Programming Languages
CSCI-GA.2110.001 Fall 2019

Homework 2
ANSWERS

1. (a) As we discussed in class, the expression $(\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))$ has no normal form. Write another expression that has no normal form. Make sure that your expression is distinct from $(\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))$, i.e. that it wouldn't be convertible to $(\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))$. Hint: Think about how you'd write a non-terminating expression in a functional language.

   Following the hint, one way to write a non-terminating expression is to use the Y combinator to write a recursive function with no base case. For example,
   $Y\ (\lambda f\ .(\lambda x.\ fx))\ \mathbf{1}$
   Another way would be to write an expression that grows larger every time β-reduction is applied, such as:
   $(\lambda x.\ (x\ x\ x))\ (\lambda x.\ (x\ x\ x))$

   (b) Write the actual expression in the λ-calculus representing the Y combinator, and show that it satisfies the property Y(f) = f(Y(f))

   Answer:
   $Y = (\lambda h.\ ((\lambda x.\ (h\ (x\ x)))\ (\lambda x\ (h\ (x\ x)))))$
   We'll (somewhat informally) show that $Y\ f = f\ (Y\ f)$ using β-conversion ($\Leftrightarrow$) as equivalence.

   $$\begin{aligned} Y\ f &= (\lambda h.\ ((\lambda x.\ (h\ (x\ x)))\ (\lambda x\ (h\ (x\ x)))))\ f \\ &\Leftrightarrow ((\lambda x.\ (f\ (x\ x)))\ (\lambda x\ (f\ (x\ x)))) \\ &\Leftrightarrow f\ ((\lambda x\ (f\ (x\ x)))\ (\lambda x\ (f\ (x\ x)))) \end{aligned}$$

   Since, as shown above,
   $Y\ f \Leftrightarrow ((\lambda x.\ (f\ (x\ x)))\ (\lambda x\ (f\ (x\ x))))$,
   and since $\Leftrightarrow$ is symmetric (i.e. if $a \Leftrightarrow b$ then $b \Leftrightarrow a$),
   $f\ ((\lambda x\ (f\ (x\ x)))\ (\lambda x\ (f\ (x\ x)))) \Leftrightarrow f\ (Y\ f)$.
   Therefore, $Y\ f \Leftrightarrow f(Y\ f)$.

   (c) Define the terms *normal order evaluation* and *applicative order evaluation.*

   ***Normal order evaluation* is the evaluation order in which the <u>leftmost</u>, <u>outermost</u> redex is always chosen to be reduced.**

   ***Applicative order evaluation* is the evaluation order in which the <u>leftmost</u>, <u>innermost</u> redex is always chosen to be reduced.**

   (d) Write an expression containing at least two *redexes*, such that normal order evaluation and applicative order evaluation would choose a different redex to reduce. Justify your answer by showing one step of reduction for each order of evaluation.

   $(\lambda x.\ +\ x\ x)\ ((\lambda y.\ y)\ \mathbf{3})$
   Normal order reduction: $(\lambda x.\ +\ x\ x)\ ((\lambda y.\ y)\ \mathbf{3}) \Rightarrow +\ ((\lambda y.\ y)\ \mathbf{3})\ ((\lambda y.\ y)\ \mathbf{3})$
   Applicative order reduction: $(\lambda x.\ +\ x\ x)\ ((\lambda y.\ y)\ \mathbf{3}) \Rightarrow (\lambda x.\ +\ x\ x)\ \mathbf{3}$

   (e) Summarize, in your own words, what the two Church-Rosser theorems state.

Church-Rosser Theorem I says that for any expression in the λ-calculus, every reduction sequence that terminates will result in the same normal form.
Church-Rosser Theorem II says that if any order of evaluation results in a normal form, then normal order evaluation will also result in a normal form.

2. (a) Write a function in ML that has the following type:
`('a -> 'b) -> ('a -> 'c) -> 'a list -> ('b * 'c) list`.

Answer:

```
fun f g h [] = []
=   f g h (x::xs) = (g x, h x) :: f g h xs
```

(b) What is the type of the following function?

```
fun f (a,b) x [] = []
 |  f (a,b) x (y::ys) = [a x, b y] @ f (a,b) x ys
```

Answer:
`('a -> 'b) * ('c -> 'b) -> 'a -> 'c list -> 'b list`

(c) Explain how the compiler would infer the type in your previous answer.

The type of variable `x` has no constraints within the function, so we'll let the type of `x` be `'a`. Since the variable `a` is applied to `x`, it must be a function whose input type is `'a`. The type of the output of `a` is unconstrained, so we'll let the output type of `a` be `'b`. Thus, the type of `a` is `'a -> 'b`. The type of variable `y` is unconstrained, so we'll let the type of `y` be `'c`. Since `b` is applied to `y`, its input type must be `'c`. The output type of `b` must be the same as the output type of `a`, since the results of calling `a` and calling `b` are in the same list (and lists are homogeneous). Therefore, the type of `b` is `'c->'b`. The type of the parameter represented by the pattern `(y::ys)` must be `'c list`, since the type of `y` is `'c`. Finally, the result type of `f` is the same as the type of `[a x, b y]`, since append (`@`) returns a list of the same type as each of its operands. Thus, the result type is `'b list`.

Putting it all together, the types of `f`'s parameters are:

- `(a,b): ('a -> 'b) * ('c -> 'b)`
- `x: 'a`
- `(y::ys): 'c list`

and the return type is `'b list`, so `f` has the specified type above.

3. (a) Define the term *dynamic dispatch* and give an example in Java.

Dynamic dispatch says that given a method call of the form

```
x.f(...)
```

the method `f` that is called is the method associated with the object that the variable `x` actually refers to, not the method associated with the declared type (class) of `x`.

For example, given the following class definitions,

```
class A {
  void f() { System.out.println("A"); }
}

class B extends A {
  void f() { System.out.println("B"); }
}
```

executing the following two lines,

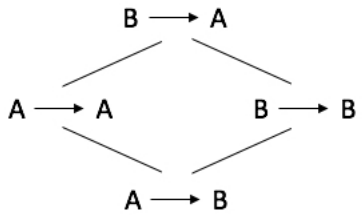
```
A x = new B();
x.f();
```

will print "B", even though `x` is declared to be of type `A`.

(b) Given a class `A` and a subclass `B` of `A`, explain why, under the *subset interpretation of subtyping*, `B` denotes a set that is a subset of the set denoted by `A`.

In the subset interpretation of subtyping, the class `A` denotes the set of all objects that have *at least* the properties (i.e. methods and fields) of `A`. Since all objects of class `B` have all the properties of `A`, and perhaps additional properties, all objects in the set denoted by `B` are also in the set denoted by `A`. Therefore, the set denoted by `B` is a subset of the set denoted by `A`.

(c) For a language (such as Scala) that allows subtyping among function types, draw a diagram showing the subtyping relationships among the types, `A->A`, `A->B`, `B->A`, and `B->B`, assuming `B` is a subtype of `A`.



(d) Suppose that `B` is a subtype of `A`. Write some code in Scala that, if it were allowed by the compiler, would illustrate that having `B->Int` be a subtype of `A->Int` would lead to an unsafe program.

Suppose classes `A` and `B` were defined as follows:

```
class A

class B extends A {
  val x = 6
  }
```

If `B->Int` were a subtype of `A->Int` (which it's not), then the following code would be legal:

```
def f(g:A=>Int) = g(new A())   // f expects an argument of type A->Int

def h(b:B) = b.x   // h is of type B->Int
```

```
f(h)  // if B->Int were a subtype of A->Int, then this would be legal
```

In this case, since calling `g` on an `A` object is really calling `h` on the `A` object, `h` would attempt to access the `x` field of the `A` object, which doesn't exist.

4. In Java generics, subtyping on instances of generic classes is invariant. That is, two different instances `C<A>` and `C<B>` of a generic class `C` have no subtyping relationship, regardless of a subtyping relationship between `A` and `B` (unless, of course, A and B are the same class).

   (a) Write a function (method) in Java that illustrates why, even if `B` is a subtype of `A`, `C<B>` should not be a subtype of `C<A>`. That is, write some Java code that, if the compiler allowed such covariant subtyping among instances of a generic class, would result in a run-time type error.

   Answer:

```
//Assuming B is a subclass of A.

void f(ArrayList<A> L) {
  A a = new A;
  L.add(a);  //if L is actually an ArrayList<B>, this
             //would stick an A object into L, which
             //would create an error if L's elements
             //are treated elsewhere as Bs.
}
```

   An unsafe use of `f()`, above, if it were allowed (which it's not) would be:

```
ArrayList<B> c = new ArrayList<B>();
f(c);   // relies on ArrayList<B> being a subtype of ArrayList<A>, which it's not.
B b = c.get(0);   // actually returns an A, which is now being treated like a B.
```

   (b) Modify the code you wrote for the above question that illustrates how Java allows a form of polymorphism among instances of generic classes, without allowing subtyping. That is, make the function you wrote above be able to be called with many different instances of a generic class.

   Answer:

```
void f(ArrayList<? super A> L) {
  A a = new A;
  L.add(a);   //Adding an A to an ArrayList of elements of a
              //supertype of A (or A itself). That's fine.
}
```

5. (a) What does the term *covariant subtyping* mean in the context of Scala generics? Just define the term, no code needed.

   Covariant subtyping for a generic class `C[]` means that if class `B` is a subtype of class `A`, then `C[B]` is a subtype of `C[A]`.

   (b) What does the term *contravariant subtyping* mean in the context of Scala generics?

   Contravariant subtyping for a generic class `C[]` means that if class `B` is a subtype of class `A`, then `C[A]` is a subtype of `C[B]`.

(c) In ML, a polymorphic list type can be defined using ML's datatype facility:

```
datatype 'a myList = nil | cons of 'a * 'a myList
```

i. Using Scala's case class facility, define a generic type `myList` that is the equivalent of the above `myList` type in ML. It should not use Scala's built-in `List` class.
Answer:

```
abstract class myList[A]
case class Nil[A]() extends myList[A]
case class Cons[A](car:A, cdr: myList[A]) extends myList[A]
```

ii. Write a polymorphic `map()` function in Scala, which takes a function `f` and a `myList` `L`, and returns a `myList` resulting from applying `f` to every element of `L` (that is, exactly what `map()` in ML does). Your `map()` should not be a method of the `MyList` class and it should be just as polymorphic as `map()` in ML.
Answer:

```
def map[T1,T2](f:T1=>T2, ml: myList[T1]): myList[T2] = ml match {
  case Nil() => Nil()
  case Cons(car,cdr) => Cons(f(car), map(f, cdr))
}
```

6. (a) What is the advantage of a reference counting collector over a mark and sweep collector?
Unlike a mark/sweep collector, a reference counting collector does not generally have to stop the program in order for all the dead objects to be reclaimed. Reference counting is *incremental*, meaning storage reclamation is generally performed a little at a time.

(b) What is the advantage of a copying garbage collector over a mark and sweep garbage collector?
Copying GC has two advantages, (1) it compacts the live objects, so that a heap pointer can be used (for faster storag allocation) and (2) its cost is proportional to the number of live objects, not to the total size of the heap (as in mark/sweep GC).

(c) Write a brief description of generational copying garbage collection.
Generational GC uses a series of heaps, each heap representing a "generation" of objects. All new objects are allocated in the "youngest" heap. When the youngest heap fills up, a copying collector copies all live objects to the heap representing the next generation (the second youngest generation). At this point, the youngest heap is empty and the program can continue executing. When the second youngest generation fills up (as a result of copying from the youngest generation during GC), the live objects are copied from the second youngest generation to the next generation (the third youngest), and so forth.

The advantage of generational copying GC is that GC is rarely performed on older generations, which contain objects that have survived multiple GCs and are highly likely to be live. The youngest generation undergoes the most frequent GC, but the youngest objects are generally temporary and are unlikely to be live when GC occurs. Since the cost of copying GC is proportional to the total size of the live objects encountered, the cost of performing GC on the youngest generation will be low.

(d) Write, in the language of your choice, the procedure `delete(x)` in a reference counting GC system, where `x` is a pointer to a structure (e.g. object, struct, etc.) and `delete(x)` reclaims the structure that `x` points to. Assume that there is a free list of available blocks and `addToFreeList(x)` puts the structure that `x` points to onto the free list.

Answer:

```
void delete(obj *x)
{
  x->refCount--;
  if (x->refCount == 0) {
    for(i=0; i < x->num_children; i++)
        delete(x->child[i]);  //assuming each child is a pointer.
    addToFreeList(x);
  }
}
```