# A Study in *Scala*bility through Abstraction

Troy Hu and Benjamin Killeen

***Abstract*—The Scala programming language combines object-oriented and functional programming in a single Java-like language intended for mass-adoption. It emphasizes component abstraction in a statically typed environment that enables remarkably modular software design. In this paper, we provide an overview of notable features in the Scala language as well as motivating examples that underlie these aspects.**[1]

## I. Introduction

Software components are simply self-contained parts of software used by larger parts or entire applications. Modern software typically relies on many common components like hashable types or iterable data structures to use as building blocks. Component abstraction enables the generalization of implementation for these features, greatly reducing duplication of effort. It is one of the most powerful tools in the programmer's utility belt, a fact which the designers of the Scala programming language aimed to exploit [1]. In the early 2000s, existing languages had little support for type-sound component abstraction, including widely used languages like Java and C#. Odersky *et al.* address this shortcoming with a language that combines elements of object-oriented and functional programming in a statically typed environment. Scala, which gets its name from "scalable," provides a powerful interface for abstraction within a development framework intended for mass adoption.

In pursuit of usability, Scala borrows many syntactic elements from Java and C#, and it integrates smoothly with components from these languages. In fact, the Scala library includes standard Java objects like `java.lang.String`, as shown in Listing 1. Scala code can take advantage of existing implementations in Java, and in the end it compiles to Java bytecode, making Scala packages available to Java programmers. At the same time, Scala maintains a distinct programming paradigm from either Java or C#. It discards some features from these languages, and it develops completely novel ideas from the $\nu$Obj calculus [2]. The example in Listing 1 highlights syntactic similarities between Java and Scala, comparing two implementations of the same program. Note how Java prepends type declarations before terms, whereas Scala affixes type declarations using the `:` operator. This and other changes effect a terser, more expressive syntax overall.

Of course, Scala's foremost strength comes from its typing system. Abstract class definitions and path-dependent types utilize the $\nu$Obj Calculus, enabling incredible flexibility through the use of traits and mixins [2]. Somewhat akin to Java's abstract classes, traits allow a programmer to rely on abstract methods for common functionalities. For example, the `Equiv[T]` trait in Fig. 3 represents an equivalence relation

[1]Source for this paper and its examples available at github.com/bendkill/scala_project.

```java
// Java:
class PrintOptions {
  public static void main(String[] args) {
    System.out.println("Options selected:");
    for (int i = 0; i < args.length; i++)
      if (args[i].startsWith("-"))
        System.out.println(" " +
          ↪ args.substring(1))
  }
}
```

```scala
// Scala:
object PrintOptions {
  def main(args: Array[String]): unit = {
    println("Options selected:")
    for (val arg <- args)
      if (arg.startsWith("-"))
        println(" " + arg.substring(1))
  }
}
```

Listing 1: Notice how Scala's general syntax and structure are similar to Java's. At the same time, there are some visible differences, e.g., unit is returned in the Scala implementation instead of void in the Java implementation.

on the type `T`, abstracting the definition of `eq` on which a concrete method, `neq` depends. Mixins enable a class to inherit from multiple traits. For instance, one might use `Equiv` in conjunction with an `Ordering` trait to represent separate relations on the same type in one object.

Finally, the uniform object model in Scala provides a cohesive programming environment. Every Scala value is an object, and every operation is a call to a method. The boolean `true`, for example, is a singleton object that extends (inherits from) the Boolean trait (see Sec. III-B). At the same time, Scala fits into a functional programming paradigm. Functions themselves are values, and the syntax allows for SML-like decomposition and pattern matching. These features result in a powerful language with a unified programming experience.

## II. Related Work

The ideas underlying Scala come from a variety of work. First and foremost, the designers rely on the $\nu$Obj calculus outlined in Odersky *et al.* (2003) for a theoretical grounding on which they build Scala's robust type system [2]. Although a thorough discussion of the $\nu$Obj calculus is beyond the scope of this paper, which focuses on the Scala language proper, we describe its main points in brief. Additionally, the ideas in Scala have been replicated or expanded on in more recent work, which we discuss here.

The $\nu$Obj calculus is, broadly, a calculus and dependent type system that includes classes and objects with types

as members [2]. *Dependent types* rely on some value for definition, and with this formalism, the $\nu$Obj calculus expresses Java's inner class system, virtual types, and family polymorphism. Furthermore, it models SML-style modules and functors, diverging from standard object-oriented type systems in three fundamental ways:

1) Objects *and* classes are considered primitive, as opposed to just objects.
2) Type checking and evaluation rely on name references rather than object records.
3) Object types are expressible using possibly nominal type components.

These alterations make possible the rigorous blend of object-oriented and functional styles evident in Scala, differing markedly from traditional languages like Java and C#.

The Rust programming language is a popular alternative to Scala that is designed to target systems-level applications [3]. The language is similar to C++ in that it allows users to interact directly with hardware. Moreover, the syntax of Rust is very similar to that of C++. Rust, however, is unlike C++ in that any program purely written in Rust is guaranteed to avoid memory errors and data races. Most notably, Rust utilizes many of Scala's core concepts to achieve its goals. Like Scala, Rust is both a functional and object-oriented programming language (to model C++). Rust also ensures type soundness in order to avoid memory errors such as dangling pointers. Unlike Scala, however, Rust emphasizes concurrency and paralellism.

The remainder of this paper focuses on the practical considerations of programming in the Scala language.

## III. Programming in Scala

Being a fully developed programming language, Scala implements numerous useful features that encourage modularity and reusability in software. In this section, we highlight just a few of these, as well as provide an overview of the general process through which software construction takes place in Scala.

### A. Unified Object Model

In Scala, every value is an object and every class is a subtype of the `Any` class. Fig. 1 shows an overview of the unified object model in Scala's standard library, under which many classes simply absorb the existing Java implementation. Immutable objects or *values*, denoted with the `val` keyword on declaration, inherit from the class `AnyVal`, and include classes like `scala.Boolean` and `scala.Char`. Mutable *variables* on the other hand, inherit from `scala.AnyRef`, which is synonymous with the generic `java.lang.Object`. These include classes like `scala.List` or traits like `scala.Ordered`. Even functions fit into this object model, such as the `scala.Function1` trait, which single-argument functions implement.

One of the fundamental aspects of Scala is its invariance with respect to value representation. Under this rule, *interpreting an value belonging to some subclass as an element of its superclass does not change that value's representation.* Put another way, this invariance rule demands that for types `S`, `T` with `S <: T`, and `x` of type `S`,

```
x.asInstanceOf[T].asInstanceOf[S] = x
```

always holds. Because of this rule, Scala's object model includes some unusual relations not found in more traditional languages. In particular, Fig. 1 uses dashed lines to denote a *View* from one type to another. As discussed in Sec. III-C, a view enables interpretation between types which are not strictly subtypes of one another. For example, both `Int` and `Float` maintain a member called `MaxValue`, which other function can rely on without requiring a strict subtype of either `Int` or `Float`. Finally, the `scala.Nothing` trait is a subtype of every other class, allowing for flexible objects like the `scala.List` object `Nil`, of type `List[Nothing]`.

Another fundamental feature of Scala's object model inclusion of operations, which are invocations of a method. For example, the usual addition operator `+` can be implemented like any other method, as in Lst. 6, and invoked in the exact same way: `x.+(y)`. Of course Scala allows for the more conventional calling style `x + y`, but it does so merely through syntactic sugaring. Here, `x` is the receiver object, `+` is a method defined in `x`, and `y` is the method's argument.

Finally, Scala differs from Java with regard to the construction of objects. Rather than define a dedicated constructor function within objects, the class name itself is simply the constructor. Object instantiations simply runs the entire body of the class, instantiated member variables, values, and definitions.

### B. Traits, Objects, and Classes

Traits in Scala are somewhat akin to abstract classes in Java, defining abstract members that should be implemented by more concrete classes. As a very simple example, consider the `Nat` trait which defines necessary operations on natural numbers:

```
trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
}
```

Classes which inherit from `Nat` must implement the `isZero`, `succ`, and `pred` methods with the corresponding types in order to inherit from `Nat`. This is advantageous for several reasons. First, as we shall see, the `Nat` class may use these abstract methods to define more complex functions, which subclasses need not implement. Second, other functions may refer to `Nats` as a whole rather than either subclass of the original trait.

Objects or singleton objects in Scala often accompany trait definitions. A singleton object is at the same time a class definition and the sole instantiated member of that class. For instance, the `Z` object represents the natural number "zero," of which there can only be one:

```
object Z extends Nat {
  def isZero = true
  def succ = S(this)
  def pred = this
}
```

This simple implementation behaves as we would expect,

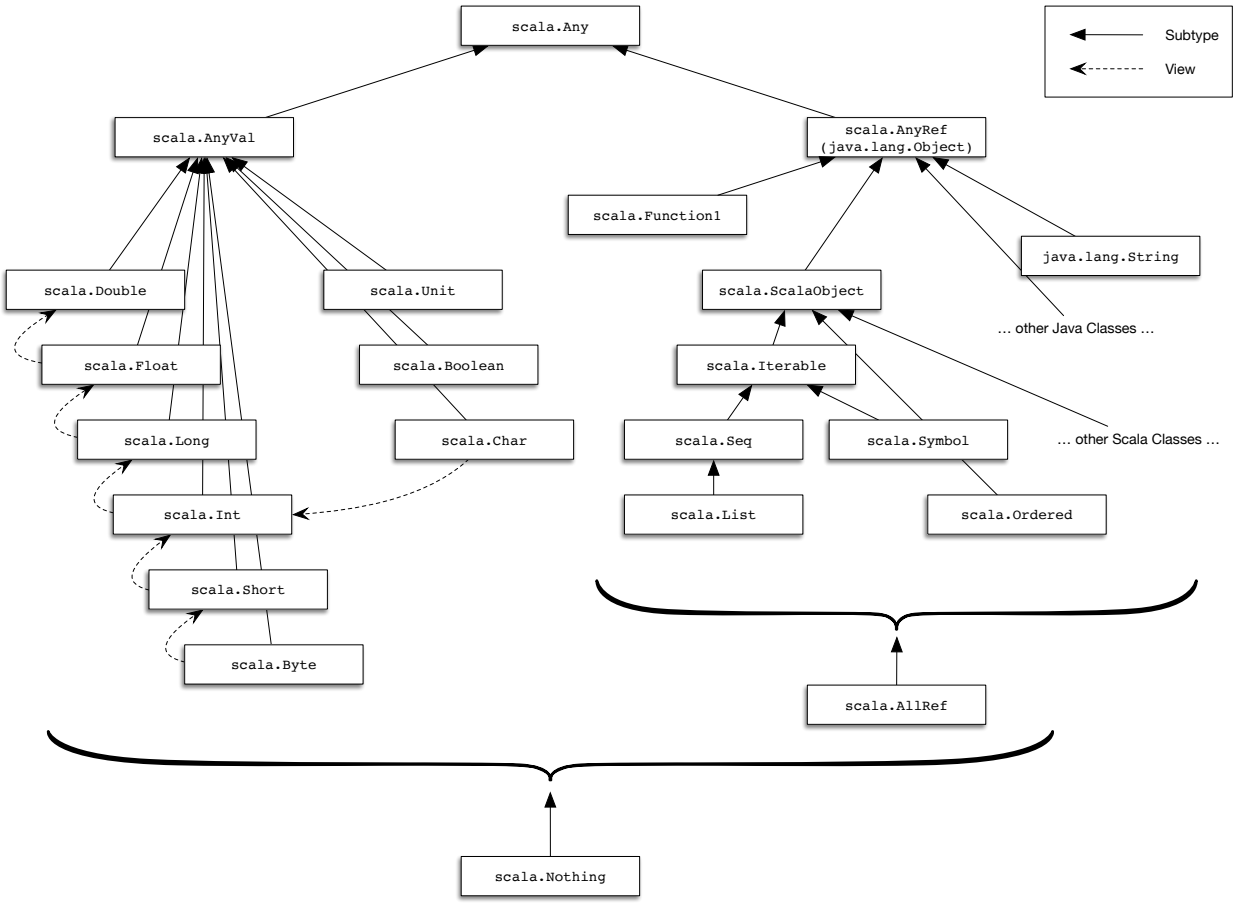


Fig. 1: A visual representation of Scala's class hierarchy

relying on the `S` class (below) to implement `Z.succ`. The choice to ensure `Z.pred = Z` was made in keeping with conventional evaluation rules. The use of the `extends` keyword represents the inheritance from `Nat`.

Similarly classes in Scala can "extend" a trait, while taking in arguments to the constructor. The `S` class

```
case class S(n: Nat) extends Nat {
  def isZero = false
  def succ = S(this)
  def pred = n
}
```

represents the "successor" of another natural `n`, either `Z` or another `S`. Here, the use of the `case` keyword results in a constructor very similar to constructors in SML, which can be instantiated without the use of `new` and easily deconstructed in pattern matching blocks.

### C. Views

In place of loose subtyping, Scala requires views to implicitly convert objects from one type to another. A view is implemented with a method that takes in an arguemnt of one type and returns an object of another type. The only difference between a view method and a normal method is that view methods require the `implicit` modifier, which goes before the method definition. This modifier allows the Scala compiler to know that it is the implcit conversion method when converting from one type to another. Scala implictly applies a view to an expression, $e$ of type $T$, when one of the following cases occur:

- The expected type of $e$ is not of type $T$.
- A member selected from $e$ is not a member of $T$.

For example, in Lst. 2, `listToSet` is the view that converts `GenList[T]` to `Set[T]`. The compiler inserts applications of the view onto `xs`.

```
trait Set[T]{
  def include(x: T): Set[T]
  def contains(x: T): boolean
}

implicit def listToSet[T](xs: GenList[T]):
 ↪  Set[T] =
  new Set[T] {
    def include(x: T): Set[T] =
      xs prepend x
    def contains (x: T): boolean =
      !isEmpty && (xs.head == x || (xs.tail
       ↪  contains x))
  }
```

Listing 2: An example of views and implcit conversions.

### D. Type Parameters

Scala's focus on component abstraction would not be complete without the inclusion of type parameters or variables. These allow a function or class to be implemented without a specific type in mind while remaining type sound. As a simple example, consider the identity function $f(x) = x$ which has type $\tau \to \tau$. This cannot be constrained to any particular type `Int` or `Boolean` without fundamentally changing its behavior. Fortunately, Scala allows for this flexibility with type variables:

```
def identity [T] (x: T): T = x
```

In this definition, `[T]` designates a type without constraints which the argument `x` belongs to, as well as the return type of the function, also `T`. Since the body of function simply returns $x$ without invoking any methods or assuming any properties, this is valid.

Scala traits can also rely on type parameters in much the same way. Lst. 3 shows a very simple trait that defines equivalence relations on arbitrary types, as long as an implementation of `eq` exists on that type. Not that one must distinguish between a type that can be compared for equality and an equality relation on some type. `Equiv[T]` represents the latter. Lst. 4 shows a similar trait that describes an ordering on some type `T`, which we use to implement ordered lists in the abstract.

```
trait Equiv[T] {
  def equal  (x: T, y: T): Boolean;
  def neq (x: T, y: T): Boolean = !equal(x,
   ↪  y)
}
```

Listing 3: An abstraction of equivalence relations in Scala.

```
trait Ordering[T] {
  def leq (x: T, y: T): Boolean;
  def geq (x: T, y: T): Boolean =  leq(y, x)
  def gt  (x: T, y: T): Boolean = !leq(x, y)
  def lt  (x: T, y: T): Boolean = !leq(y, x)
  def eq  (x: T, y: T): Boolean =  leq(x, y)
   ↪  &&  leq(y, x)
  def neq (x: T, y: T): Boolean = !leq(x, y)
   ↪  || !leq(y, x)
}
```

Listing 4: An abstraction of ordering relations in Scala.

Scala also allows for constraints on type parameters. These include the subtype, supertype, and view relations: `<:`, `>:`, and `<%` respectively. These can be used in concert with one another, as in the following method for lexicographic comparison of lists:

```
trait LexList[+T] {
  // ...standard list methods...
  def < [U >: T <: Ordered[U]](that:
   ↪  LexList[U]): Boolean = {
    !that.isEmpty &&
    (this.isEmpty ||
     this.head < that.head ||
     this.head == that.head ||
     this.tail < that.tail)
  }
}
```

In `LexList.<`, `that` must contain elements that are supertypes of the elements of `this`, since `U >: T`. Moreover, both elements must inherit from the standard `Ordered[T]` trait, since `U <: Ordered[U]`. This makes possible the calls to `<` and `==`. View relations are used similarly, where `U <% T` requires that an implicit view exists to convert objects of type `U` to type `T`.

Crucially, Scala includes strict enforcement of the variance of these types. A type constructor `C` is *covariant* if for `A <: B`, `C[A] <: C[B]`. Similarly, `C` is *contravariant* if `C[A] >: C[B]` and invariant if neither relation can be guaranteed. Type parameters are declared invariant by default and covariant or contravariant using the + or − prefixes respectively. In the `LexList` example as well as Lst. 7, we declare the type constructor to be covariant, as lists should be. Scala enforces this declaration at compile time by ensuring that `T` is used in covariant positions throughout the body of the trait.

This enforcement is commonly a source of some confusion. The `OrdList[T]` from Lst. 7 has an `insert` method, for instance, that takes an argument `x: U` where `U >: T`. Intuitively, one would think that `x: T` should be sufficient, but in fact this is not possible in order for `OrdList` to be covariant. Consider the bad definition:

```
// bad typing for insert!
trait OrdList[+T] {
  def insert(x: T, o: Ordering[T]):
   ↪  OrdList[T];
}
```

and suppose we had `Integer` in addition to `Nat`, both with a common supertype `Number`. Since `Nat <: Number` and `Integer <: Number`, covariance requires that

```
OrdList[Nat] <: OrdList[Number]
```

and

```
OrdList[Integer] <: OrdList[Number]
```

Suppose `insert` accepted `x: T` for `OrdList[T]`. Then an `OrdList[Nat]` could not insert an `Integer`, a fact which causes the following function to fail:

```
def pushNumber (x: Number,
                l: OrdList[Number],
                o: Ordering[Number]):
    OrdList[Number] = l.insert(x, o)
```

Under subtyping, `push` accepts an `x: Integer` and `l: OrdList[Nat]`, but with these arguments, the call to `insert` will fail. Under the proper type definition, however, inserting `x` into `l` will result in an `OrdList` of their common supertype, `Number`. This is proper covariance.

### E. Mixins

Scala allows for multiple inheritance through the use of mixins. These address the diamond inheritance problem, which refers to possible ambiguities that arise from multiple inheritance. Every class inherits methods primarily through the first superclass listed, allowing for a single path of superclass dependencies. This disambiguates collisions of class members, as in Lst. 5, which implements the usual ordering on $\mathbb{N}$

by defining `leq` as well as an equivalence relation that compares the evenness of two naturals, through `equal`. The collision here occurs between `Ordering[T]`'s `neq`, which depends on `leq`, and `Equiv[T]`'s `neq`, which depends on `equal`. Because `NatOrderingWithEquiv` extends first from `Ordering[Nat]`, it will use the definition that depends on `leq`.

```
trait NatOrderingWithEquiv extends
 ↪  Ordering[Nat] with Equiv[Nat] {
  def leq (x: Nat, y: Nat): Boolean =
   ↪  NatOrdering.leq(x,y)
  def equal (x: Nat, y: Nat): Boolean =
   ↪  x.isEven == y.isEven
}
```

Listing 5: A single object representing two relations on Naturals, an ordering and an equivalence relation.

## IV. Application: Peano Arithmetic Calculator

Although this paper includes significant engagement with the Scala language merely to describe its features, we consider illustrative examples a poor indicator of a language's friendliness toward component abstraction. In order to truly evaluate Scala's scalability, we implement a full-scale interpreter for a small calculator language called `peano`. The full description of this language is beyond the scope of this paper, but it includes sufficient complexity to take full advantage of Scala's flexibility. `peano` gets its name from the Peano Arithmetic, with which all expressions are evaluated [4]. The `peano` interpreter is implemented in a mostly functional style with separate objects `Scanner`, `Parser`, `Desugarer`, `TypeChecker`, and `Evaluator` that interpret `peano` expressions. It includes the standard operations on naturals, integers, and rationals: addition, multiplication, *etc.*, while also providing higher level functions like GCD and LCM. These challenges gave us the opportunity to work with Scala in the same setting that large-scale developers do, using its built-in project manager `sbt`.

In general, we found Scala to be simple to work with. Scala development is vast improvement over existing workflows in Java and C#, even without taking advantage of component abstraction, due to the language's more expressive syntax. It's error messages are informative, and the built-in package manager `sbt` takes care of the many nuances of software construction and deployment. The scope of `peano`, initially envisioned as a much simpler calculator language that used built-in numeric types for evaluation, attests to our enjoyment of programming in Scala.

The source for `peano` as well as basic usage instructions can be found at github.com/bendkill/peano.

## V. Discussion

After 15 years, Scala has been regularly updated and is currently at stable release version 2.12.8. Over this period, the core principles of Scala have remained the same. Scala has achieved widespread adoption in the industry with companies such as Twitter and Apple utilizing the language. Moreover, Scala has both a large academic and non-academic user base. The language is often cited or used in computer science research. In addition, Scala's user community is thriving. There exist chatrooms, subreddits, and research conferences devoted entirely to Scala. The community maintains up-to-date documentation and installation instructions at scala-lang.org.

## References


[1] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the Scala programming language," Tech. Rep., 2004.

[2] M. Odersky, V. Cremet, C. Röckl, and M. Zenger, "A Nominal Theory of Objects with Dependent Types," in *ECOOP 2003 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, L. Cardelli, Ed. Springer Berlin Heidelberg, 2003, pp. 201–224.

[3] N. D. Matsakis and F. S. Klock, II, "The Rust Language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. New York, NY, USA: ACM, 2014, pp. 103–104.

[4] H. Grassmann, *Lehrbuch der Arithmetik für höhere Lehranstalten*. Th. Chr. Fr. Enslin, 1861.

```scala
package nat

import ord._

trait Nat {
  def isZero: Boolean;
  def pred: Nat;
  def succ: Nat;
  def + (that: Nat): Nat = if (that.isZero) this else this.succ + that.pred
  def - (that: Nat): Nat = if (that.isZero) this else this.pred - that.pred
  def * (that: Nat): Nat = if (that.isZero) that else this + this * that.pred
  def ^ (that: Nat): Nat = if (that.isZero) that.succ else this * (this ^ that.pred)
  def <= (that: Nat): Boolean = NatOrdering.leq(this, that)
  def >= (that: Nat): Boolean = NatOrdering.geq(this, that)
  def <  (that: Nat): Boolean = NatOrdering.lt(this, that)
  def >  (that: Nat): Boolean = NatOrdering.gt(this, that)
  def == (that: Nat): Boolean = NatOrdering.eq(this, that)
  def != (that: Nat): Boolean = NatOrdering.neq(this, that)
  def isEven: Boolean = if (this.isZero) true else !this.pred.isEven
  def isOdd: Boolean = !this.isEven
}

object NatOrdering extends Ordering[Nat] {
  def leq (x: Nat, y: Nat): Boolean =
    (x.isZero, y.isZero) match {
      case (true, _) => true
      case (false, true) => false
      case (false, false) => leq(x.pred, y.pred)
    }
}

object Zero extends Nat {
  override def toString () = "Z"
  def isZero = true
  def succ = Succ(this)
  def pred = this
}

case class Succ(n: Nat) extends Nat{
  override def toString () = s"S($n)"
  def isZero = false
  def succ = Succ(this)
  def pred = n
}
```

Listing 6: Complete implementation of the `nat` package.

```scala
package ordlist
import ord._

trait OrdList[+T] {
  def isEmpty: Boolean;
  def insert[U >: T](x: U, o: Ordering[U]): OrdList[U];
}

object Emp extends OrdList[Nothing] {
  override def toString = "Emp"
  def isEmpty = true
  def insert[U >: Nothing](x: U, o: Ordering[U]) =
    Cons(x, this.asInstanceOf[OrdList[U]])
}

case class Cons[+T] (head: T, tail: OrdList[T]) extends OrdList[T] {
  override def toString = s"$head :: $tail"
  def isEmpty = false
  def insert[U >: T](x: U, o: Ordering[U]) =
    if (o.leq(x, head)) Cons(x, this) else Cons(head, tail.insert(x, o))
}
```

Listing 7: Complete implementation of the `ordlist` package, which requires an `Ordering[T]`. Note how `OrdList[T]` is covariant, so `insert` requires argument `x` to be of a supertype of `T`.