Asad Ranavaya

CSCI-GA-2110-1-001

Prof. Goldberg

19 December 2021

Homework 2

1. SML
   1. All lists in ML must be homogeneous to allow for static type checking by the compiler.
   2. fun f g h i x = [let val (y::ys) = g x

val (z::zs) = h y

val z = i z

in z

end]

* 1. ('a \* 'b -> bool) -> ('c \* 'd -> 'a) -> ('c -> 'b) -> 'c \* 'd -> int -> int
  2. The ML type inference would infer the function as follows: The first variables type is found by looking at “z+1” where addition means z is an int. Then it would look at the “op <” function which operates on tuples. It returns ('a \* 'b -> bool). Because of this, the two functions used by the infix must return 'a and 'b. Therefore the tuple (x,y) must be ('c \*'d). With these it can be seen that function f’s type is ('c \*'d -> 'a) and function g’s type is ('c -> 'b).

1. Lambda Calculus
   1. A function that would reduce to normal form under normal order but not under applicative order is:  
      (λ y. 1) ((λ x. (x x)) (λ x. (x x)))
   2. Using the Y combinator, the Fibonacci recursive function can be written as  
      Y (λf. λx. if (= x 0) 1 (if (= x 1) 1 (+ (f (- x 1)) (f (- x 2)))))  
      Reductions are as follows:  
      As in the lecture, the name of the function in Y f = f (Y f) can be written as fib which will replace the Y (λf. λx. if (= x 0) 1 (if (= x 1) 1 (+ (f (- x 1)) (f (- x 2))))), where fib is Y f.  
      Fib 3 = Y (λf. λx. if (= x 0) 0 (if (= x 1) 1 (+ (f (- x 1)) (f (- x 2))))) fib 3  
      = (λx. if (= x 0) 0 (if (= x 1) 1 (+ (fib (- x 1)) (fib (- x 2))))) 3  
      = (λx. if (= 3 0) 0 (if (= 3 1) 1 (+ (fib (- 3 1)) (fib (- 3 2)))))   
      = if false 0 (if (= 3 1) 1 (+ (fib (- 3 1)) (fib (-3 2))))  
      = if false 1 (+ (fib (-3 1)) (fib (-3 2)))  
      = + (fib 2) (fib 1)  
      = + ((λx. if (= x 0) 0 (if (= x 1) 1 (+ (fib (- x 1)) (fib (- x 2))))) 2) ((λx. if (= x 0) 0 (if (= x 1) 1 (+ (fib (- x 1)) (fib (- x 2))))) 1)  
      = + (if ( = 2 0) 0 (if (= 2 1) 1 (+ (fib (- 2 1)) (fib (- 2 2))))) (if ( = 1 0) 0 (if (= 1 1) 1 (+ (fib (- 1 1)) (fib (- 1 2)))))  
      = + (if false 0 (if (=2 1) 1 (+ fib(-2 1)) (fib (-2 2))) (if false 1 (if (=1 1) 1 (+ (fib (-1 1))(fib (-1 2))))  
      = + (if false 1 (+ (fib (-2 1)) (fib (-2 2)))) (if true 1 (+ (fib (-1 1))(fib (-1 2))))  
      = + ( (+ fib 1 fib 0) 1)  
      = + ( (+ ((λx. if (= x 0) 0 (if (= x 1) 1 (+ (fib (- x 1)) (fib (- x 2))))) 1) ((λx. if (= x 0) 0 (if (= x 1) 1 (+ (fib (- x 1)) (fib (- x 2))))) 0)) 1)  
      = + ( (+ (if (= 1 0) 0 (if =1 1) 1 (+ (fib (-1 1)) (fib (-1 2)))) (if(= 0 0) 0 (if (= 0 1) 1 (+ fib(-0 1)) (fib (- 0 2))) ) 1)  
      = +( (+1 0) 1)   
      = +(1 1)  
      = 2  
      As shown, the Fibonacci of 3 is 2 using lambda calculus and the Y-combinator.
   3. The expression in λ-calculus representing the Y-combinator is:  
      Y = (λh. ((λx. (h (x x))) (λx (h (x x)))))  
      It satisfies Y(f) <-> f(Y(f)) by:  
      Y f = (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))))  
      Using the previous statement of Y f <-> ((λx. (f (x x))) (λx (f (x x))))  
      It can be seen that:  
      f ((λx (f (x x))) (λx (f (x x)))) is interchangeable with f (Y f).  
      As a result, the Y-combinator satisfies this property: Y f <-> f( Y f)
   4. Church-Rosser Theorem I states that an expression can only be reduced to one normal form. Church-Rosser Theorem II states that if an expression has a terminating reduction to normal form, then normal order reduction will also reduce the expression to normal form.
2. OOP
   1. The three features of an Object-Oriented Language are as follows:
      1. Inheritance
      2. Encapsulation of data
      3. Object subtyping
   2. Subset interpretation of subtyping is a way to define how sets of objects interact. More specifically, a subtype’s set of values is a subset of the values of the parent.
   3. Class derivation in Java allows subtyping where if a class B extends a parent class A, then every function that can be invoked on A can also be invoked on B, or the values of type B are a subset of the values of A.  
      Example:  
      class A{}  
      class B extends A{}  
      A a = new B();   
      Anything that the object A can call is already defined in class B because it extends A.
   4. Function subtyping in languages that allow it such as Scala, allow it in contravariance position, covariance position, but not both. If we have for example, a class A and class B which extends A, it goes to follow that all of A’s members are also in B. For example, a function that takes a class A and returns a generic type can have parameters of both A and B because the return of type of the function whose parameter is A is a subset of the return for parameter B.

Likewise, the opposite is true for a covariant function subtype. If you have a function which takes a generic parameter T and returns a type B, then it also returns a type A. Therefore, A and B can both be used in the function.

These restrictions that Scala has emplaced on contravariance and covariance positions allows the language to safely use subtyped functions.

Example in Scala:

Class A (a : integer){

Val x = a

}

Class B (a : integer, b : integer) extends A(a){

Val y = b

}

Object main {

Def testFunc1 (a : A) : B = new B(1, 2)

Def testFunc2 (b : B) : A = new A(1)

Var bToA : B = testFunc1(bToA)

Var aToB : A = testFunc2(aToB)

}

In this situation testFunc1 works on bToA because B is a subtype of A. However, testFunc2 does not work because B -> A is not a subset of A -> B.

1. Java Generics
   1. Code:  
      void testFunction(ArrayList<A> list){  
       A a = new A;  
       list.add(a);  
      }  
      In this function, if list is of type ArrayList<B>, the code would try to inject type A into a list of type B. Later if the ArrayList list is accessed in say a get method, it will try to store an A object into a B object.
   2. Code:  
      void testFunction(ArrayList<? Super A> list){  
       A a = new A;  
       list.add(a);  
      }  
      Super typing A allows the testFunction to be called with many different instances of a generic class.
2. Scala
   1. minTree is defined as follows with helper function min:  
      def min[T <: Ordered[T]] (val1:T, val2:T, val3:T) ={  
       if (val1 < val2)  
       if (val1 < val 3)   
       val1  
       else   
       val3  
       else  
       if (val2 < val3)  
       val2  
       else  
       val3  
      }  
        
      def minTree[T <: Ordered[T]](tree:Tree[T]) : T = {  
       tree match = {  
       case Leaf(x) => x  
       case Node(x, l, r) => min(x, minTree(l), minTree(r))  
       }  
      }
   2. Scala generic class that supports covariant subtyping:  
      1. class C[+E] (a:E) {  
          val b:E = a  
          def getVal():E = b  
         }
      2. class A {}  
         class B extends A{}  
         object testMain{  
          def testFunction(x : C[A]) : A = {  
          x.value()  
          }  
          def main(args: Array[String]) {  
          val cClass = new C[A](new A)  
          val aClass: A = testFunction(cClass)  
          val c2Class = new C[B](new B)  
          val a2Clas: A = testFunction (c2Class)  
          }  
         }
   3. Scala generic class that supports contravariant subtyping:
      1. class C[-E]() {  
          val a : List[] = List()  
          def addValue(b : E){  
          a = a ++ b  
          }  
         }
      2. class A{}  
         class B extends A{}  
         object testMain{  
          def testFunction(x: C[B], y :B){  
          x.addValue(y)  
          }  
          def main(args: Array[String]) {  
          val cClass = new C[B]()  
          val bClass:B = new B  
          testFunction(cClass, bClass)  
          val c2Class = new C[A]()  
          testFunction(c2Class, bClass)  
          }  
         }
3. Garbage Collection
   1. Mark and Sweep is more advantageous over a reference counting collector in that it can easily handle cycle structures. It also has better space utilization.
   2. Copying garbage collectors offer object compacting which helps with heap fragmentation and also allows for a heap pointer. This means that no free list is needed to allocate new objects. In terms of size, mark and sweep is proportional to the size of the heap because it must trace the whole heap, while copying garbage collectors only use the size of live objects.
   3. Generational garbage collection is based off of the statistical probability that objects that live longer need to be collected from memory less often than short lived objects. It works by creating a series of heaps, each with an associated lifetime of the objects it holds. Each object is added to the youngest generation, and every time the heap needs to perform garbage collection, the surviving objects get promoted to the next heap, which signifies a longer lasting object. This method greatly increases the performance of garbage collection because the total size of the live objects on the lowest heap will be small compared to the entire heap.
   4. Code:  
      public void delete(void\* x){  
       //decrement reference count  
       x->references -= 1;  
       if(x->references == 0){  
       for(int i = 0; i < x->sizeOfChildren; ++i){  
       delete(x->childPntr[i]);  
       }  
       addToFreeList(x);  
       }  
      }