Type inference with functions:

currying

add(x, y) => an integer that is the sum of x and y

intput: (int x int) → output: int

add: int → int → int

add(6) with currying, this produces a function:

add6(y)

add6(10) => 16

append(l1 l2)

append(‘(a b c)) => appendabc

appendabc(‘(x y z)) => ‘(a b c x y z)

Types and subtypes

Many languages (Java) lets us create a subtype of a type were the subtype is going to be more specific. Then we can use type subtype where ever the type is expected.

Ex: public class MyFrame extends JFrame {

MyFrame is a subtype of JFrame

public class Integer implements Comparable<Integer> {

Integer is a subtype of Comparable

Parameterized type: A type that contains a type.

LinkedList<T>

LinkedList<String>

String[] Array is a type, it is parameterized to say what is stored in the array

What about subtypes of parameterized types?

What about types where the parameterized type is a subtype?

Ex: LinkedList<Object> vs. LinkedList<String> (String is a subtype of Object)

Object[] vs String[]

Are these types comparable, and when are they compatible?

T[A] T[B] T is a parameterized type, and A and B are instantiations of the parameter. B is a subtype of A

Covariant: The types are compatable if the parameterized type is a subtype. We can use T[B] where ever T[A] is expected

contravariant: The types are compatible if the parameterized type is a super type. We can use T[A] where ever T[B] is expected

invariant: The parameterized type must exactly match. If T[A] is expected, we can’t use T[B]. If T[B] is expected, we can’t use T[A].

Java arrays are covariant. If you have Object[] x, we can use any nonprimitive type array for x.

List<String> l, we can use ArrayList<String> for l. ArrayList<String> is a subtype of List<String>.

Square extends Rectangle. LinkedList<Rectangle> list. Can I use LinkedList<Square> assigned to list? No.

Java parameterized types with generics are invariant.

For arrays, Java has to add code into the program everytime there is an array assignment to make sure the value assigned is the appropriate type.

String[] stringArray = new String[10]

Object[] genArray = stringString

genArray[0] = new JFrame() ← if genArray is really a String array, this has to prevented, but can’t be done with static type checking.

Public static int linearSearch(Object x, Object[] array) {

public static <T> int linearSearch(T x, T[] array) {

public class B extends A {

B can override a method of A.

When can I use a function to override another function (in Java)?

C → D → E → F (input types C, D, E, output type F)

(Java requires the same name, same input parameter signature, the output types must be the same or narrower)

Overloading is invariant for the input types, covariant on the output type.

Public class Rectangle implements Comparable<Rectangle> {

Public class Square extends Rectangle {

Public static <T extends Comparable<? super T>> int binarySearch(T element, LinkedList<T> list)

works with LinkedList<Rectangle>, but not with LinkedList<Square>

Square extends Comparable<Rectangle>

LinkedList<? Super Square> list

so I can assign LinkedList<Square>, LinkedList<Rectangle> or LinkedList<Object> to list. In this case, we have contravariance.