The University of Hong Kong

COMP3259: Principles of Programming Languages

Tutorial 5

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

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

18 March 2025

Table of Contents

1 Introduction

In this tutorial, we continue with the language with Boolean values added in Tutorial 4. We will introduce new notions to reason about free and bound variables. This tutorial will deepen your understanding of lexical scoping.

2 Free Variables and Closed Expressions

Sometimes we are going to check whether an expression is legal. One of these cases is that some variables exist without any declaration in advance. These illegal variables are called *free variables*.

The set of free variables within an expression contains the variables that are not bound by any 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.

Question 1. Now you are asked to define the function `fv` that calculates the set of free variables within an expression. Go to the file `Interp.hs`, you will find the following type signature:

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

Hint: You may use the following two auxiliary Haskell functions:

```
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 lists without repeated elements. In other words, `union` is similar to `++`, except that it 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"]
```

If an expression has no free variable, we call it a *closed expression*. In other words, a closed expression implies that all variables within it are bound by variable declarations.

Question 2. Complete the definition of `closed` in `Interp.hs`.

```
closed :: Exp -> Bool
```

3 Nameless Representation

Sometimes, we may have to deal with expressions with free variables. For example:

```
var x = y; var y = 1; x + y
```

If we follow the conventional evaluation rule to substitute x with variable y, the expression will evaluate to:

```
var y = 1; y + y
```

Which is wrong because the y is a free variable in the expression that x is being substituted into, but after the substitution it becomes bound to variable declaration `var y = 1`. In other words, y is *captured*. We want to avoid such variable capture by *Capture-Avoiding Substitutions*.

3.1 Renaming

Note that we give every variable a name in our language. In fact, every bound variables can be renamed **to a fresh name** without changing the semantics of expressions, which effectively prevents unexpected capture. For example, consider the expression we show above:

```
var x = y; var y = 1; x + y
```

We can replace the inner y by z:

```
var x = y; var z = 1; x + z
```

Although different variable names are used, they are obviously two equivalent expressions under the notion of *α-equivalence*. After substitution of x, we get:

```
var z = 1; y + z
```

As shown above, we can rename a variable to a fresh name every time we find a name conflict. However, in this way, we should maintain a set of used names, which is troublesome.

3.2 De Bruijn Indices

Instead of renaming variables, we suggest a more straightforward (but less readable) idea to rewrite the expression: referring to a variable by the distance to its binder. The distance is defined as the number of occurrences of other binders in-between. This approach is well known as *De Bruijn indices*.

Here, we replace named variables by De Bruijn indices. Each index is a natural number, corresponding to the distance to its binder. For example, the expression:

```
var x = 2; var x = x; x + 3
```

can be translated to

```
var = 2; var = {0}; {0} + 3
```

In this example, both x's are replaced by $\{0\}$, referring to their respective closest binders (note that they are not the same.) Another example is:

```
var x = 4; var y = 5; var z = 6; x + y + z;
```

This code can be translated to:

```
var = 4; var = 5; var = 6; {2} + {1} + {0};
```

Here, $\{0\}$ represents z because there is no binder between the declaration and the occurrence of z; $\{1\}$ represents y because there is one binder (z) between the declaration and the occurrence of y; and $\{2\}$ represents x because there are two binders (y and z) between the declaration and the occurrence of x.

3.3 Target Syntax

Now we have a better representation of expressions, it's time to implement the translation algorithm from the source language (using names) to the target language (using indices).

First of all, we need to define the target language:

```
type Index = Int

data TExp = TLit Value
          | TUnary UnaryOp TExp
          | TBinary BinaryOp TExp TExp
```

```
  | TIf TExp TExp TExp
  | TVar Index
  | TDecl TExp TExp
  deriving Show
```

As you see above, we use `Int` as `Index` and add a new datatype `TExp` to represent the syntax of the target language. `TVar` now takes an index instead of a names, and `TDecl` does not include any variable name explicitly anymore.

What we going to do next is to translate the abstract syntax in the source language to that in the target language. For example, concerning the source code:

```
var x = 4; var y = 8; x + y
```

we want to translate the syntax from:

```
source :: Exp
source = Decl "x" (Literal (IntV 4))
  (Decl "y" (Literal (IntV 8)) (Binary Add (Var "x") (Var "y")))
```

to:

```
target :: TExp
target = TDecl (TLit (IntV 4))
  (TDecl (TLit (IntV 8)) (TBinary Add (TVar 1) (TVar 0)))
```

3.4 Translation Algorithm

The algorithm is quite straightforward: we maintain a list of variable names when traversing the source AST and building the target AST. Once we meet a binder, we add the bound name to the list. Once we meet a name reference, we determine its index by searching the list.

Question 3. You are asked to write a function that translates from the source language to the target language.

```
translate :: Exp -> TExp
translate source = convert source [] -- starts with an empty environment
  where
    convert :: Exp -> [String] -> TExp
    convert = error "TODO"
```

Since the AST of the target language is different from that of the source language, we need to write a new evaluation function.

Question 4. You are asked to write a new evaluation function for the target language.

```
tevaluate :: TExp -> Value
tevaluate e = teval e [] -- starts with an empty environment
  where
    teval :: TExp -> [Value] -> Value
    teval = error "TODO"
```

Question 5. Just like what were done in Tutorial 4, you are also asked to write a type checker for target language.

```
tcheck :: TExp -> TEnv -> Maybe Type
tcheck = error "TODO"
```