Arizona State University

CSE 340 – Spring 2018

Homework 5

Due Monday April 9 by 11:59 PM

I. Type Equivalence. Consider the following type declarations in a fictitious language.

```
TYPE
      A1  : int;                        // integer
      A2  : bool;                       // boolean
      A3  : pointer to float;
      A4  : pointer to int;
      A5  : pointer to bool;
      A6  : pointer to A1;
      A7  : pointer to A2;
      A8  : structure { x : int; }
      A9  : structure { a : int; }
      A10 : structure { x : float; }
      A11 : structure { a : int ; b : float; }
      A12 : structure { b : float; a : int ; }
      A13 : structure { x : A1 ; b : float; }
      A14 : structure { x : bool ; b : int; }
      A15 : structure { x : A2 ; b : A1; }
      A16 : func ( x : int ) int;             // function of int that returns int
      A17 : func ( x : float ) int;           // function of float that returns int
      A18 : func ( x : int, y : int ) int;
      // A19 is a function that takes two parameters and return int. The first
      // parameter is a function that takes two int as parameters and return int
      // and second second parameter is int.
      A19 : func ( x : A18, int) int;
      A20 : func ( x : func (int, int) int, int) int;
      A21 : func ( x : func (int, float) int, int) int;
      A22 : func ( x : func (int, int) float, int) int;
      A23 : structure { a : pointer to A24;  b : pointer to A23; }
      A24 : structure { a : pointer to A23;  b : pointer to A24; }
      A25 : structure { a : pointer to A26;  b : A19;  c : A20; }
      A26 : structure { a : pointer to A25;  b : A20;  c : A19; }
      A27 : array [4][5] of A25;      // array 4 rows 5 columns
      A28 : array [4][5] of A26;
      A29 : array [5][4] of A26;
```

Determine which types are structurally equivalent.

(Note: the definition of structural equivalence that I gave in class is a permissive definition that few languages support and is different from the one in the textbook. You should use the definition I gave).

You do not need to show your work. Give your answer by filling the following tables.

Pointer Types

	A3	A4	A5	A6	A7
A3	T	F	F	F	F
A4	F	T	F	T	F
A5	F	F	T	F	T
A6	F	T	F	T	F
A7	F	F	T	F	T

Function Types

	A16	A17	A18	A19	A20	A21	A22
A16	T	F	F	F	F	F	F
A17	F	T	F	F	F	F	F
A18	F	F	T	F	F	F	F
A19	F	F	F	T	T	F	F
A20	F	F	F	T	T	F	F
A21	F	F	F	F	F	T	F
A22	F	F	F	F	F	F	T

Structure Types

	A8	A9	A10	A11	A12	A13	A14	A15	A23	A24	A25	A26
A8	T	T	F	F	F	F	F	F	F	F	F	F
A9	T	T	F	F	F	F	F	F	F	F	F	F
A10	F	F	T	F	F	F	F	F	F	F	F	F
A11	F	F	F	T	F	T	F	F	F	F	F	F
A12	F	F	F	F	T	F	F	F	F	F	F	F
A13	F	F	F	T	F	T	F	F	F	F	F	F
A14	F	F	F	F	F	F	T	T	F	F	F	F
A15	F	F	F	F	F	F	T	T	F	F	F	F
A23	F	F	F	F	F	F	F	F	T	T	F	F
A24	F	F	F	F	F	F	F	F	T	T	F	F
A25	F	F	F	F	F	F	F	F	F	F	T	T
A26	F	F	F	F	F	F	F	F	F	F	T	T

Array Types

	A27	A28	A29
A27	T	T	F
A28	T	T	F
A29	F	F	T

Selected Explanations

Functions

Two functions are structurally equivalent if an only if

1. They have the same number of parameters
2. The return types are structurally equivalent
3. The i'th parameter type in one function is structurally equivalent to the i'th parameter type of the other function, I = 1 to number of parameters.

In particular A18 = func (int , int) int A20 = func (func (int , int) int , int) int

If we list the types side by side, we have

- First parameter: int vs. func(int, int) int // not structurally equivalent

- Second parameter: int vs. int // structurally equivalent
- Return type: int vs. int // structurally equivalent

So, the two types are not structurally equivalent because the first parameter in A18 is not structurally equivalent to the first parameter in A20

Arrays

A28 and A29 are not structurally equivalent because the number of entries in the first dimension of A28 is 4 which is different from the number of entries in the first dimension of A29 which is 5.

Pointers

A1 is structurally equivalent to int because of the declaration A1 : int. It follows that A6 is structurally equivalent to A4 because both are pointers to structurally equivalent types.

II. Consider the following variable declarations in conjunction with the type declarations in problem I

```
VAR                              // var declaration section
      q : A27;
      r : A27;
      s : A28;
      t : A29;
      u : array [4][4] of A25;
      v, w : structure {
                  a : A19;
                  b : int;
            };

      x, y : structure {
                  a : A19;
                  b : int;
            };
      g : func (x : int , y : int) int
          { return x + y ;
          };
      h : A18;
      n : int;
```

Assume that assignments between variables are allowed if the types of the variables are equivalent. For each of the following, list all type equivalence schemes under which the expression is valid. Consider name equivalence, internal name equivalence, and structural equivalence for each case. Assume that if two variables are equivalent under name equivalence, they are also equivalent under internal name equivalence. Assume integer constants have type int. Give your answers in the form of a table where each entry is a YES or a NO

	Name	Internal Name	Structural
`q = r ;`	YES	YES	YES
`r = s ;`	NO	NO	YES
`s = t ;`	NO	NO	NO
`t = u ;`	NO	NO	NO
`u = v ;`	NO	NO	NO
`v = w ;`	NO	YES	YES
`w = x;`	NO	NO	YES
`h = g;`	NO	NO	YES
`n = g(3,4);`	YES	YES	YES
`n = x.a(g,3) ;`	NO	NO	YES
`n = x.a(h,3) ;`	YES	YES	YES

Selected Explanations

- `n = g(3,4) ;` This statement has two parts

1. Calling `g` with arguments `3` and `4`. `g` has two parameters of type `int`. The provided arguments are of type `int`, so the call is valid under name equivalence.
2. Assigning the return value of `g` to `n`. The value returned by `g` is of type `int` and the type of `n` is `int`, so the assignment is valid under name equivalence.

Since both the all and the assignment are valid under name equivalence, the whole statement is valid under name equivalence.

- `n = x.a(g,3) ;` This statement has two parts

1. Calling `x.a()` with arguments `g` and `3`. `x.a()` expects two parameters, one of type `A19` and one of type `int`. `g` has two parameters of type `int` and returns an `int` which is structurally equivalent to `A19` but is not name equivalent or internal name equivalent to `A19` . Value `3` is of type `int` which is name equivalent to the second parameter expected by `x.a()`. So, the call is only valid under structural equivalence.
2. Assigning the return value of `x.a()` to `n`. The value returned by `x.a()` is of type `int` and the type of `n` is `int`, so the assignment is valid under name equivalence.

We conclude that the whole statement is valid under structural equivalence only.

III. For each of the following definitions, give the type of the function and its arguments. If there is a type mismatch, you should explain the reason for the mismatch. Note that for recursive functions, **`let rec`** is used.

a. `let f a = 1:`
   `Tf = T -> int`
   `f is a function that takes a parameter a of any type and returns an int value 1`

b. `let f a = a`
   `Tf = T -> T`
   `f is a function that takes a value of a variable a of any type and returns the same value as the result`

c. `let f a b = a + b + 1.0`
   `Type mismatch`
   `+ cannot be applied to 1.0 because 1.0 is float not int`

d. `let f a b = a b`
   `Tf = (T1 -> T2) -> T1 -> T2`
   `a is applied to be, so it must be a function, that takes b as a parameter, say b is of type T2, and returns a value of type T2`

e. `let f a b c = c.(a+b)`
   `Tf = int -> int -> (T array) -> T`
   `c is an array because it is used as an array and f returns the element c.(a+b) of c whose type is T. a and b are both int because they are added together using the + operator for int. The array index is also an int which is consistent with the types of a and b`

f. `let f a b c = c.(a+b) + a`
   `Tf = int -> int -> (int array) -> int`
   `This is similar to the previous question. One difference is that the element c.(a+b) is added to a. So, we conclude that the element must be int and the return type of f is also int.`

g. let f a b c = if c.(a) then a else b

Tf = int -> int -> (bool array) -> int

Here c is also an array but the element c.(a) is used as a condition for an if expression. We conclude that c must be a (bool array). a is used an index, so it must be int. b must also be an int because b and the if expression result must have the same type as a.

h. let f a b c = if c.(a.(b)) then a else b

type mismatch

As before c must be a (bool array). a must also be an array because it is used as an array in the expression a.(b). The expression a.(b) is used as an index in the expression c.(a.(b)), so we conclude that type of a is (int array). b must be int because it is used as an array index. The type of the if expression must be the same as the type of a and b, the two branches of the if expression. But type of a = int array and type of b = int cannot be the same, so we have a type mismatch.

i. let f a b c = a.(c) <= b +1

Tf = (int array) -> int -> int -> bool

Here b must be int because it is used in the expression b+1. a.(c) is compared to b+1, so a.(c) must also be int. The expression a.(c) is int so a must be (int array) and c must be int. The return value of the function is the result of the comparison, so it is bool.

j. This has two part. The first part declares an array a and the second part gives the function whose type you need to determine

1. let a = [| 1; 2; 3; 4 |]
2. **let f i j = a.(i) +. j**

Type mismatch

The first expression initializes a as an (int array). The expression a.(i) must therefore be int. But a.(i) is added to j using the +. operator for float, so a.(i) it must be float. This conflicts with the fact that a.(i) is int, so we have a type mismatch.

k. let rec f a b c = if a = 0 then 1 else (f (a-1) b c) * b * c

Tf = int -> int -> int -> int

f 3 3 4 = 1728

f computes (b*c)^a (b times c to the power a