Lecture 10

1 Bounded Quantification

We’ve used higher-order functions to write generic sorting functions. Figure 20.1 is a small variation of the kind of function we’ve seen before: instead of taking a comparator function as an argument, it takes a toInt function that maps A-values to integers, then sorts by the natural ordering on integers.

For example, given this type:

```scala
case class Person(name: String, age: Int)
```

We can sort people in ascending order by age:

```scala
val personList: List[Person] = ...
sort((p: Person) => p.age, personList)
```

However, it can be annoying to pass the toInt function around, especially since there is a natural ordering for Persons. We can address this issue by adding a toInt method to Person and modifying insert to invoke this method instead:

```scala
def insert[A](x: A, alist: List[A]): List[A] = alist match {
  case hd :: tl => if (x.toInt <= hd.toInt) ...
  case Nil => ...
}
```

Unfortunately, this code does not type-check: x and hd both have type A, which could be any type. There is no guarantee that A-values have a toInt method.

We can address this problem by introducing a trait for objects that have a ToInt method and modify Person to extend this trait. We could then rewrite the sorting function, as shown in fig. 20.2. Unfortunately, the type of sort is not helpful. When we apply it to a List[Person], we get back a list List[IntLike], and lose track of the fact that we were working with Persons:

```scala
def insert[A](toInt: A => Int, x: A, alist: List[A]): List[A] = alist match {
  case hd :: tl => if (toInt(x) <= toInt(hd)) { x :: hd :: tl } else { hd :: insert(toInt, x tl) }
  case Nil => List(x)
}
```

```scala
def sort[A](toInt: A => Int, alist: List[A]): List[A] = alist match {
  case Nil => Nil
  case hd :: tl => insert(toInt, hd, sort(toInt, sort))
}
```

Figure 20.1: Sorting by mapping values to integers.
trait IntLike {
    def toInt(): Int
}

case class Person(name: String, age: Int) extends IntLike {
    def toInt(): Int = age
}

def insert(x: IntLike, alist: List[IntLike]): List[IntLike] = alist match {
    case Nil => List(x)
    case hd :: tl => if (x.toInt <= hd.toInt) { x :: hd :: tl } else { hd :: insert(x, tl) }
}

def sort(alist: List[IntLike]): List[IntLike] = alist match {
    case Nil => Nil
    case hd :: tl => insert(hd, sort(tl))
}

val alist = List(Person("Alice", 12), Person("Bob", 7), Person("Carol", 3))
val sortedList = sort(alist)
sortedList.head.age // type error, since head has type IntLike

We need to know that the type of the argument and the type of the result of sort are the
same, which is what generics did for us:

def sort[A](alist: List[A]): List[A]

However, the problem with this generic type was that A could be any type, and not necessary
a type with a toInt method, which is what this type ensured:

def sort(alist: List[IntLike]): List[IntLike]

We need to ensure that alist and the result have the same type and that they implement
IntLike. We can do this by using bounded quantification:

def sort[A <: IntLike](alist: List[A]): List[A]

You should read this as “sort consumes and produces lists of A, where A is a subtype of
IntLike”. Figure 20.3 shows a complete version of this code. Note that none of this is
Scala-specific. Figure 20.4 converts this code into Java.

Finally, note that when we write:

def sort[A](alist: List[A]): List[A]

What we’re really saying is that A can be any type, or:

def sort[A <: Any](alist: List[A]): List[A]

The bound Any is implicit. Using bounded quantification, we’re restricting A to be any type
that implements IntLike.

2 Type Parameters to Traits

Traits in Scala (and interfaces in Java) can take type arguments. For example, the following
trait T takes three type arguments:
def insert[A <: IntLike](x: A, alist: List[A]): List[A] = alist match {
  case Nil => List(x)
  case hd :: tl => if (x.toInt <= hd.toInt) { x :: hd :: tl } else { hd :: insert(x, tl) }
}

def sort[A <: IntLike](alist: List[A]): List[A] = alist match {
  case Nil => Nil
  case hd :: tl => insert(hd, sort(tl))
}

Figure 20.3: A well-typed sort using bounded quantification.

trait T[X,Y,Z] {
  def f(x: X): Y
  def g(z: Z): Z
}

Therefore, when a class extends the trait, it needs to supply type-arguments:

class C1 extends T[??? , ??? , ???] {
  def f(x: Int): String = x.toString
  def g(z: Boolean): Boolean = !z
}

We can infer the arguments by examining the body of the class.\footnote{Surprisingly, Scala’s type-inference cannot infer these arguments for us.}

- The class states that the argument of f is an Int and the trait states that the argument of f is an X. Therefore, X = Int.
- The class states that the result of f is a String and the trait states that the result of f is a Y. Therefore, Y = String.
- The class states that the argument and result of g are both Boolean and the trait states that the argument and result of g are both Z. Therefore, Z = Int.

Therefore, the class can extend the trait as follows:

class C1 extends T[Int, String, Boolean] {
  def f(x: Int): String = x.toString
  def g(z: Boolean): Boolean = !z
}

Consider the following class that is trying to extend the same trait:

class C2 extends T[Int, String, ???] {
  def f(x: Int): String = x.toString
  def g(z: Boolean): Int = 42
}

This extension is impossible, since it would imply that Boolean = Int, which is a contradiction.

Consider the following class:

class C3(head: Int, tail: C3) extends T[??? , ??? , ???] {
  def f(x: Int): C3 = new C3(x, this)
  def g(z: C3): C3 = z
}

We can reason through the type parameters in exactly the same way as before:
interface IntLike {
    int toInt();
}

class Person implements IntLike {
    String name;
    int age;
    Person(String name, int age) { this.name = name; this.age = age; }
    public int toInt() { return age; }
}

interface List<T> { }
class Empty<T> implements List<T> { }
class Cons<T> implements List<T> {
    public final T head;
    public final List<T> tail;
    public Cons(T head, List<T> tail) {
        this.head = head;
        this.tail = tail;
    }
}

class Sorting {
    static <T extends IntLike> List<T> insert(T x, List<T> alist) {
        if (alist instanceof Empty) {
            return new Cons<T>(x, new Empty<T>());
        }
        else {
            T hd = ((Cons<T>)alist).head;
            List<T> tl = ((Cons<T>)alist).tail;
            if (x.toInt() < hd.toInt()) {
                return new Cons<T>(x, new Cons<T>(hd, tl));
            }
            else {
                return new Cons<T>(hd, insert(x, tl));
            }
        }
    }
    static <T extends IntLike> List<T> sort(List<T> alist) {
        if (alist instanceof Empty) {
            return alist;
        }
        else {
            T hd = ((Cons<T>)alist).head;
            List<T> tl = ((Cons<T>)alist).tail;
            return insert(hd, sort(tl));
        }
    }
}

Figure 20.4: Figure 20.3 converted into Java (with lists and person too).
• The class states that the argument of f is an Int and the trait states that the argument of f is an X. Therefore, X = Int.

• The class states that the result of f is a C3 and the trait states that the result of f is a Y. Therefore, Y = C3.

• The class states that the argument and result of g are both C3 and the trait states that the argument and result of g are both Z. Therefore, Z = C3.

Here is the complete definition:

```scala
class C3(head: Int, tail: C3) extends T[Int, C3, C3] {
  def f(x: Int): C3 = new C3(x, this)
  def g(z: C3): C3 = z
}
```

Consider the following generalization:

```scala
class C4[A](head: A, tail: C4[A]) extends T[???, ???, ???] {
  def f(x: A): C4[A] = new C4[A](x, this)
  def g(z: C4[A]): C4[A] = z
}
```

We can apply the same reasoning to get:

```scala
class C4[A](head: A, tail: C4[A]) extends T[Int, C4[A], C4[A]] { ... }
```

### 3 Leveraging Recursive Bounded Quantification

It is cumbersome to sort values by explicitly mapping them to integers, which is what we did before. E.g., if we wanted to people sort by name, it would be very annoying to convert all names to unique integers. Alternative, if we want to sort a list of time objects, it would be annoying (and needless) to convert them all to seconds from some fixed date.²

A better approach would be to give sortable classes a lessThan method and then have them inherit from a common trait:

```scala
case class Person(name: String, age: Int) extends Comparable = {
  def lessThan(other: Person): Boolean = this.name < other.name
}
case class Time(h: Int, m: Int, s: Int) extends Comparable = {
  def lessThan(o: Time): Boolean = this.h < o.h || (this.h == o.h && this.m < o.min || (this.m == o.min && this.s < o.s))
}
```

But, defining Comparable is tricky:

```scala
trait Comparable {
  def lessThan(other: ???): Boolean
}
```

The problem is that Person requires the type of the argument to be Person, whereas Time requires it to be Time. However, Comparable should be a generic trait that any type can extend. When a type in a trait must vary, we can simply turn it into a type variable,

```scala
trait Comparable[T] {
  def lessThan(other: T): Boolean
}
```

²Most computers store the current time as the number of seconds elapsed since Jan 1, 1970.
However, the definitions of Person and Time above are now incomplete, since they extend Comparable, which takes a type argument:

```scala
case class Person(name: String, age: Int) extends Comparable[???] = {
  def lessThan(other: Person): Boolean = ...
}
case class Time(h: Int, m: Int, s: Int) extends Comparable[???] = {
  def lessThan(o: Time): Boolean = ...
}
```

We can reason about the types in exactly the way we did before. For example, Person states that the argument of lessThan is Person, whereas the trait states that it is T. Therefore, T = Person:

```scala
case class Person(name: String, age: Int) extends Comparable[Person] = ...
```

We can make a similar argument for Time:

```scala
case class Time(h: Int, m: Int, s: Int) extends ComparableTime = ...
```

Now that both classes extend the Comparable[T] trait, let’s update insert to use it. Here is our first attempt, which uses generics to constrain the type of the argument and result:

```scala
def insert[A](x: A, alist: List[A]): List[A] = alist match {
  case Nil => List(x)
  case head :: tail => if (x.lessThan(head)) { x :: head :: tail } else { head :: insert(x, tail) }
}
```

As before, this code will not type-check, since A could be any type, whereas the body requires A-typed objects to have a lessThan method. Fortunately, we have a trait for these kinds of objects:

```scala
def insert[A <: Comparable](x: A, alist: List[A]): List[A] = ...
```

However, this will not type-check either, because Comparable needs a type argument. So, we really need to write something like this:

```scala
def insert[A <: Comparable[???]](x: A, alist: List[A]): List[A] = ...
```

Let’s reason through this systematically again:

- x has type A <: Comparable[???],
- x.lessThan takes an argument of type ???,
- x.lessThan is applied to head, which has type A,
- Therefore, ??? = A.

Figure 20.5 shows a complete version of sorting using the Comparable trait.

## 4 Expression Evaluator

We’ve seen that we can use case-classes to represent arithmetic expressions and how to write a simple, recursive evaluator. Consider the two evaluators in fig. 20.8. The only difference between them is that fig. 20.6 evaluators integer-valued expressions whereas fig. 20.7 evaluates float-valued expressions. We should be able to abstract away their commonalities by creating a generic type:
trait Comparable[T] {
  def lessThan(other: T): Boolean
}

  case Nil => List(x)
  case head :: tail => if (x.lessThan(head)) { x :: head :: tail } else { head :: insert(x, tail) }
}

  case Nil => Nil
  case head :: tail => insert(x, sort(tail))
}

Figure 20.5: Sorting with a recursive bound.

sealed trait Expr
  case class Const(n: Int) extends Expr
  case class Add(e1: Expr, e2: Expr) extends Expr
  case class Mul(e1: Expr, e2: Expr) extends Expr
  case class Sub(e1: Expr, e2: Expr) extends Expr
  case class Div(e1: Expr, e2: Expr) extends Expr

def eval(expr: Expr): Int = expr match {
  case Const(n) => n
  case Add(e1, e2) => eval(e1) + eval(e2)
  case Mul(e1, e2) => eval(e1) * eval(e2)
  case Sub(e1, e2) => eval(e1) - eval(e2)
  case Div(e1, e2) => eval(e1) / eval(e2)
}

Figure 20.6: Evaluation with Ints.

sealed trait Expr
  case class Const(n: Float) extends Expr
  case class Add(e1: Expr, e2: Expr) extends Expr
  case class Mul(e1: Expr, e2: Expr) extends Expr
  case class Sub(e1: Expr, e2: Expr) extends Expr
  case class Div(e1: Expr, e2: Expr) extends Expr

def eval(expr: Expr): Float = expr match {
  case Const(n) => n
  case Add(e1, e2) => eval(e1) + eval(e2)
  case Mul(e1, e2) => eval(e1) * eval(e2)
  case Sub(e1, e2) => eval(e1) - eval(e2)
  case Div(e1, e2) => eval(e1) / eval(e2)
}

Figure 20.7: Evaluation with Doubles.

Figure 20.8: Two very similar evaluators.
sealed trait Expr[A]
case class Const[A](n: A) extends Expr[A]
case class Add[A](e1: Expr[A], e2: Expr[A]) extends Expr[A]
case class Mul[A](e1: Expr[A], e2: Expr[A]) extends Expr[A]
case class Sub[A](e1: Expr[A], e2: Expr[A]) extends Expr[A]
case class Div[A](e1: Expr[A], e2: Expr[A]) extends Expr[A]

With this type, we can write integer-valued expressions:

```scala
val e1: Expr[Int] = Add(Const(12), Const(3))
```

and float-valued expressions too:

```scala
val e1: Expr[Double] = Div(Const(12), Const(5))
```

The obvious way to generalize `eval` is as follows:

```scala
def eval[A](expr: Expr[A]): A = expr match {
  case Const(n) => n
  case Add(e1, e2) => eval(e1) + eval(e2)
  ...
}
```

Unfortunately, this doesn’t work. The problem is that `eval(e1)` and `eval(e2)` produce values of type A, which could be any type. There is no guarantee that values of this type can be added, multiplied, and so on. For example, we could have written:

```scala
val e1: Expr[String] = Div(Const("Hello"), Const("Goodbye"))
```

The problem is that the type of `eval` is too generic: it should only be applicable to a type with addition, division, etc. defined appropriately. We can define a trait that defines these operations:

```scala
trait NumLike[A] {
  def add(other: A): A
  def mul(other: A): A
  def sub(other: A): A
  def div(other: A): A
}
```

We can create a wrapper for `Int` and `Double` that implements this trait:

```scala
case class N(n: Int) extends NumLike[N] {
  def add(other: N): N = N(n + other.n)
  def mul(other: N): N = N(n * other.n)
  def sub(other: N): N = N(n - other.n)
  def div(other: N): N = N(n / other.n)
}
```

```scala
case class F(x: Double) extends NumLike[F] {
  def add(other: F): N = F(f + other.f)
  def mul(other: F): N = F(f * other.f)
  def sub(other: F): N = F(f - other.f)
  def div(other: F): N = F(f / other.f)
}
```

We can now define the evaluator as follows:

```scala
def eval[T <: NumLike[T]](expr: Expr[T]): T = expr match {
  case Const(n) => n
  case Add(e1, e2) => eval(e1).add(eval(e2))
  case Mul(e1, e2) => eval(e1).mul(eval(e2))
  case Sub(e1, e2) => eval(e1).sub(eval(e2))
  case Div(e1, e2) => eval(e1).div(eval(e2))
}
```

Although we’ve managed to reuse a lot of code in our evaluator, we had to wrap `Ints` and `Doubles`, which was quite annoying. We’ll learn how to address this problem soon.
5 Bounded Quantification in the Java and Scala standard libraries

Bounded quantification is used extensively in libraries. For example, here is the signature of the Integer class in Java:

```java
public final class Integer extends Number implements Comparable<Integer>
```

The standard Comparable interface is very similar to the one we defined.