1 Type resolution

Ex. 1 (FA 08) This problem involves coming up with new typing rules, that is, new rules concerning the predicate typeof(E, T, Env). You may assume that we already have rules for the rest of the language, and that there are rules for predicate subtype(T₀, T₁) that make it mean “T₀ is a subtype of T₁.”

I wish to describe type rules for a map (dictionary) type. A map in this language takes keys of some particular type (call it the domain type) and produces values of some other type (call it the codomain type). So m[k], where m has a map type, produces a value of its codomain type (or some subtype of it) as long as k has the appropriate domain type (or some subtype of it). We’ll use the notation map(D, C) to mean “the type of maps with domain type D and codomain type C.”

(a) Give appropriate rules for the type of m[k], that is, appropriate Prologish rules for typeof(index(m, k), T, E) (where index(m, K) is the AST for m[k]).

Answer:

typeof(index(M, K), T, Env) :- typeof(M, map(D, C), Env), subtype(T, C), typeof(K, T₁, Env), subtype(T₁, D).

(b) Give appropriate rules for assignment to m[k]. Assignment always produces a Void value, so the problem is to give correctness rules for typeof(assign(m, k, v), void, Env), where assign(m, k, v) is the AST for m[k]=v.

Answer:

typeof(assign(M, K, V), void, Env) :- typeof(M, map(D, C), Env), typeof(K, T₁, Env), typeof(V, T₂, Env), subtype(T₁, D), subtype(T₂, C).

Ex. 2 Show the steps involved in performing type inference on the program. (1) List each call to Unify and Freshen, and (2) show the bindings (or errors) produced.

```python
def foo(f, l):
    if tl(l) = []
        return hd(l)
    else
        return f(hd(l), foo(f, tl(l)))
foo(lambda x,y: x+y, "Hello world!")
```

Answer:

Unify($foo, (f, $l)->$a0) # function definition
Unify($l, list($a1)) # list rest rule
Unify($a0, $a1) # return
Unify($f, ($a2, $a3)->$a4) # call to f
$Unify($a0, $a4) # return
$Unify($a2, $a1) # bind arg 1 to f
$Unify($a3, $a0) # bind arg 2 to f

$Freshen($foo) = (($b0, $b0)->$b0, list($b0)->$b0)
$Unify($lambda, ($x, $y)->$b1) # function definition
$Unify($x, int) # addition rule
$Unify($y, int)
$Unify($b1, int) # return
$Unify($lambda, ($b0, $b0)->$b0) # bind argument 1 of call
$Unify($b0, $x) # follows from lambda binding
$Unify($b0, $y) # follows from lambda binding
$Unify($b0, $b1) # follows from lambda binding
$Unify(list($b0), str) #arg2 # bind argument 2 of call
Error: Can’t bind list(int) -> str

2 Dynamic vs Lexical Scoping

Ex.2 Recall that a dynamically scoped variable is looked up in the call stack that is active at the moment when a name needs to be resolved and found in the most recent stack frame. By contrast, a lexically scoped variable is looked up according to the textual arrangement of their definitions.

Consider the following program

```python
x = 161
def add(y):
    return x + y
def f(x):
    return add(3)
```

(a) What does $f(5)$ return under lexically scoped rules? What about dynamically scoped rules?

**Answer:** Lexical: 164 Dynamic: 8
(b) Suppose our compiler from project 2 used dynamic scoping for variables. How would this change the compiler’s processing of declarations?

**Answer:** All variables would be declared in the global scope. This also inherently means that variables can assume different types during run time, so we cannot tie declarations to usage during compilation. We would need to implement a run time check to handle declaration resolution.

(c) How would this change the number of passes we need to perform over the AST for declaration resolution?

**Answer:** We would no longer need to resolveSimpleIds since all declarations point to the global environment.

### 3 Runtime organization

**Ex.2** Recall that in our runtime organization, methods maintain a dynamic link to the base of the stack frame of the method in which they are invoked, and a static link to the stack frame in which they are defined.

Consider the following program:

```python
def counter():
    d = {'x': 0}
    def incr(n):
        d['x'] += n
        return d['x']
    return incr
da = counter()
print da(23)
print da(47)
```

(a) What does the following code print? Why aren’t static links on the stack sufficient for implementing this program?

**Answer:** 23, 70. When count returns, it is removed from the stack, so incr will have nothing to point to. We need to store this higher order function on the heap.