Programming Languages
CSCI-GA.2110.001 Fall 2021

Homework 2

ANSWERS

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

   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) -> ('c -> 'd) -> 'a -> 'd list
   ```

   Answer:

   ```
   fun foo f g h x = let val (b::bs) = f x
                         val (c::cs) = g b
                     in [h c]
                     end
   ```

   or

   ```
   fun foo f g h x = [h (hd (g (hd (f x))))]
   ```

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

   ```
   fun foo (op <) f g (x,y) z  =  if f(x,y) < g x then z + 1 else z - 1
   ```

   Answer:

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

   Note: Your type variables (`'a`, `'b`, etc.) can be different, of course, as long as there is a one-to-one correspondence between your type variables and the type variables in the above type expression.

   (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.

   This is what the compiler uses to infer the type of `foo`:

   - **There is no restriction on the types of `x` and `y`, so the type of `x` can be `'c` and the type of `y` can be `'d`.**
   - **Since `(x,y)` is passed to `f`, where `x` is of type `'c` and `y` is of type `'d`, above, the input type of `f` is `'c * 'd`. There is no restriction on the output type of `f`, so the output type of `f` can be `'a`. Thus, the type of `f` is `('c * 'd) -> 'a`.**

- **Since `x` is passed to `g`, the input type of `g` is `'c` (see above). There is no restriction on the output type of `g`, so the output type of `g` can be `b`. Thus, the type of `g` is `'c -> 'b`.**
- **Since the result of `<` in the body of `foo` is used as the condition in an `if` expression, the result type of `<` is `bool`. Since the inputs to `<` are the outputs of `f` and `g`, and the output types of `f` and `g` are `'a` and `'b`, respectively, the input type of `<` is `'a * 'b`. Thus, the type of `<` is `('a * 'b -> bool)`.**
- **Since the expressions `z + 1` and `z - 1` appear in the body of `foo`, `z` must be of type `int`.**
- **Since either the result of `foo` is either `z + 1` or `z - 1`, the return type of `foo` must be `int`.**

Putting the above information together, since the parameters to `foo` are `<`, `f`, `g`, the tuple `(x,y)`, and `z`, and the return type is int, the type of `foo` is:

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

2. (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, which I did in class) 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** $(=\ x\ \mathbf{0})\ \mathbf{1}\ (\mathbf{if}\ (=\ x\ \mathbf{1})\ \mathbf{1}\ (+\ (f\ (\mathbf{-}\ x\ \mathbf{1}))\ (f\ (\mathbf{-}\ x\ \mathbf{2})))))$

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

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

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

$$\begin{aligned}\mathbf{FIB\ 4} &\Leftrightarrow (\lambda f.\ \lambda x.\ \mathbf{...}\ )\ \mathbf{FIB\ 4}\\ &\Rightarrow (\lambda x.\ \mathbf{if}\ (=\ x\ \mathbf{0})\ \mathbf{1}\ (\mathbf{if}\ (=\ x\ \mathbf{1})\ \mathbf{1}\ (+\ (\mathbf{FIB}\ (\mathbf{-}\ x\ \mathbf{1}))\ (\mathbf{FIB}\ (\mathbf{-}\ x\ \mathbf{2})))))\ \mathbf{4}\\ &\Rightarrow \mathbf{if}\ (=\ \mathbf{4}\ \mathbf{0})\ \mathbf{1}\ (\mathbf{if}\ (=\ \mathbf{4}\ \mathbf{1})\ \mathbf{1}\ (+\ (\mathbf{FIB}\ (\mathbf{-}\ \mathbf{4}\ \mathbf{1}))\ (\mathbf{FIB}\ (\mathbf{-}\ \mathbf{4}\ \mathbf{2}))))\\ &\Rightarrow \mathbf{if\ false\ 1}\ (\mathbf{if}\ (=\ \mathbf{4}\ \mathbf{1})\ \mathbf{1}\ (+\ (\mathbf{FIB}\ (\mathbf{-}\ \mathbf{4}\ \mathbf{1}))\ (\mathbf{FIB}\ (\mathbf{-}\ \mathbf{4}\ \mathbf{2}))))\\ &\Rightarrow \mathbf{...}\\ &\Rightarrow +\ \mathbf{(FIB\ 3)\ (FIB\ 2)}\\ &\Rightarrow \mathbf{...}\end{aligned}$$

(c) Write the actual expression in the λ-calculus representing the Y combinator, and show that it satisfies the property Y(f) $\Leftrightarrow$ f(Y(f)).

Answer:

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

We'll show that $Y\ f \Leftrightarrow f\ (Y\ f)$ using β-conversion ($\Leftrightarrow$) as equivalence.

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

Note that starting with the underlined expression, the next steps are simply performing β-conversion in the opposite direction from the first two steps (i.e. conversion in the $\Leftarrow$ direction).

(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. That is, given an expression, all reduction sequences for that expression that terminate will result in the same normal form.

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.

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

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

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

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) Provide an intuitive answer, and give an example, showing why class derivation in Java satisfies the subset interpretation of subtyping.

A class with certain properties (fields and methods) can be thought of as defining the set of all values that have <u>at least</u> those properties. Since any subclass derived from that orginal class also has at least those properties, the values in the set defined by the subclass will also be in the set defined by the original class. For example, given

```
class A {
  int x;
  int value() { return x+1; }
}
```

class A can be thought of as defining the set of all objects with at least an integer `x` field and a method `value()` that returns an integer. The definition

```
class B extends A {
  int y;
  int value() { return x+y+1; }
}
```

defines a subclass B of A, which can be thought of as defining the set of all objects with at least an integer `x` field, a method `value()` that returns an integer, and an integer y field. This set (defined by class B) is clearly a subset of the set defined by class A.

(d) Provide an intuitive answer, and give an example (in code), showing why subtyping of functions, in languages (such as Scala) that allow it, satisfies the subset interpretation of subtyping.

As discussed in class, function subtyping is contravariant in the input type. The type $T \to Int$ can be thought of as the set of all functions that can be applied to an object of type T and return an integer.

Suppose B is a subtype of A. Any function that can be applied to an A can also be applied to a B, because a B is an A. This means that any function that can be applied to an A and return an Int, i.e. any function that is in the set denoted by $A \to Int$, can also be applied to a B and return an int, and thus is in the set denoted by $B \to Int$. Therefore the set denoted by $A \to Int$ is a subset of the set denoted by $B \to Int$. Note that not every function that can be applied to a B can also be applied to an A, thus the set denoted by $A \to Int$ is a *proper* subset of the set denoted by $B \to Int$.

Also as discussed in class, function subtyping is covariant in the output type. The type $Int \to T$ can be thought of as the set of all functions that can be applied to an Int and return a T object.

Suppose, again, that B is a subtype of A. Since every function that returns a B object also returns an A object (because a B is an A), any function in the set $Int \to B$ is also in the set $Int \to A$. Thus, the set denoted by $Int \to B$ is a subset of the set denoted by $Int \to A$. Note that not every function that returns an A also returns a B, since an A is not a B, thus the set denoted by $Int \to B$ is a *proper* subset of the set denoted by $Int \to A$.

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) 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 `minTree` that takes a Tree[T], for any ordered T, and returns the minimum value in the tree. Be sure to use good Scala programming style.

   Answer:

```
//helper function
def min[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 minTree[T <: Ordered[T]](t: Tree[T]): T = t match {
  case Leaf(value) => value
  case Node(value, left, right) => min(value, minTree(left), minTree(right))
}
```

(b) i. 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 (e.g. as the types of fields of the class or the the output types of methods)

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

ii. 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) i. 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
  }
}
```

ii. Give an example of the use of your generic class.

Answer:

```
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
  }
}
```

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

Mark-and-sweep GC is able to collect cyclical structures that are no longer reachable. Reference counting will not collect cycles, even if they are not accessible by the program.

(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)` 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);
  }
}
```