Programming Languages
CSCI-GA.2110.001 Spring 2017

Homework 2
Due Monday, May 8

ANSWERS

1. (a) In the λ-calculus, give an example of an expression which would reduce to normal form under normal-order evaluation, but not under applicative-order evaluation.

Answer:

$(\lambda\ y.\ 3)\ ((\lambda\ x.\ (x\ x))\ (\lambda\ x.\ (x\ x)))$

(b) Write the definition of a recursive function (other than factorial) using the Y combinator. Show a series of reductions of an expression involving that function which illustrates how it is, in fact, recursive (as I did in class for factorial).

Answer:

The Fibonacci function can be defined using the Y combinator as:

$Y\ (\lambda f.\ \lambda x.$ **if** $(= x$ **0) 1 (if** $(= x$ **1) 1 (+** $(f$ **(-** x **1))** $(f$ **(-** x **2)))))**

For convenience, let FIB be above expression. Since, for all f, Y f = f (Y f),

$$\begin{aligned}\textbf{FIB} &= \textbf{Y}\ (\lambda f.\ \lambda x.\ \ldots\ ) \\ &= (\lambda f.\ \lambda x.\ \ldots\ )\ (\textbf{Y}\ (\lambda f.\ \lambda x.\ \ldots\ )) \\ &= (\lambda f.\ \lambda x.\ \ldots\ )\ \textbf{FIB}\end{aligned}$$

To see that FIB is, in fact, the Fibonacci function, consider applying FIB to 4 and performing β-reduction and δ-reduction:

$$\begin{aligned}\textbf{FIB 4} &= (\lambda f.\ \lambda x.\ \ldots\ )\ \textbf{FIB 4} \\ &\to (\lambda x.\ \textbf{if}\ (= x\ \textbf{0})\ \textbf{1}\ (\textbf{if}\ (= x\ \textbf{1})\ \textbf{1}\ (+\ (\textbf{FIB}\ (\text{-}\ x\ \textbf{1}))\ (\textbf{FIB}\ (\text{-}\ x\ \textbf{2})))))\ \textbf{4} \\ &\to \textbf{if}\ (= \textbf{4 0})\ \textbf{1}\ (\textbf{if}\ (= \textbf{4 1})\ \textbf{1}\ (+\ (\textbf{FIB}\ (\text{-}\ \textbf{4 1}))\ (\textbf{FIB}\ (\text{-}\ \textbf{4 2})))) \\ &\to \textbf{if false 1}\ (\textbf{if}\ (= \textbf{4 1})\ \textbf{1}\ (+\ (\textbf{FIB}\ (\text{-}\ \textbf{4 1}))\ (\textbf{FIB}\ (\text{-}\ \textbf{4 2})))) \\ &\to \ldots \\ &\to +\ (\textbf{FIB 3})\ (\textbf{FIB 2}) \\ &\to \ldots\end{aligned}$$

(c) 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, $Y\ f \Leftrightarrow f(Y\ f)$.

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

Answer:

The first Church-Rosser theorem (actually, a corollary to it) states that an expression cannot be reduced to two distinct normal forms.

The second Church-Rosser theorem states that if there exists a reduction sequence from an expression to a normal form, then normal order reduction will also reduce the expression to normal form. That is, normal order reduction is the reduction order most likely to terminate.

2. (a) In ML, why do all lists have to be homogeneous (i.e. all elements of a list must be of the same type)? **Answer:**

In order to perform static type checking, the compiler has to be able to determine the type of each value at compile time (even if the type of the value contains a type variable). If lists were allowed to contain elements of different types, then for code that selects an element from a list, the compiler would not be able to determine what the type of that element is.

(b) Write a function in ML whose type is `('a -> 'b list) -> ('b -> 'c list) -> 'a -> 'c`.

Answer:

```
fun f g h x = let val (y::ys) = g x
                  val (z::zs) = h y
              in z
              end
```

(c) What is the type of the following function (try to answer without running the ML system)?

```
fun foo (op >) x (y,z) =
   let fun bar a = if x > y then z else a
   in  bar [1,2,3]
   end
```

Answer:

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

(d) Provide an intuitive explanation of how the ML type inferencer would infer the type that you gave as the answer to the previous question.

Answer:

The only constraint on an infix parameter, such as `(op >)`, is that it take a tuple of two elements and return a value. In this case, though, the output of `(op >)`, when it is used in `if x > y then ...`, must be a boolean. So, the type of `(op >)` can be inferred to be `'a * 'b -> bool`.

Since `x` and `y` are used as the operands of `(op >)`, their types must be `'a` and `'b`, respectively. Since `z` and `a` are used as the two branches of the conditional within `bar`, `z` and `a` must have the same type – no matter what the type of `a` is.

Since `bar` is called on an `int list` (i.e. `a` is an `int list` in that call to `bar`), and `z` must have the same type in all calls to `bar`, `z` must be an `int list`. The result type of `foo` is the result type of `bar [1,2,3]`, which is `int list`.

Since the parameters to `foo` are `(op >)`, `x`, and `(y,z)`, and the return type is `int list`, the type of foo is `('a * 'b -> bool) -> 'a -> 'b * int list -> int list`.

3. (a) As discussed in class, what are the three features that a language must have in order to considered object oriented?

   Answer:

   - **Encapsulation of data and code (methods).**
   - **Inheritance.**
   - **Subtyping with dynamic dispatch.**

   (b) What is the "subset interpretation of suptyping"?

   Answer:

   It's the interpretation of subtyping such that the set of values defined by a subtype of a parent type is a subset of the set of values defined by the parent type. That is, if type B is a subtype of type A, then any value in the set of values defined by type B is also in the set of values defined by type A.

   (c) Explain why function subtyping must be contravariant in the parameter type and covariant in the result type. If necessary, provide examples to illuminate your explanation.

   Answer:

   Here is a Scala example showing that covariant subtyping on the input (parameter) type of a function would lead to a type error.

```
class A { val x = 2 }
class B extends A { val y = 3 }    // subtype of A, with a y
                                   // field that A doesn't have

var f:  A => Int;    //variable expecting to be
                     //assigned a value of type A => Int

def g(b: B): Int  = b.y ;  //function of type B => Int
```

   If Scala allowed `B => Int` to be a subtype of `A => Int`, then it would have to allow the assignment

```
f = g
```

   so that whenever `f` gets called, `g` would actually be executed. In that case,

```
f(new A())
```

   would be legal, since `f` is of type `A=>Int`, but since `g` is actually being called, `g` would try to access the `y` field of the `A` object – which doesn't exist. Clearly, this would be an error.

   To show that contravariant subtyping on the input type of a function is safe, here is an example (given the same definitions of `A` and `B`):

```
var h:  B => Int;    //variable expecting to be assigned a value of type B => Int
def k(a: A): Int  = a.y ;  //function of type A => Int
```

Allowing `A => Int` to be a subtype of `B => Int`, the assignment

```
h = k
```

is fine, so that whenever `h` gets called, `k` would actually be executed. In that case,

```
h(new B())
```

would cause the `B` object to be passed to `k`, which is expecting an `A` object. This, of course, is fine, since a `B` object is an `A` object. This shows that contravariant subtyping on the input type is fine.
The following example shows that covariant subtyping on the output type of a function is fine, but contravariant subtyping is not (again, given the definitions of class `A` and class `B`, above).

```
var p: Int=>A  //variable of type Int=>A
var q: Int=>B  //variable of type Int=>B
def r(x:Int) = new A() //function of type Int=>A
def s(x:Int) = new B() //function of type Int=>B

p = s //writing an Int=>B to a variable of type Int=>A (covariance)
val a:A = p(3);   //p is really s, so it returns a B. This is fine,
                  //since writing a B object to an A variable is OK.

q = r //writing an Int=>A to a variable of type Int=>B (contravariance)
val b:B = q(3) //q is really r, so it returns an A. This is a type error,
               //since you can't write an A object to a B variable.
               //This shows contravariance on the output type must be
               //prohibited.
```

(d) Provide an intuitive answer showing why function subtyping satisfies the subset interpretation of subtyping. Be sure to consider both the contravariant and covariant aspects of function subtyping.

Assume that type `B` is a subtype of type `A`.

- **Contravariance on the parameter type: The type `A->T` denotes the set of all functions that can be applied to an `A` object and return a `T` (for any given type `T`). The type `B->T` denotes the set of all functions that can be applied to a `B` object and return a `T`. Any function that can be applied to an `A` object, and thus is in `A->T`, can also be applied to a `B` object, thus is in `B->T`. Therefore, since every element of `A->T` is also an element of `B->T`, `A->T` is a subset of `B->T`.**
- **Covariance on the result type: Any function in `T->B` returns a `B` object. Since a `B` object is also an `A` object (because `B` is a subtype of `A`), a function returning a `B` object also returns an `A` object. Thus, any element of `T->B` is also in `T->A`. Therefore, `T->B` is a subset of `T->A`.**

(e) Give an example in Scala that demonstrates subtyping of functions, utilizing both the contravariance on the parameter type and covariance on the result type.

Answer:

An example showing the subtyping of functions in Scala is the following.

```
class A(z: Integer) {
  val x = z
}

class B(z: Integer, w: Integer) extends A(z) {
  val y = w
}

object hw2 {

  def fab(a: A): B = new B(a.x, 3)  //of type A->B
  def fba(b: B): A = new A(b.x+b.y)  //of type B->A

  var ba: B=>A = fab   //fine, since A->B is a subtype of B->A
  var ab: A=>B = fba   //error, since B->A is not a subtype of A->B

  def main(args: Array[String]) {
  }
}
```

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.
}
```

   (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) In Scala, write a generic class definition that supports covariant subtyping among instances of the class. For example, define a generic class C[E] such that if class B is a subtype of class A, then C[B] is a subtype of C[A].

   In the definition of the generic class C[E], you have to be careful that type E is only used in covariant position (as field types or the the output types of methods, not as variables to be assigned to or as values to be inserted).

```
class C[+E](x:E) {
  val y:E = x
  def value():E = y
}
```

   (b) Give an example of the use of your generic class.

   Answer:

```
class A {}

class B extends A {}

object foo2 {

  def test(m: C[A]):A = m.value()

  def main(args: Array[String]) {
    val L1 = new C[A](new A)
    val a1: A = test(L1)
    println(L1)

    val L2 = new C[B](new B)
    val a2: A = test(L2)   //use of coviarant subtyping
    println(L2)
  }
}
```

   (c) In Scala, write a generic class definition that supports contravariant subtyping among instances of the class. For example, define a generic class C[E] such that if class B is a subtype of class A, then C[A] is a subtype of C[B].

   In the definition of the generic class C[E], you have to be careful that type E is only used in contravariant position (as the input types of methods and not as the output types nor as the types of fields).

```
class C[-T]() {

  var v: List[Any] = List()
  def insert(x: T) {
    v = x::v
  }
}
```

(d) Give an example of the use of your generic class.

```
class A {}

class B extends A {}

object foo {

  def test(m: C[B], z: B) {
    m.insert(z)
  }

  def main(args: Array[String]) {
    val L1 = new C[B]()
    test(L1, new B)

    val L2 = new C[A]()
    test(L2, new B)   //use of contravariant subtyping in first argument
  }
}
```

(e) Consider the following Scala definition of a tree type, where each node contains a value.

```
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]
```

Ordered is a built-in trait in Scala (see `http://www.scala-lang.org/api/current/index.html#scala.math.Ordered`). Write a Scala function that takes a Tree[T], for any ordered T, and returns the smallest (minimum) value in the tree. Be sure to use good Scala programming style.

Answer:

```
  //helper function
  def max[T <: Ordered[T]](a: T, b: T, c: T) = {
    if (a < b)
      if (a < c) a else c
   else
    if (b < c) b else c
  }

  def maxTree[T <: Ordered[T]](t: Tree[T]): T = t match {
    case Leaf(value) => value
    case Node(value, left, right) => max(value, maxTree(left), maxTree(right))
  }
```

6. (a) What is the advantage of a reference counting collector over a mark and sweep collector?

   Answer:

   A reference counting collector doesn't force the program to stop for a significant amount of time in order to perform GC of the entire heap space, as a mark/sweep collector does. The reference counting collector reclaims objects as soon as they are no longer accessible, rather than waiting until the heap is full. Thus, reference counting is more suitable for (soft) real-time programs than mark/sweep GC.

   (b) What is the advantage of a copying garbage collector over a mark and sweep garbage collector?

   Copying GC performs compaction of the live objects, so that a simple heap pointer can be used for storage allocation (no free list needed). The cost of copying GC is also proportional to the total size of live objects, whereas the cost of a mark-and-sweep GC is proportional to total size of the heap.

   (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)` deletes the pointer x. Assume that there is a free list of available blocks and `addToFreeList(x)` puts the structure that `x` points to onto the free list.

```
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);
  }
}
```