Programming Language – Homework 2

Yuqian Zhang

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.

Solution:

(λy.3) ((λx. x x) (λx. x x))

Under normal-order evaluation :3

Under applicative-order evaluation: will not terminates always be (λy.3) ((λx. x x) (λx. 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).

Solution:

Use recursive definition of Y combinator: Y f = f(Y f)

Fibonacci Number:

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

Give a name to the whole expression above as fib.

Suppose we apply fib to 2.

Y (λf. λx. …) 2 ⇔(λf. λx. …) (Y(λf. λx. …)) 2

β ⇔ λx. if (= x 0) 0
else if (= x 1) 1
(+ ((Y(λf. λx. …)) (- x 1)) ((Y(λf. λx. …)) (- x 2))) 2

β ⇔if (= 2 0) 0
else if (=2 1) 1
(+((Y(λf. λx. …)) (- 2 1)) ((Y(λf. λx. …)) (- 2 2)))

δ ⇔(+((Y(λf. λx. …)) 2) ((Y(λf. λx. …)) (- 2 2)))

δ ⇔(+((Y(λf. λx. …)) 1) ((Y(λf. λx. …)) 0))

β ⇔ λx. if (= x 0) 0
else if (= x 1) 1
(+ ((Y(λf. λx. …)) (- x 1)) ((Y(λf. λx. …)) (- x 2))) **1**

β ⇔if (= 1 0) 0
else if (=1 1) 1

Now we have: (+ 1 ((Y(λf. λx. …)) 0))

β ⇔ λx. if (= x 0) 0
else if (= x 1) 1
(+ ((Y(λf. λx. …)) (- x 1)) ((Y(λf. λx. …)) (- x 2))) **0**

β ⇔ **if (= 1 0) 0**
else if (=1 1) 1

Finally, reduced to (+ 1 0)

δ ⇔1 (the recursive result of fib 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)).
Solution:

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

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

$\beta\Leftrightarrow(\lambda x.\ f\ (x\ x))\ \lambda x.\ f\ (x\ x)$

$\beta\Leftrightarrow\ f\ \underline{(\lambda x.\ f\ (x\ x))(\ \lambda x.\ f\ (x\ x))}$

This is equivalent to the previous β reduction of Y f

$\beta\Leftrightarrow\ f\ (\ Y\ f\ )$

Thus, $Y\ f = f\ (\ Y\ f)$

(d) Summarize, in your own words, what the two Church-Rosser theorems state.
Solution:
1) All terminating reduction sequences for an expression will result in the same normal form.
2) If any reduction sequence terminates, then normal order reduction will 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)?
Solution:
Because ML is statically typed.

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

```
 fun compose f g x =
          let val (v::vs) = (f x)
               val(j::js) = (g v)
          in j
          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
```

Solution:
val foo = fn : ('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.
Solution:

1) Determining the type of z as a list of integer. Because in let fun bar, a and z are connected with ‘then’ and ‘else’, an a is a list of integer type, which implies the result is list of integer as well.
2) Determining the return type of (op >) is Boolean type, because if x > y then implies that the result of true or false.
3) x and y as parameters of the infix (op >) have no clue what type they are. Thus, x->’a,y->’b

3.
(a)As discussed in class, what are the three features that a language must have in order to considered object oriented?
Solution:
1)Encapsulation of data and code into a single data structure.
- fields and methods within an object.
2)Inheritance
- define a new type based on an existing type, where the new type can reuse code defined in the existing type.
-the child type inherits the methods of the parent type as well as the fields.
3)Subtyping with dynamic dispatch
-subtyping: treating a type as if it were another type. anywhere a type T can be used, a subtype of T can be used.
-dynamic dispatch: determining at run time which method to call, based on the actual type of an object, not the declared type.

(b) What is the “subset interpretation of subtyping”?
Solution:
The set denoted by the parent type is a superset of the set denoted by the child type. Equivalently, a subtype denotes a subset of the set denoted by the parent type.

(c) Explain why function subtyping must be contravariant in the parameter type and co- variant in the result type. If necessary, provide examples to illuminate your explanation.
Solution:
Using ML language to explain, though ML does not have function subtyping.
1) function subtyping must be contravariant in the parameter type.
Assume B is a subtype of A. (means A is not a B, but B is A)
fun f (g: A-> int)= g(new A()) ; [A->int]
fun h(x:B) = x.foo() +1 ; [B->int]
⇨ f(h) is not ok!
You cannot use a B->int where an A->int is expected. Because the foo function may not exist in A.
fun f (g: B-> int)= g(new B()) ; [B->int]
fun h(x:A) = x.foo() +1 ; [A->int]
⇨ f(h) is ok!
You can use an A->int wherever a B->int is expected. Because B is A.

2) function subtyping must be co- variant in the result type.
Assume D is a subtype of C. (means C is not D, but D is C)
fun f (g:int->C) = let y :C= g(6);
fun h (x:int) = new D();
⇨ f(h) is ok!
You can use an int->D wherever a int->C is expected Because D is C.
fun f (g:int->D) = let y :D= g(6)
fun h (x:int) = new C()
⇨ f(h) is not ok!
You cannot use an int->D wherever a int->C is expected Because C is not D.

(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.
Solution:
Since any function expecting an A can be applied to B, any function in A ->int is also in B->int. So, A->int is a subset of B->int.
Since B object is an A object, any function returning B also returns A. So, any function in int->B also in Int->A. So, int->B is a subset of int->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.
Solution:
contravariance on the parameter type:
The assumption is B is the subset of A, B extends A.
class A
class B extends A
def fa (g: A=>Int)= g(new A()) expects[A->int]
def fb (g: B=>Int) = g(new B()) expects[B->int]
def p (x: A) = 6 p is type of A->int
def q (x: B) = 6 q is type of B->int
fa(p) works fine.
fa(q) error. A is not B.
fb(p) works passing an A->Int to a function expecting a B->Int, works fine (contravariant subtyping)
fb(q) works fine.
covariance on the result type:
def f (g:Int=>A) = g (5)
f ((x:Int => new B()) passing int->B to a function expecting an int->A works fine(covariance subtyping)

4.

(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.

Solution:

```
void Add (C<A>a ){
        a.add(new A());
}
C<B> b = new C<B>();
Add(b);
B x = b.get(0)// run-time error: Cause it does not know which type of object to get.
```

(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.

Solution:

```
void Add (C<? super B>b ){
        b.add(new B());
}
C<A> a = new C<A>();
Add(a);
```

5.

(a) In Scala, write a generic class definition that supports covariant subtyping among in- stances 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].

solution:

```
class C[+E](x: E){
  |  def  f (y : Int) : E =  x
}
```

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

```
C[A] = new C[B]
```

(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].

solution:

```
class C[-E]{
  |  def  f (x : E): Int = 0
}
```

(b) Give an example of the use of your generic class.
C[B] = new C[A]

(e)

```
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]
```

Solution:

```
def smallest [T<:Ordered[T]] (tr: Tree[T]): T =  {
        case Leaf (v)=> v
        case Node(v,l,r)=> (v. min(smallest(l))).min(smallest(r))
}
```

6.

(a) What is the advantage of a reference counting collector over a mark and sweep collector?
The benefit of this is that it works a lot better for real-time programs, because it doesn't "stop the world". Storage reclamation is incremental, happens a little bit at a time.

(b) What is the advantage of a copying garbage collector over a mark and sweep garbage collector?
The benefit to this, is that it automatically compacts these objects as they're copied, making allocation very easy with a heap pointer. It also means that the cost of garbage collection is only proportional to the size of the live objects, O(L).

(c) Write a brief description of generational copying garbage collection.
This just like the copying GC, but it has more than two heaps. You start with a large heap, for "generation 0", where objects are allocated. When this fills up, you copy all live objects from the gen 0 heap to the next heap, gen 1. When gen 1 fills up, you copy every live object into gen 2, and so on.

(d) Write, in the language of your choice, the procedure delete(x) in a reference count- ing 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.

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