CS 164 Programming Languages and Compilers **Spring 2019**

Discussion 4/8: Typing and Runtime Organization

Instructor: Paul N. Hilfinger GSIs: Nikhil Athreya, Vaibhav Sakore

1 Type Resolution

1. 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`(T_0, T_1) that make it mean "T_0 is a subtype of T_1."

   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 Prolog 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).
   ```

2. For a statically typed dialect of Python, give the type of the function `iterate` defined below in ML notation (using `'a`, `'b`, etc. for type variables, $A \to B$ for function types, $A*B$ for a tuple whose elements have types A and B, etc.) For a function that takes no argument and returns a value of type T, write the type as $() \to T$. *Show your reasoning* (which need *not* involve showing the application of the unification algorithm). Be sure to define the type of `iterate` itself plus any non-free type variables you introduce in the process.

```
def iterate(f, x):
  y = f(x)
  def g():
    return iterate(f,y)
  return tuple(y,g)
```

Answer: First, assign a fresh type to each identifier.

```
iterate :  'i1
f :  'f1
x :  'x1
g :  'g1
y :  'y1
```

Apply type constraints.

```
'i1 = 'f1 -> 'x1 -> 'r1 [iterate(f,x) is a function; r1 is fresh]
'f1 = 'x1 -> 'r2 [f is a function, as in y = f(x); r2 is fresh]
'y1 ='r2 [y = f(x)]
'x1  = 'y1 [iterate(f,y)]
'g1 = () -> 'r1 [g()]
'r1 = 'y1*'g1 [return tuple(y,g)]
```

Perform unification.

```
'x1 = 'y1
'y1 = 'r2
'f1 = 'r2 -> 'r2
'r1 = 'r2 * (() -> 'r1)
'i1 =  ('r2 -> 'r2) -> 'r2 -> 'r2 * (() -> 'r1)
```

2 Runtime Organization

1. 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:

```
def counter():
  d = {'x': 0}
  def incr(n):
    d['x'] += n
    return d['x']
  return incr

a = counter()
print a(23)
print a(47)
```

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

Answer:

23, 70. When `counter` returns, its stack frame is removed from the stack, so `incr` will have nothing to point to (there will be a dangling pointer). We need to store this closure on the heap.