MP 4 - Type Inference

CS421 Agha - Spring 2015

rev. 1.1

Introduction

Cooperation: You may work in pairs for this MP ***only***, your partner may not be necessarily the same as the one for last MP.

Assigned: March 21, 2015
Due: 6:00pm, April 6th, 2015 **(the deadline is firm)**
Outline: This assignment covers material from chapter 7 of the EOPL book, 3rd edition.
Grading: This assignment will account for 8% of your grade.
Reference: The Github link below might be helpful. If you use any code not written by you, you need to cite the source in the comments .
`https://github.com/mwand/eopl3`

Academic honesty instructions can be found here:

`https://wiki.cites.illinois.edu/wiki/display/cs421sp15/Assignments?src=contextnavchildmode`

Pay special attention to the policies regarding the use of other people's code.

NOTE: The submission instructions have changed. Please refer to the above link for details.

Problems

For this MP, you need to write a procedure `type-of` in Scheme. This procedure will only have one argument, which would be an expression in the following language, represented as string. `type-of` should return the type of the expression.

In the grammar below, terminals are in courier font, for example, `newpair`. Nonterminals are italicized, as in *Expr*. Parentheses are literals (i.e. part of the specified grammar), while braces are part of the meta-language we use to describe the grammar. We use the * symbol to denote zero or more repetitions; for example, {(*Expr Var*)}* means zero or more occurrences of (*Expr Var*), including the parentheses. When it is clear from context, we omit the braces, so that *Var** means zero or more repetitions of *Var*.

Assume the language given by the following grammar:

Expr ::= `let` { *ID* = *Expr* }* `in` *Expr*
| `letrec` { *ID* = *Expr* }* `in` *Expr*
| `proc` ({*ID*}*) *Expr*
| (*Expr* {*Expr*}*)
| `begin` *Expr* {; *Expr*}* `end`
| `if` *Expr* `then` *Expr* `else` *Expr*
| *ArithmeticOp* (*Expr* {, *Expr*}*)
| *ComparisonOp* (*Expr* , *Expr*)
| `newpair`(*Expr*, *Expr*)
| `first` (*Expr*)
| `second` (*Expr*)
| *Integer*
| *ID*
| `true`
| `false`

Here, *ID* represents identifiers, that is, names of variables and procedures. You may assume they are simply strings of characters; *ArithmeticOp* denotes the usual +, -, *, / operators; *ComparisonOp* denotes comparison operators <, >, =. The arithmetic and comparison operators allow only integer operands. Note that the `let` here in our language works like `let*` in scheme

`newpair(e1 e2)` constructs a new pair, consisting of the two expressions `e1` and `e2`. Notice that the following is possible in our language:

```
newpair( 1, newpair ( 2, newpair ( 3, 4 ) ) )
```

Also, `first` and `second` return the first and second element of the given pair, respectively:

```
> let x = newpair (1, 2) in first(x)
1

> second( newpair (1, 2) )
2
```

For the various types you need, use the following datatype definition:

```
(define-datatype type type?
  (int-type)
  (bool-type)
  (proc-type
    (arg-types (list-of type?))
    (return-type type?))
  (pair-type
    (first-type type?)
    (second-type type?))
  (tvar-type
    (sym symbol?))
  (bad-type))
```

The intention of the above datatype is to provide a consistent representation for all types the program can produce. Pay attention to the form of `proc-type`: it contains the types of the arguments, as well as the type it returns. For example, what we usually represent as

`int-type` × `bool-type` → `int-type`

will be represented as:

```
#(struct:proc-type (#(struct:int-type) #(struct:bool-type)) #(struct:int-type))
```

Examples follow:

```
> (type-of "1")
#(struct:int-type)

> (type-of "true")
#(struct:bool-type)

> (type-of "let f = proc (x) +(x,1) in (f 2)")
#(struct:int-type)

> (type-of "newpair(1, true)")
#(struct:pair-type #(struct:int-type) #(struct:bool-type))

> (type-of "- (1, true)")
#(struct:bad-type)

> (type-of "> (false, 9)")
#(struct:bad-type)

> (type-of "= (true, true)")
#(struct:bad-type)

> (type-of "if =(1, 2) then 5 else false")
#(struct:bad-type)
```

```
> (type-of "let f=proc(x) +(x,1) in (f true)")
#(struct:bad-type)

> (type-of "let f=proc(x) x in if (f true) then (f 3) else (f 5)")
#(struct:int-type)

> (type-of "let f=proc(x) x in newpair((f true), (f 8))")
#(struct:pair-type #(struct:bool-type) #(struct:int-type))

> (type-of "proc (x) +(x, 1)")
#(struct:proc-type (#(struct:int-type)) #(struct:int-type))

> (type-of "letrec ill = proc(x) (ill x) in (ill 5)")
#(struct:tvar-type t)

> (type-of "(proc(y) if (y true) then (y 4) else 0  proc(x)  x)")
#(struct:int-type)

> (type-of "let x=5 in let f=proc() x in let x=true in (f)")
#(struct:int-type)
```

You may assume that all our programs do not have unbound identifiers. In other words, our tests will not include programs which have unbound identifiers.

We use a type variable to represent the type of a well-typed expression, whose type has not been instantiated yet. The type variable constructor takes a symbol as the only argument. The symbol is used to distinguish different type variables (i.e., different symbols indicate different type variables). You can define your own procedure to generate symbols for type variables (i.e., no restrictions on symbols).

Here is an example where a type variable is needed,

```
> (type-of "proc (x) x")
#(struct:proc-type (#(struct:tvar-type t)) #(struct:tvar-type t))
```

In the above example, the result is clearly a `proc-type`, but the type of the parameter has not been instantiated. Thus, we use a type variable `t` to represent the type of `x`. Notice that since this is an identity procedure, the type of the body should be the same as the type of the parameter. Hence, we need only one type variable to represent the type of both the parameter and the body. In your implementation, the symbol does not need to be `t`. We only check if you use the same type variable for both the parameter and the body by calling `equal?` on symbols of type variables.

However, it would be wrong if you use two different type variables for the parameter and the body, respectively. The following output would be wrong because type variable `t1` may not be equal to the type variable `t2`.

```
> (type-of "proc (x) x")
#(struct:proc-type (#(struct:tvar-type t1)) #(struct:tvar-type t2))
;; wrong if type variable t1 is not qual to type variable t2
```

Here is an example with two different type variables,

```
> (type-of "proc (x y) newpair(x, y)")
#(struct:proc-type
  (#(struct:tvar-type t1) #(struct:tvar-type t2))
  #(struct:pair-type #(struct:tvar-type t1) #(struct:tvar-type t2)))
```

We first create a type variable `t1` for `x`. Since the type of `y` is not necessarily the same as the type of `x`, we create another fresh type variable `t2` for `y`. Notice how each type variable appears twice, once as the type of a parameter, and once inside the `pair-type` (which is the output type of the function). Again, you are not required to use the same symbols `t1` and `t2` for these type variables. We only check if you use different type variables for the type of `x` and `y` by calling `equal?` on symbols of the type variables.

Also, observe that the above program is well-typed, even though the values of the two type variables are not precisely determined. However, using the above procedure in the following program is guaranteed to work correctly:

```
> (type-of "let f = proc(x y) newpair(x, y) in first( (f 1 true) )")
#(struct:int-type)
```

The value of the above `let` expression would be 1. We are hence dealing with *parametric polymorphism*.

NOTE: You do *not* need to implement the semantics of the above language. For this MP, we are only concerned with the automatic inference of types.