Programming Languages
CSCI-GA.2110.001 Spring 2015

Homework 2
Due Sunday, May 10

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 Sunday, May 10. I will be posting the solutions after that.

1. As you know, the sequence of Fibonacci numbers can be defined as follows:
   fib (0) = 0, fib(1) = 1, fib(j) = fib(j-1) + fib(j-2).

   (a) Write in Scheme the function (fib n) which returns the nth Fibonacci number (i.e. fib(n) in the above definition). This Scheme code should reflect the above definition of Fibonacci numbers.

   Ans:
```
( define ( fib n )
    ( cond ( ( = n 0 ) 0 )
    ( ( = n 1 ) 1 )
    ( else ( + ( fib ( - n 1 ) ) ( fib ( - n 2 ) ) ) ) ) ) )
```

   (b) If you wrote the Scheme code to directly reflect the definition of the Fibonacci sequence (which you were supposed to), the complexity of the program would be exponential (i.e. O(2n)), due to two recursive calls in the body of the function. For this part, write a Scheme function, (lfib n), that computes the nth Fibonacci number in linear time (with only a single recursive call). Do not define any function outside of lfib, but you can use letrec within lfib.

   Ans:
```
( define ( lfib n )
        ( letrec ( ( lfib1 (lambda (x y i)
                ( cond ( (= i 0) x)
                        ( else ( lfib1 y (+ x y) (- i 1) ) ) ) ) ) )
            ( lfib1 0 1 n ) ) )
```

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.

   Ans: $(\lambda x.3)((\lambda x.xx)(\lambda x.xx))$

   (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).

   Ans: recursive function for nth Fibonacci number using Y combinator
   Y (λf . λx . if (= x 0) 0 if (= x 1) 1 (+ f (- x 1) f (- x 2)))

   Illustration: (fib 4)

Y f = f (Y f)

Y (λf . λx . if (= x 0) 0 if (= x 1) 1 (+ f (- x 1) f (- x 2))) 4

⇨ (λf . λx . if (= x 0) 0 if (= x 1) 1 (+ f (- x 1) f (- x 2)))
(Y (λf . λx . if (= x 0) 0 if (= x 1) 1 (+ f (- x 1) f (- x 2)))) 4

⇨ (λx . if (= x 0) 0 if (= x 1) 1 (+ (Y (λf . λx . if (= x 0) 0 if (= x 1) 1 (
+ f (- x 1) f (- x 2)))) (- x 1) (Y (λf . λx . if (= x 0) 0 if (= x 1) 1 (+
f (- x 1) f (- x 2))) (- x 2))))) 4

⇨ if (= 4 0) 0 if (= 4 1) 1 (+ (Y (λf . λx . if (= x 0) 0 if (= x 1) 1 (+ f (-
x 1) f (- x 2)))) (- 4 1) (Y (λf . λx . if (= x 0) 0 if (= x 1) 1 (+ f (- x
1) f (- x 2))) (- 4 2))))

⇨ (+ (Y (λf . λx . if (= x 0) 0 if (= x 1) 1 (+ f (- x 1) f (- x 2)))) 3
(Y (λf . λx . if (= x 0) 0 if (= x 1) 1 (+ f (- x 1) f (- x 2)))) 2)

(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)).

Ans: Yf = (λh. (λx . h (x x)) (λx . h (x x)))f
⇨ (λx . f (x x)) (λx . f (x x))
⇨ f ((λx . f (x x)) (λx . f (x x)))
⇨ f (Y f)

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

Church – Rosser theorem I: An expression can be reduced to only one normal form irrespective of the ordering in which the reductions are performed.

Church – Rosser theorem II: If an expression can be reduced to normal form then there exists an normal order reduction for the expression i.e. no such expression exists which can be reduced to normal form using applicative order of evaluation but cannot be reduced using normal order of evaluation.

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

Ans: ML is statically typed language and uses type inference. So if a heterogeneous list is provided then it cannot infer the proper type of the list and hence cannot determine the list of operations valid for the type.

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

Ans: fun p f g x = let val (b::bs) = (f x); val (c::cs) = (g b) in c end;

val p = fn : ('a -> 'b list) -> ('b -> 'c list) -> 'a -> 'c

(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
```

Ans: val foo = fn: (‘a *‘b -> bool) -> ‘a -> ‘b * int list -> int list

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

Ans: As the output is ‘a’ which is the input for function ‘bar’ and the given input is an ‘int list’ so the result of the function is an ‘int list’.
As ‘Z’ is also a return value for function ‘bar’ so it should have the same type as ‘a’ hence it will also be ‘int list’.
As (op >) is a function being used like an infix operator it must take a tuple any type and return a Boolean as it is being used in the if statement to make a decision using X and Y so X and Y could also be of any type. So, X is inferred to be of type ‘a and Y to be of type ‘b and (op >) to be (‘a * ‘b -> bool).

Foo being a curried function which takes a function and a variable and a tuple and results in an int list.
It will have the below type:
val foo = fn: (‘a *‘b -> bool) -> ‘a -> ‘b * int list -> int list

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

Ans: 1. Encapsulation of data and code

2. Inheritance- creating a new type using the definition of an existing type

3. Subtyping with dynamic dispatch

(b) i. What is the "subset interpretation of subtyping"?

Ans: The definition of Subtyping says that a type can be used as if it were another type. So, if type B is a subtype of type A, then any value of type B can be used as a value of type A.
For example, if an Object type of class B is a child object of Object of type class A then Object of type class B would have all the methods required to qualify to become an Object of type class A with some additional methods of type class B. So Objects of type class B are subset of Objects of type class A with more restriction and can be used as a value for Object of type class A.

ii. Explain why function subtyping must be contravariant in the parameter type and covariant in the result type. If necessary, provide examples to illuminate your ex- planation.

Ans: 1. Contravariant in the parameter:

If B <: A then A -> int <: B -> int

The function subtyping must be contravariant in the parameter type as if we pass an Object of type parent (super) class to a function expecting an Object of type child class then the function might be using some features of the child class which might be missing in the parent class. But if we pass an Object of type subclass to a function expecting a superclass the subclass will have all the featured of the super class being derived from the same and hence will work without any problem.

For example, let 'vehicle' be the super class and 'car' be the derived class of vehicle and 'BMW' be the derived class of 'car'.

```
Procedure f (y: car){
  Int I;
  return I;
}
var x: vehicle;
```

f(x); -> will not work as procedure f expects an Object of type car, and an Object of type car might have extra attributes which would be missing in an Object of type Vehicle and the function f might me making use of those extra attributes. So passing an Object of type Vehicle would be wrong.

Whereas, if var z: BMW; f(z) -> will work, as BMW is a class derived from car it will have all the attributes of car and hence the function will work without any problem.

2: Covariant in result type:

If B <: A then int -> B <: int -> A

The function subtyping must be covariant in the result type as if function returns an Object of a subclass which is assigned to a variable of type super class, it will work as the subclass has all the attributes that are present in the super class. Whereas, if a function returns an object of type super class which is assigned to a variable of type subclass, it will not work as the super class might be missing some attributes that are present in the subclass.

For example, let 'vehicle' be the super class and 'car' be the derived class of vehicle and 'BMW' be the derived class of 'car'.

```
var f: int -> vehicle
```

function g(x:int){
return new car(x);
}
f = g; -> this assignment will work

whereas,
var g: int -> car
function f(x:int){
return new vehicle(x);
}
g = f; ->this assignment will not work

iii. Provide an intuitive answer showing why function subtyping satisfies the subset interpretation of subtyping.

Ans: As already discussed in the above solution, we can return a car where a vehicle is expected but not a vehicle where a car is expected as the super class might have few attributes missing that the subclass might have.
As all the cars are vehicles (being derived from that class cars) but the other way does not hold true, so all the cars can be considered as a subset of vehicles.
Similarly, as for a function of type int - > vehicle, int - > car can be used but not the other way. The set of all functions of type int - > car can be considered as a subset of the set of all functions of type int -> vehicle.
Interpreting in a generic way: if type B<: of type A then int -> B <: int -> A.
This is called covariant subtyping.

When an object is passed as a parameter, following the same example of vehicle and car, if a function is expecting a vehicle we can pass a car but the other way will not work due the same reason that vehicle might be missing few attributes that car might have.
So for a function of vehicle -> int, passing a function of type car - > int will work but not the other way. The set of all functions of type vehicle - > int can be considered as a subset of the set of all functions of type car - > int.
Interpreting in a generic way: if type B <: of type A then A->int <: B->int.
This is called contravariant subtyping.

Consolidating the above cases, it can also be written as if B <: A then A -> B <: B->A
As the subtyping relationship is transitive, it can be concluded as, if B <: A, C <: D then A -> C <: B -> D.

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

Ans:
```
Class E [ - T , + R ] ( x : T ,  y : R ){
        def  m ( x : T ) : R  =  y
}
```

(c) 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, r:Tree) 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.

Ans:
```
def minimum(t: Tree[T]):T{
        t match {
      case Node(v,l,r) => if((v < minimum(l)) && (v < minimum(r))) v
                          else if ((v > minimum(l)) && (minimum(l) >
minimum(r))) minimum(r)
                          else minimum(l)
       case Leaf(v) => v
```

(d) 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).

i. 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.

Ans:
```
class vehicle{}
    class car extends vehicle{}
    public static void main(String[] args){
         List<car> L = new ArrayList<car>();
         List<vehicle> L2 = L;
         L2.add(new vehicle());

    }
```

ii. 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 sub- typing. That is, make the function you wrote above be able to be called with many different instances of a generic class.

Ans:
```
class vehicle{

    }
    class car extends vehicle{

    }

    public class test{
```

```
    public static void main(String[] args){
          List<car> L = new ArrayList<car>();
          List<? super vehicle> L2 = new ArrayList<vehicle>();
          L2.add(new car());
          }
}
```

(e) 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].
Ans: Class C [+ E]

ii. Give an example of the use of your generic class.
Ans: def f (x : C[A]) = 6

f (new C[A]()) this will give res: Int = 6

f (new C[B]()) this will also give res: Int = 6

but if we define def f (x : C[B]) = 6
f (new C[A]()) this will give a type mismatch

(f) 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].
Ans: class C [- E]

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

Ans: def f (x : C[B]) = 6

f (new C[A]()) this will give res: Int = 6

f (new C[B]()) this will also give res: Int = 6

but if we define def f (x : C[A]) = 6
f (new C[B]()) this will give a type mismatch

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

Ans: Unlike Mark and sweep collector, reference counting collector do not stop the process and perform the garbage collection. In reference counting collector the storage reclamation happens in little bit of time.

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

Ans: The copying garbage collector has cheap allocation as compared to the mark and sweep garbage collector. As whenever a free block is need the copying garbage collector allocated the required space just above the heap pointer and moves the heap pointer up unlike mark and sweep collector where it has to traverse through a free list until it finds a free block of required size.

(g) Write a brief description of generational copying garbage collection.

Ans: Generational copying garbage collection works just like copying garbage collection with the below assumption.

The objects which have lived long are assumed to live longer hence the overhead of copying them again and again as in copying collector can be reduced. The new objects which get created would be dead soon.

So generational garbage collector uses a bunch of heaps with decreasing space.

Every time a new object is created it is allocated space in the lowest level of heap space. Once the lowest level is full the garbage collector is called and all the live objects from the lowest level are copied to the second lowest level then the program continues using the lowest level of heap space. With this process, there will be a time when the second last level will also be full, then the garbage collector copies the live objects to the third lowest level. At some point in this process we find all the long lived objects at the highest level of the heap space which are not copied again and again hence reducing the overhead of copying long lived objects on every call to garbage collector.

(h) 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.

Ans: delete(x)

```
{ x->refcount = x->refcount – 1;
        if x->refcount := 0 then
                for each pointer field x->p do
                { delete(x->p);
                        addToFreeList(x->);
                }
}
```