Programming Languages
CSCI-GA.2110.001 Fall 2021

Homework 2
Due Sunday, December 19 at 11:55pm

You should write the answers using word, latex, etc., and upload them as a PDF document. **Important: You must turn this in by 11:55pm on December 19. I will be posting the solutions shortly after.**

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)?
Answer:
To perform static type checking, the compiler has to confirm the type of the value at compile time. If the elements are different types, then when compiling the code, we cannot confirm the type of the value that we use.

   (b) Write a function in ML whose type is:

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

Answer:

```
fun test f g h x =
   let val (y::ys) = f x
       val (j::js) = g y
       val k = h j
   in
   [k]
   end
```

   (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
```

   (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:

```
1. Find each variable that has no constraints on it, then Assign the type
of that variable to be a type variable('a,'b,etc):
The parameter we have is (op <),f,g,(x,y),and z.
1) There are two parameters of infix. According to the if statement, we
could see the two type of infix parameter is the return value of f and g,
so we assign 'a and 'b.
2) The variable of x,y also can be any type, so we assign 'c to x, 'd to y.
And the result type of infix is bool.
2. Determine the types of the other variable, based on how those variable
```

```
are used and on the types from step.
The result type is (z+1) or (z-1), then we know the type of z is int.
3. Computing the result type of the function:
('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: example: $(\lambda y.3)((\lambda y.x\ x)(\lambda y.x\ x))$
normal order evaluation:
$(\lambda y.3)((\lambda y.x\ x)(\lambda y.x\ x)) = 3$;
application order evaluation:
$(\lambda y.3)((\lambda y.x\ x)(\lambda y.x\ x)) = (\lambda y.3)((\lambda y.x\ x)(\lambda y.x\ x))$;[infinite reduction sequence]

   (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:
Function Fibonacci(F): y(λf.λn. if(= n 0) 1 if (= n 1) 1 (+ (f (- n 1)) (f (- n 2))))
According Y(f) $\Leftrightarrow$ f(Y(f)), we could get:
F 3 = y (λf.λn. if(= n 0) 1 if (= n 1) 1 (+ (f (- n 1)) (f (- n 2)))) 3
= (λf.λn. if(= n 0) 1 if (= n 1) 1 (+ (f (- n 1)) (f (- n 2)))) F 3
$=>_\beta$ if(= 3 0) 1 if (= 3 1) 1 (+ (F (- 3 1)) (F (- 3 2)))
$=>_\beta$ (+ (F 2) (F 1))
$=>_\beta$ (+ (λ.....))
So, we can see that Y combinator could get the correct result of recursive.

   (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:
Assuming Y is (λh.(λx. h(x x)) (λx. h(x x)).
yf = (λh.(λx. h(x x)) (λx. h(x x))) f
$=>_\beta$ (λx. f(x x)) (λx. f(x x))
$=>_\beta$ f ((λx. f(x x)) (λx. f(x x)))
$=>_\beta$ f ((λh.(λx. h(x x)) (λx. h(x x)))f)
= f(yf)
Therefore, this expression represents Y combinator in the λ-Calculus.

   (d) Summarize, in your own words, what the two Church-Rosser theorems state.
Answer:
1. if both the two reduction applying to the expression terminate, the normal forms of the same expression should be the same.
2. if there is at least one termination sequence, then the normal order sequence of the expression must also terminate.

3. (a) As discussed in class, what are the three features that a language must have in order to considered object oriented?
Answer:
1. inheritance
2. encapsulation of data and procedure(code) into a single structure
3. subtyping with dynamic dispatch

(b) What is the "subset interpretation of suptyping"?
Answer:
The set of values defined by a subtype B is the subset of the set of values defined by B's parent type A.

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

```
class Vehicle{int speed; void acclerate();}
class Car extends Vehicle{int num;}
```

We can get the information that Car is the subtype of Vehicle. According the definition in Java, the object Car has all the fields that in Vehicle, thus Object Car is an Object Vehicle. That means the set denoting Car is the subset of the set of denoting Vehicle. So generally, if B is the subtype of A, B is A, b contains all fields and methods that in A. The set of B must be the subset of A. So Java class derivation satisfies the subset interpretation of subtyping.

(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.
Answer:

```
class A{ int x}
class B extends A{int y}}
def f(g: B => int) {...}
def ff(g: A => int) {...}
def h(x: A) { return 6; }
def hh(x: B) { return 5; }
f(h); //Fine
ff(hh); //Error.

def fA(g : int => A) { A a = g(3); }
def fB(g : int => B) {B b = g(2);}
def h(x : int) { return new A(); }
def j(y : int) { return new B(); }
fA(j); // Fine.
fB(h); // Error.
```

From above, we get B is subtype of A. Subtyping in Scala show that subtyping in input type is contravariant, in output type is covariant.

For the first part, it means that the A => int is subtype of B=>int. So methods with A as input is subset of methods with B as input. For the second part, it means that int => B is the subtype of int => A. So methods with B as output is subset of methods with A as output. So Scala satisfies the subset interpretation of subtyping.

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:

```
class A{}
class B extends A{}
class C extends B{}
void addToList(ArrayList<B> L) {
    L.add(new B());
}
```

We know B is subtype of A,C is subtype of B. We object C call the function, adding B to C list. But in Java, it is not allowed. Bacause B instance is not a C instance. So it will cause a runtime error.

(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 addToList(ArrayList<? super B> L) {
    L.add(new B());
}
```

After the modification, object C can not call the function, adding B to C list, because B is not C. But B and A can call the function. Adding B to A list will not cause run-time error, since B is A.

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.

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

(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].
Answer: C[+T]

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

```
class A{}
class B extends A{}
val x: C[A] = new C[B]
```

(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].
Answer: C[-T]

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

```
class A{}
class B extends A{}
val x: C[B] = new C[A]
```

6. (a) What is the advantage of a mark-and-sweep garbage collector over a reference counting collector?
Answer:
Mark-and-sweep can handle cycle structure that cannot be reached. But reference counting cannot collect cycles that are not be reached any more.

(b) What is the advantage of a copying garbage collector over a mark and sweep garbage collector?
Answer:
1. Copying garbage collector use simple heap structure(not free list) because it compact the live object.
2. The cost of copying garbage collector is proportional to the total size of live objects, but the cost of mark and sweep garbage collector is proportional to the total size of the heap.

(c) Write a brief description of generational copying garbage collection.
Answer:
Generational copying garbage collection use multiple heaps. New objects are always inserted to the youngest heap. When the youngest heap fills up, all live objects are removed from the youngest heap to the second youngest heap. Then the youngest heap is empty. Then the program continues. When the second youngest heap fills up, all live objects are removed from it to the next youngest heap, continues this similar process and so forth.

(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(object x) {
```

```
    x.refCnt--;
    if(x.refCnt == 0) {
        for(int i = 0; i < x.childNum; i++)
            delete(x.child[i]);
    }
    addToFreeList(x);
}
```