```

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

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

As before, whenever we add new features to our language, we will have to rethink how evaluation works in the language. This time, it is slightly more complicated than before because we changed the representation of operators. Moreover, we have Boolean values in the language, so evaluating an expression can return either 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 two functions above should be self-explanatory.

Question 1. 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 *environment* as a collection of *bindings*. Go to the file `Interp.hs`, you will find the following definitions:

```
evaluate :: Exp -> Value
evaluate e = eval e []  -- starts with an empty environment
  where eval :: Exp -> Env -> Value
        eval = error "TODO: Question 2"
```

Question 2. Fill in the missing part of the interpreter function `eval`. Remember to use `unary` and `binary` functions in the cases of `Unary` and `Binary` constructors, respectively.

If you play around with the new evaluation function for a while, you may encounter something as follows when you mistakenly enter some invalid expressions:

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

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

3.1 Types Do Matter

For our small language, types are represented as:

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

This datatype 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 source code. If a program passes the static type-checking, then the program is guaranteed to satisfy some set of type-safety properties.

For example, `1 + true` does not type-check because an addition expects two Integers. In the expression above, the second argument is not an Integer but a Boolean value.

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

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

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

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

You may notice the similarity between this type signature and that of an environment-based interpreter, except for two differences:

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

This is because 1) if we look on `tcheck` as an abstract interpreter, then types play the role of abstract values; 2) using `Maybe` makes the code more robust when tracking 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 3. Go to the file `TypeCheck.hs` and implement the two functions above.

3.2 Typing Rules for Expressions

To type-check expressions in our language, we need a set of typing rules. We will go over some examples, together with their associated inference rules. The inference rules are composed of typing judgements of the form:

$$\Gamma \vdash e : T$$

where Γ is a type environment, e is an expression, and T is the type being checked. We may read it as *the expression e has type T, under the type environment Γ*. Possible ways to construct an environment are formally described as:

$$\Gamma \ ::= \ \cdot \mid \Gamma, x : T$$

It means that an environment can be either the empty set ($\cdot$) or an environment extended with a variable x of type T. In short, an envionment is a collection of pairs of variable names and their types.

Now, we are ready to look at the set of typing rules:

- **Typing Arithmetic Expressions**: Type-checking arithmetic expressions is fairly trivial. 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` then 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. The inference rule for the description above can be formalized as:

$$\frac{\Gamma \vdash e_1 : \mathbb{Z} \qquad \Gamma \vdash e_2 : \mathbb{Z}}{\Gamma \vdash e_1 + e_2 : \mathbb{Z}}\ \text{T+}$$

Similar rules can be derived for other binary operations. For unary operations, one can write, for example:

$$\frac{\Gamma \vdash e : \mathbb{Z}}{\Gamma \vdash -e : \mathbb{Z}}\ \textsc{TNeg}$$

- **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* expressions (i.e. with no free variables.) Inference rules for declarations and variables can be formalized as:

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma,\, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash (\texttt{var}\ x = e_1;\ e_2) : T_2}\ \textsc{TDecl}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{TVar}$$

- **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 so then return that type; otherwise type-checking fails.

Note that if any sub-expression fails to type-check (`Nothing`), then the whole `if` expression will also fail to type-check.

The inference rule for `if` expressions can be formalized as:

$$\frac{\Gamma \vdash e_1 : \mathbb{B} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash (\texttt{if}\ e_1\ e_2;\ \texttt{else}\ e_3) : T}\ \textsc{TIf}$$

Question 4. Implement the function `tcheck` according to the typing rules above.

Now consider the following expression:

```
var x = 1; 2 + x
```

Supposing that we want to derive its type, we may use the set of inference rules previously defined to formally type-check this expression, as in:

$$\frac{\Gamma \vdash 1 : \mathbb{Z} \qquad \dfrac{\Gamma, x : \mathbb{Z} \vdash 2 : \mathbb{Z} \qquad \dfrac{x : \mathbb{Z} \in (\Gamma, x : \mathbb{Z})}{\Gamma, x : \mathbb{Z} \vdash x : \mathbb{Z}}\ \textsc{TVar}}{\Gamma, x : \mathbb{Z} \vdash 2 + x : \mathbb{Z}}\ \textsc{T+}}{\Gamma \vdash \texttt{var}\ x = 1;\ 2 + x : \mathbb{Z}}\ \textsc{TDecl}$$

This is very similar to what you have done previously for evaluation. We can see the correspondence between concrete interpretation and abstract interpretation.

3.3 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 raise an exception with an error message.

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

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 5. Complete the definition of `tcalc`.

With the help of type-checker, our evaluator can now successfully reject these ill-formed programs that get stuck in evaluation. The type system gives us a valuable guarantee: **well-typed programs cannot go wrong**!