PL HW2

Alex Lin

December 2020

1

1.a

(1) $(\lambda x.E)[M/x] = (\lambda x.E)$
(2) $(\lambda x.E)M = E[M/x]$

In (1) a substitution is performed, all free occurrences of x in E will be substituted by M. Since x is bounded here, there are no free occurrences of x in E, thus no substitution is performed. Note that here the output is a lambda function.

In (2) a lambda reduction is performed, by definition it is equivalent to $E[M/x]$. Note that here the output is an argument.

1.b

Y combinator can be represented by $(\lambda h.(\lambda x.h(x\,x))(\lambda x.h(x\,x)))$.

1.c

Yf
$= (\lambda h.(\lambda x.h(x\,x))(\lambda x.h(x\,x)))f$
$= (\lambda x.f(x\,x))(\lambda x.f(x\,x))$
$= f((\lambda x.f(x\,x))(\lambda x.f(x\,x)))$
$= f(\lambda h.(\lambda x.h(x\,x))(\lambda x.h(x\,x))f)$
$= f(Yf)$

1.d

I will write a recursion function called sum, that gives the sum of all integers between parameter x and integer 3, inclusive.

Let sum be $Y(\lambda f.\lambda x.\,\text{if}\,(=x\,3)\;3\;(+\,x\;(f(-\,x\;1))))$.

Using the function definition,

$sum\;5$
$= Y(\lambda f.\lambda x.\,\text{if}\,(=x\,3)\;3\;(+\,x\;(f(-\,x\;1))))5$
$= (\lambda f.\lambda x.\,\text{if}\,(=\;x\,3)\;3\;(+\,x\;(f(-\,x\;1))))(Y(\lambda f.\lambda x.\,\text{if}\,(=\;x\,3)\;3\;(+\,x\;(f(-\,x\;1))))\;5)$
$= (\lambda x.if\;(=\;x\;3)\;3\;(+\,x\;(sum(-\,x\;1)))\;5)$
$= \text{if}\,(=\;5\;3)\;3\;(+\,5\;(sum(-\,5\;1)))$
$= \text{if}\,(=\;5\;3)\;3\;(+\,5\;(sum\;4))$
$= (+\,5\;(sum\;4))$
$= (+\,5\;(Y\;(\lambda f.\lambda x.\,\text{if}\,(=\;x\;3)\;3\;(+\,x\;(f(-\,x\;1)))))\;4)$
$= (+5((\lambda f.\lambda x.\text{if}(=x3)3(+x(f(-x1))))(Y(\lambda f.\lambda x.\text{if}(=x3)3(+x(f(-x1)))))4))$
$= (+\,5\;((\lambda x.\,\text{if}\,(=\;x\;3)\;3\;(+\,x\;(sum(-\,x\;1))))\;4))$
$= (+\,5\;(\;\text{if}\,(=\;4\;3)\;3\;(+\,4\;(sum(-\,4\;1)))))$
$= (+\,5\;(+\,4\;(sum(-\,4\;1))))$
$= (+\,5\;(+\,4\;(sum\;3)))$
$= (+\,5\;(+\,4\;3))$
$= (+\,5\;7)$
$= 12$

1.e

The first Church-Rosser Theorem:

Given an expression, for any two terminating reduction sequences on the expression, α and β, the normal form reached by α and β must be the same.

The second Church-Rosser Theorem:

Given an expression, if it has at least one termination sequence, then the normal order sequence for that expression must also terminate.

2

2.a

fun f g h k x = k (hd (h [g x]))

2.b

(int list →'a)→('b * 'c →bool) →'b →int list * 'c →'a

2.c

First we find all unconstrained parameters, here there is no constraints on the output of foo, thus its denoted as 'a; there is no constraint on x and z, thus they are denoted as 'b and 'c.

Next we can tell that bar is (int list $\rightarrow$int list) as it takes in [1,2,3] as input, and it can use that same value as output. Since the input of f is output of bar, therefore f takes in ”int list” as input, we can also tell that the output of f is the output of foo which is 'a, thus f is (int list $\rightarrow$'a).

(op >) is used on x and z, and the output is used in an if statement, thus the output of (op >) is boolean, thus (op >) is ('b * 'c $\rightarrow$bool).

y must have the same type as [1,2,3], thus y is of type ”int list”. therefore (y z) is of type (int list * 'c).

Combining everything we can infer the type of function foo.

3

3.a

First feature is encapsulation of data with the functions(methods) that operate on the data.
Second feature is inheritance.
Third feature is subtyping with dynamic dispatch.

3.b

For a type T, there is a set that represents all objects of type T, if type K is subtype of T, then the set that represents all objects of type K will be a subset of set T.

3.c

In Java, if I define a class Vehicle, and its subclass Car as follows:

```
class Vehicle {
    int speed;
    void accelerate(int x) {
    speed += x;
    }
}

class Car extends Vehicle {
    string model;
}
```

Since Car is subtype of Vehicle, any object of Car will also have fields of Vehicle (data speed and method accelerate), thus any object of Car is also considered an object of Vehicle. In set representation, this means that the set denoting Car is a subset of the set denoting Vehicle.

Intuitively, if type B is subtype of type A, then object of B must contain all fields of A, thus object of B is also considered object of A, thus the set of B must be a subset of set of A.

3.d

In Scala, consider class A and class B, where B is subtype of A, with additional field called x.

(1)

```
def foo(g:A→Int) = g(new A)
def h(b:B) = b.x
foo(h)
```

Running the three lines above will give you error. h expects an input type of B, running foo(h) will pass in an object A into h, since object of type A is not of type B, the expected input type (B) is different from the actual input type (A) for function h.

We know this will cause issues since h will try to access field x of the parameter, which is specific to object of type B.

This example shows that function subtyping can't be covariant on input type.

(2)

```
def bar(g:Int→B) = g(3)
def j(x:Int) = new A
bar(j)
```

Running the thee lines above will give you error. Function bar expects an output type of B, and j will output an object A, an A is not an B, thus the actual output of bar (A) doesn't match the expected output of bar (B).

This example shows that function subtyping can't be contravariant on output type.

3.e

Function subtyping in Scala shows that function subtyping is contravariant in input type, and covariant in output type.

In set interpretation, the first part means that all functions that takes in A object as input, is a subset of all functions that takes in B object as input. This is intuitive, since B object is also A object, any function that takes in A object as input can obviously take in B object as input. As a result, set of functions with A object as input is subset of functions with B object as input.

The second part means that all functions that output B object, is a subset of all functions that output A object. This is also intuitive, since B is A, functions that output B object can be considered as functions that output A object, therefore set of functions with B object as output, is the subset of functions with A object as output.

4

4.a

```
public static void add_A(ArrayList<A> L) {
    L.add(new A());

ArrayList<B> L = new ArrayList<B>();
L.add_A();
}
```

In the above Java function, given B is subtype of A, if we allow that ArrayList(B) to be subtype of ArrayList(A), then we can run the above Java program. Doing so will give an run-time error, since we will be inserting an A object into an ArrayList<B> object.

4.b

```
public static void add_A(ArrayList<? super A> L) {
    L.add(new A());
```

```
ArrayList<C> L = new ArrayList<C>();
L.add_A();
```

Assuming that B is subtype of A and also A is subtype of C, after the modification, the method add_A can't be called on object of B, but can be called on object of A and C. Note that for calling add_A on ArrayList<C> would not result in errors, since A is C, adding instance of A into ArrayList<C> is a valid operation. In addition, many instances of ArrayList<T> can be called, for all Ts such that A is subtype of T.

5

5.a

```
def f(other:Tree[T]) = {
    other match {
        case Leaf(v) => v
        case Node(v, l, r) =>
            if (v < f(l) && v < f(r)) v
            else if (f(l) < f(r)) f(l)
            else f(r)
    }
}
```

5.b

```
class C[+T]
```

5.c

```
val x:C[A] = new C[B]
```

5.d

```
class D[-T]
```

5.e

```
val y:D[B] = new D[A]
```

6

6.a

Compared to a reference counting collector, a mark-and-sweep collector can better handle cycles of dead objects. When encounter cycles, mark-and-sweep collector checks the mark bit, if it visit an object that has already been visited, it knows cycle has been detected and it won't continue down the cycle. In comparison, reference counting collector doesn't have a good way of detecting cycles, and dead objects in cycles may never be put into the Free List.

6.b

The cost of garbage collection using a copying collector is proportional to the number of live objects in the heap, whereas the cost for a mark-and-sweep is proportional to the entire heap. In cases where there are relatively low number of live objects, the copying collector can run much faster.

6.c

For generational copying garbage collection, many different heaps are used. New objects are always inserted into the first (youngest) heap. Whenever a heap gets filled, A traversal over the entire heap will be performed, and all live objects will be copied to the next youngest heap. After a full traversal is finished and all live objects being copied to the next heap, the current heap can be cleared into new objects.

In generational copying garbage collection, the youngest heap can get filled up most frequently, as new objects always gets inserted into the youngest heap.

6.d

```
def delete(x):
    for all y that structure of x points to:
        y.count -= 1
        if y.count == 0:
            delete(y)
    addToFreeList(x)
    return
```