Type-Directed Programming — Motivation

Effective Programming in Scala

Type-Directed Programming: Intuition

We have seen that the compiler is able to *infer types* from *values*:

```
val x = 42    // x: Int = 42
val y = x + 1 // y: Int = 43
```

The Scala compiler is also able to do the opposite, namely to *infer values* from *types*.

Why is this useful?

Type-Directed Programming: Intuition

We have seen that the compiler is able to *infer types* from *values*:

```
val x = 42    // x: Int = 42
val y = x + 1 // y: Int = 43
```

The Scala compiler is also able to do the opposite, namely to *infer values* from *types*.

Why is this useful?

When there is exactly one "obvious" value for a type, the compiler can provide that value to us.

Let us look at a motivating example.

Sorting Lists of Numbers

Consider a method `sort` that takes as parameter a `List[Int]` and returns another `List[Int]` containing the same elements, but sorted:

```
def sort(xs: List[Int]): List[Int] =
  ...
  ... if x < y then ...
  ...
```

At some point, this method has to compare two elements `x` and `y` of the given list.

Making `sort` more General (1)

Problem: How to generalize `sort` so that it can also be used for lists with elements other than `Int`, such as `Double` or `String`?

A first step is to use a **type parameter** `A` for the type of elements:

```
def sort[A](xs: List[A]): List[A]
```

Now, we can call this method with an arbitrary type of list, such as `List[Int]`, `List[Double]`, or `List[String]`.

If we call `sort` with a `List[Int]` as parameter, the compiler unifies the type parameter `A` with `Int`, so the return type is `List[Int]`.

Making sort more General (2)

```
def sort[A](xs: List[A]): List[A] =
  ...
  ... if x < y then ...
  ...
```

Unfortunately, we can't implement this method because there is no universal way of comparing values of type A!

Parameterization of `sort`

One solution is to pass the comparison operation as an additional parameter:

```
def sort[A](xs: List[A])(lessThan: (A, A) => Boolean): List[A] =
  ...
  ... if lessThan(x, y) then ...
  ...
```

Calling Parameterized sort

We can now call sort as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val strings = List("apple", "pear", "orange", "pineapple")

sort(xs)((x, y) => x < y)
// : List[Int] = List(-5, 2, 3, 6, 7)

sort(strings)((s1, s2) => s1.compareTo(s2) < 0)
// : List[String] = List(apple, orange, pear, pineapple)
```

Parameterization with Ordering

There is already a class in the standard library that represents orderings:

`scala.math.Ordering[A]`

Provides ways to compare elements of type `A`. So, instead of parameterizing with the `lessThan` function, we could equivalently parameterize with `Ordering`:

```
def sort[A](xs: List[A])(ord: Ordering[A]): List[A] =
  ...
  ... if ord.lt(x, y) then ...
  ...
```

Ordering Instances

Calling the new `sort` can be done like this:

```
import scala.math.Ordering

sort(xs)(Ordering.Int)
sort(strings)(Ordering.String)
```

This makes use of the values `Int` and `String` defined in the `scala.math.Ordering` object, which produce the right orderings on integers and strings.

```
object Ordering:
  val Int = new Ordering[Int]:
    def compare(x: Int, y: Int) =
      if x < y then -1 else if x > y then 1 else 0
```

Reducing Boilerplate

Problem: Passing around `Ordering` arguments is cumbersome.

```
sort(xs)(Ordering.Int)
sort(ys)(Ordering.Int)
sort(strings)(Ordering.String)
```

Sorting a `List[Int]` value always uses the same `Ordering.Int` argument, sorting a `List[String]` value always uses the same `Ordering.String` argument, and so on...

Simple cases are repetitive and tedious. Complex cases (e.g., sorting by multiple criteria) are painful to maintain. Can the compiler automate this process for us?