Updated: 2021-08-31

6.4
Type Inference
Making the compiler write your code

Northeastern
University

Type Inference

- Type Inference…
  - is what allows the left-hand-side or the right-hand-side of an “=“ or “=>” determine the type of an identifier;
  - actually, with =>, it’s only the RHS typically that can infer type.
- So, when we write:

  val x = 1

  - The compiler is able to deduce that the type of *x* is *Int*.
- Similarly, when we write:

  val xs: List[Double] = List(1, 2, 3)

  - The compiler is able to deduce that the type of *List* required on the right-hand-side is a *List[Double]*, even though the elements appear to be *Int*s.

Extensions of this idea

- Suppose we define a case class:

  case class Complex(real: Double, imag: Double)

- And, further let's suppose that we have method:

  def process2[P1, P2, T :< Product](f: (P1, P2) => T): Processor[T]

- Where *Processor[T]* is some trait…
  - We can create a processor using the *process2* method, even though we may never actually invoke *f*.
  - But the compiler *uses* the actual provided value of *f* to determine the underlying type or the resulting *Processor*.
  - And the function *f* will almost always come directly from the *apply* method of the case class's companion object.
- Example: *jsonFormat2* in Poet.scala

Let’s look at an example

```
def comparer2[P0:Comparer, P1:Comparer, T<:Product](f: (P0,P1)=>T): Comparer[T]
     = comparer[T, P0](0) orElse comparer[T, P1](1)
```

- This method *comparer2* builds a *Comparer[T]* where *T* is a case class or tuple (because it is constrained to be a sub-class of *Product*) with **two** parameters, of types *P0* and *P1*.
- The resulting *Comparer[T]* acts by first comparing the 0th (i.e. first) parameters of any two *T* objects, and if they are the same, it will compare their 1st (i.e. second) parameters.
- How does it know how to compare the *P0* values or the *P1* values?
- Because of the context bounds specifying that there must be an implicit parameter of *comparer2* of type *Comparer[P0]* and another of type *Comparer[P1]*.
- In this example, the method *f* is never invoked at all! It is only there for type inference.

Example of use

```
case class Composite(i: Int, s: String)
object MyComparers extends Comparers {
  val comparer: Comparer[Composite] = comparer2(Composite.apply)
}
import MyComparers._
val c1a = Composite(1, "a")
val c2a = Composite(2, "a")
val c1z = Composite(1, “z")
comparer(c1z)(c1a) shouldBe Less
comparer(c2a)(c1a) shouldBe Less
comparer(c2a)(c1z) shouldBe Less
comparer(c1a)(c1a) shouldBe Same
comparer(c1a)(c2a) shouldBe More
comparer(c1a)(c1z) shouldBe More
```

- In this specification, we need to write *Composite.apply*, not just *Composite* because there is an explicit companion object to Composite. Otherwise, we could drop the “.apply” part of the parameter.
- There are implicit *Comparer[Int]* and *Comparer[String]* values defined in the companion object of *Comparer*.

An alternative formulation

```
case class Composite(i: Int, s: String)
object Composite {
  implicit val comparer: Comparer[Composite] = Comparer.same[Composite] :| (_.s) :| (_.i)
}
it should "implement Compare" in {
  // NOTE: this uses the implicit val Composite.comparer
  val c1a = Composite(1, "a")
  val c2a = Composite(2, "a")
  val c1z = Composite(1, "z")
  Compare(c1a, c1z) shouldBe Less
  Compare(c1a, c2a) shouldBe Less
  Compare(c1z, c2a) shouldBe More
  Compare(c1a, c1a) shouldBe Same
  Compare(c2a, c1a) shouldBe More
  Compare(c1z, c1a) shouldBe More
}
```

Lens functions

- In the companion object of *Composite*, we define an implicit *Comparer[Composite]* which compares the *s* field first, then the *i* field.
- Its definition is based on the composition of “lens” functions to build a *Comparer* for a *Composite*.