Principles of Programming Languages - Homework II (Solutions)

K. Subramani
LDCSEE,
West Virginia University,
Morgantown, WV
{ksmani@csee.wvu.edu}

1 Problems

1. Consider the following **C** fragment. Categorize the three assignments as legal or illegal, providing a justification for your answer.

```
{
   int x, *y, z[3];

(1)    &x = (int *)malloc (sizeof(int));
(2)    z= (int *)malloc(sizeof(int)*3);
(3)    *y = 3;

}
```

Solution:

(1) This assignment is illegal, since addresses of variables are assigned by the system and cannot be reassigned.

(2) This assignment is illegal for the same reason as above; on encountering the declaration

```
int z[3];
```

the system allocates spaces (statically) to the array variable z. This space cannot be reassigned.

(3) This assignment is legal and will not result in a compilation error. However, at run time, there could be a memory error; since y may not be pointing to a location under the program's control.

□

2. Consider the following **C** declarations:

```
{
  int x[10];
  int y[5];
}
```

Are x and y type equivalent in **C**? How is the expression $(x == y)$ handled by the **C** compiler?

Solution: Both x and y are array variables; as per **C**'s type equivalence rules, x and y must be type equivalent.

As per **C** semantics, x and y are also implicit constant pointers, which point to $x[0]$ and $y[0]$ respectively. The expression $(x == y)$ is therefore legal and will always return **false**, since $x[0]$ and $y[0]$ are stored in distinct memory locations. □

3. Assume that **C** uses strict type-checking, i.e., there is no type conversion without explicit casting. Further, assume that **C** has a `bool` data type. Use the Hindley-Milner type checking algorithm to derive the most general type for the following **C** function:

```
fact(n)
{
  if (n == 0)
    return 1;
  else
    return n*fact(n-1);
}
```

Solution: We first assign the type $\alpha \rightarrow \beta$ to the function $fact()$ to indicate that $fact()$ receives an object of type α and returns an object of type β. It follows that the type of n is α. If the expression $(n == 0)$ is to be type correct, then n (and hence α) must be an integer, (since in **C**, the float equivalent of 0 is denoted by 0.0). Since the function returns 1 if the test $(n == 0)$ succeeds, it follows that β must be an integer as well. Thus, as per the Hindley-Milner type checking algorithm, we can deduce the type of $fact()$ to be $int \rightarrow int$. This conclusion is verified in the **else** portion of the **if** statement; since $n * fact(n-1)$ must be integral as per the above discussion, and n is integral, $fact(n-1)$ must return an integer for type correctness to hold.

□

4. Consider a variant of the **C** language, called **C**$*$ in which only the **do - while** loop is specified as part of the syntax. Show how you would capture the semantics of **while** and **for** statements, using the **do - while** construct.

Solution:

(i) The **while** statement has the following syntax:

```
while (e)
  {
    S;
  }
```

We cannot replace it with the structure:

```
do
  {
    S;
  } while (e);
```

since the control expression e may be **false** prior to entering the **while** loop. However, the following modification works:

```
if (e) {
  do
   {
     S;
   } while(e);
       }
```

Note that S is executed if and only if (and only as long as) the control expression e evaluates to **true**.

(ii) The typical **for** loop has the following structure:

```
for( e1; e2; e3) {
  S;
 }
```

where $e1$ is the initializer; $e2$ is the control expression and $e3$ is the update. As discussed in class, we can replace the above structure with the following **while** loop:

```
e1;
while(e2)
   {
        S;
        e3;
   }
```

The **while** loop can then be represented by an equivalent **do-while** structure using the technique discussed in (i).

□

5. (i) Briefly explain the difference(s) between the normal and applicative orders of evaluation.

   (ii) Assume that **C** does not use short-circuiting in using the **and** operator. Professor Kurtowski attempts to remedy this situation by writing the following function for the **and** operator:

```
int and( int a, int b)
 {
   return a ? b : 0;
 }
```

Will the professor's technique work, given the semantics of the **if**-expression in **C**? Justify your answer.

Solution:

(a) In the normal order of evaluation (also called delayed order), each operator (or function) begins its evaluation before its operands are evaluated and an operand is evaluated only if its value is needed for the calculation of the value of the operations.

In the applicative order of evaluation, all operands **must** be evaluated before the operator commences evaluation.

In the absence of side-effects, both orders produce identical results when evaluating an expression.

(b) This technique will not work, since **C** uses applicative order for evaluating functions; accordingly, all the parameters of the **and** function (viz., a and b) will be evaluated before its body is entered. However, this violates the intent of short-circuiting which requires that b be evaluated only if a is **true**.

□