Programming Languages - Homework 2

Name: Peng-Yu Chen (pyc305)

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

   (2) $(\lambda x.E)M \underset{\beta}{\Leftrightarrow} E[M/x]$

   In (1), there are no free occurrences of x in $(\lambda x.E)$, so we substitute nothing.
   In (2), we substitute x in E with M.

   (b) $Y = (\lambda h.\ (\lambda x.\ h\ (x\ x))(\lambda x.\ h\ (x\ x)))$

   (c)

$$\begin{aligned} Yf &= (\lambda h.\ (\lambda x.\ h\ (x\ x))\ (\lambda x.\ h\ (x\ x)))\ f && \text{definition of } Y \\ &\underset{\beta}{\Leftrightarrow} (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)) && \beta\text{-reduction of } \lambda h \\ &\underset{\beta}{\Leftrightarrow} f\ ((\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))) && \beta\text{-reduction of } \lambda x \\ &= f\ (Yf) && \text{the second equality} \end{aligned}$$

   (d)

Let `fib` be Yf

f be $(\lambda f.\ \lambda x.$ `if (< x 2) x (+ (f (- x 1)) (f (- x 2)))`$)$

$$\begin{aligned} Yf &= f\ (Yf) \\ \texttt{fib} &= f\ \texttt{fib} \\ \texttt{fib 3} &= f\ \texttt{fib 3} \\ &= (\lambda f.\ \lambda x.\ \texttt{if (< x 2) x (+ (f (- x 1)) (f (- x 2)))})\ \texttt{fib}\ 3 \\ &\underset{\beta}{\Leftrightarrow} (\lambda x.\ \texttt{if (< x 2) x (+ (fib (- x 1)) (fib (- x 2)))})\ 3 \\ &\underset{\beta}{\Leftrightarrow} \texttt{if (< 3 2) 3 (+ (fib (- 3 1)) (fib (- 3 2)))} \\ &\underset{\delta}{\Leftrightarrow} \texttt{+ (fib 2) (fib 1)} \end{aligned}$$

   (e) Church-Rosser Theorem I: if two expressions E_1 and E_2 are interconvertible, then there is an expression E, which can be converted from both E_1 and E_2.
   By having the above property, we can conclude that there is no way a single expression can be converted to two different normal forms.

   Church-Rosser Theorem II: if an expression E_1 can be converted to a normal form expression E_2, then we can use normal order reduction to convert E_1 to E_2.

2. (a) `fun foo f g h a = hd (map h (g [f a]))`

   (b) `(int list -> 'c) -> ('a * 'b -> bool) -> 'a -> int list * 'b -> 'c`

   (c) `x`'s type is unconstraint, so let's assume its type is `'a`.
   `z`'s type is unconstraint, so let's assume its type is `'b`.
   `w`'s type is constraint since `bar` applies to `w` and also applies to `[1, 2, 3]`, `w`'s type is `int list`.
   `y`'s type is constraint by `if x > z then y else w`, `y`'s type should be same as `w`'s type `int list`.
   `bar`'s type is constraint by `bar w = if x > z then y else w`, so its type is `int list -> int list`.
   `(op >)`'s type is constraint by `x > z`, so its type should be `('a * 'b -> bool)`
   `f`'s input type is constraint by `f (bar [1, 2, 3])`, which is `int list`.
   `f`'s output type is unconstraint, so let's assume it's `'c`.
   Combine the parameters' types of `foo` together, we have:

- `f: (int list -> 'c)`
- `(op >): ('a * 'b -> bool)`
- `x: 'a`
- `(y, z): int list * 'b`
- return type is `'c`

3. (a) i. Encapsulation of data with the functions that operate on that data
   ii. Inheritance
   iii. Subtyping with dynamic dispatch

   (b) Assume we have a class `Vehicle` and a class `Car` which inherits from the class `Vehicle`.
   But, a `Car` object may have more fields than a `Vehicle` object.
   We know that the set defined by a type (class) `T` is the set of all objects that have the properties of `T`, and may have additional properties.
   Since all objects of `Car` have the properties of `Vehicle`, and may have additional properties, all objects in `Car` set are also in `Vehicle` set. So, the set of `Car` $\subseteq$ the set of `Vehicle`.

   (c) Assume we have the following two classe `Vehicle` and `Car`:

```
class Vehicle {
  int speed;
}

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

   Since all objects of `Car` have the properties of `Vehicle`, and have an additional property `model`, all objects in `Car` set are also in `Vehicle` set. So, the set of `Car` $\subseteq$ the set of `Vehicle`.

   (d) (1) Suppose we have a class `A` and a class `B` which inherits from the class `A`. We cannot call `f(h)` because `h` gets called on an `A` object. That's not allowed since an `A` object is not a `B` object. In the following example, we access `b.y` which is a field that only appears in `B` and not in `A`.

```
object CovariantDemo extends App {
  class A {
    val x = 1
  }

  class B extends A {
    val y = 2
  }

  def f(g: A => Int) = g(new A)
  def h(b: B) = b.y // access a field that only appears in B and not in A

  f(h)  // not allowed because an A object is not a B object
}
```

   (2) Suppose we have a class `A` and a class `B` which inherits from the class `A`. In the following example, calling `g(3)` is actually calling `h(3)` which returns an `A` object. However, we cannot assign an `A` object to type `B`.

```
object ContravariantDemo extends App {
  class A
  class B extends A

  def f(g: Int => B): Unit = {
    val b: B = g(3) // actually calling h(3) which returns an A object
  }
```

```
    def h(x: Int) = new A

    f(h) // not allowed because we cannot assign an A object to type B
  }
```

(e) Suppose we have a class `A` and a class `B` which inherits from the class `A`.

(1) We know that `A => Int` contains the set of all functions that can be called on an `A` object and return an `Int`. For example, `def f(a: A) = 1` is in `A => Int`.
Also, any functions in `A => Int` can be called on a `B` object and is therefore in `B => Int`.
Therefore, `A => Int` ⊆ `B => Int`.

(2) Consider `def f(g: Int) = new B`, the set of all functions that can be called on an `Int` and return a `B` object.
Since any function returning a `B` is returning an `A`, all functions in `Int => B` are in `Int => A`.
Therefore `Int => B` ⊆ `Int => A`.

4. (a) Suppose the compiler allows covariant subtyping among instances of a generic class. Assume `ArrayList<B>` is a subtype of `ArrayList<A>`:

```
void addA(ArrayList<A> al) {
  al.add(new A());
}

ArrayList<B> bl = new ArrayList<>();
addA(bl);        // assume ArrayList<B> is a subtype of ArrayList<A>
B b = bl.get(0); // error, returns an A, which is not a B
```

(b)
```
void addA(ArrayList<? super A> al) {
  al.add(new A()); // it's okay to add an A to an ArrayList of elements of
                   // A's supertype (including A)
}
```

5. (a)
```
abstract class Tree[T <: Ordered[T]]
case class Node[T <: Ordered[T]](v: T, l: Tree[T], r: Tree[T]) extends Tree[T]
case class Leaf[T <: Ordered[T]](v: T) extends Tree[T]

class OrderedInt(x: Int) extends Ordered[OrderedInt] {
  def get = x
  def compare(that: OrderedInt) = x - that.get
  override def toString(): String = (s"$x")
}

object Prog {
  def min[T <: Ordered[T]](tree: Tree[T]): T = tree match {
    case Node(value, left, right) =>
      val minL = min(left)
      val minR = min(right)
      val minChild = if (minL < minR) minL else minR
      return if (value < minChild) value else minChild
    case Leaf(value) => value
  }

  def main(args: Array[String]) = {
    val x: Tree[OrderedInt] = Node(
      new OrderedInt(-1),
      Node(new OrderedInt(0), Leaf(new OrderedInt(3)), Leaf(new OrderedInt(5))),
      Node(new OrderedInt(8), Leaf(new OrderedInt(7)), Leaf(new OrderedInt(1)))
    )
    print(min(x)) // should print -1
  }
}
```

(b)
```
class A(x: Int) {
  override def toString() = "A<" + x + ">"
}

class B(x: Int, y: Int) extends A(x) {
  override def toString() = "B<" + x + "," + y + ">"
}

class C[+E](l: List[E]) {
  override def toString() = (s"$l")
}
```

(c)
```
object CovariantDemo {
  def printC(l: C[A]) = {
    println(l)
  }

  def main(args: Array[String]) = {
    val listA = new C(List(new A(1)))
    val listB = new C(List(new B(1, 2)))

    printC(listA)
    printC(listB) // uses covariant subtyping (C[B] <: C[A])
  }
}
```

(d)
```
class A(x: Int) {
  override def toString() = "A<" + x + ">"
}

class B(x: Int, y: Int) extends A(x) {
  override def toString() = "B<" + x + "," + y + ">"
}

class C[-E]() {
  var l: List[Any] = List()
  def insert(x: E) = l = x :: l

  override def toString() = (s"$l")
}
```

(e)
```
object ContravariantDemo {
  def insertC(l: C[B], b: B) =
    l.insert(b)

  def main(args: Array[String]) = {
    val listB = new C[B]()
    val listA = new C[A]()

    insertC(listB, new B(1, 2))
    insertC(listA, new B(3, 4)) // uses contravariant subtyping (C[A] <: C[B])
    listA.insert(new A(5))

    println(listB)
    println(listA)
  }
}
```

6. (a) Mark-and-sweep garbage collector (GC) won't be affected if there's a cycle, while reference counting collector might need a backup mark-and-sweep collector when the heap fills up with cycles.

   (b)
   - Copying GC compacts the live objects, so we could use a heap pointer for faster storage allocation.
   - Unlike mark-and-sweep GC, the cost of copying GC is proportional to the amount of the live objects, not the size of the heap.

   (c) Generational copying GC uses many heaps instead of two heaps. Each heap contains objects of similar age. These heaps are called generations, for obvious reasons. When a heap fills up, the live objects in that heap is copied to the next (older) heap.

   When a heap fills up, the generational copying GC copies the live objects in only that heap to the next older heap, so no other heap is touched during GC.

   The advantages of generational copying GC over the simple copying GC are:

   - Since the youngest heap doesn't contain all the older objects, rather only the youngest objects, GC will be triggered less frequently.
   - Since young objects tend to die at a higher rate than older objects, and only live objects are copied to the older heap, GC will take less time on the youngest heap.
   - GC will rarely happen on the older heaps which contain most of the live objects.

   (d)
```
void delete(Object* x) {
  x->refcount = x->refcount - 1;
  if (x->refcount == 0) {
    for (Object* child : x->children) {
      delete(child);
    }
    addToFreeList(x);
  }
}
```