Dependent Types in Scala (draft)

Yao Li

Static type systems are the world's most successful application of formal methods. Types are simple enough to make sense to programmers; they are tractable enough to be machine-checked on every compilation; they carry no run-time overhead; and they pluck a harvest of low-hanging fruit.

– Brent A. Yorgey, et al. *Giving Haskell a Promotion.*

Introduction

Dependent types can help prevent bugs and guide programmers to construct correct implementations, by enabling using extremely expressive types as specifications.

For example, in a language with full dependent type support like Gallina, you can write something like this:

```
Fixpoint rep (A: Type) (n: nat) (a: A): Vector A n:=
    match n with
    | O => VNil A
    | S n' => VCons A n' a (rep A n' a)
    end.
```

where `Vector A n` is a vector of `A` and its length is `n`. In this program, the return type of the `rep` function depends on a value, so it is called a dependent type. This program would not type check if the function implementation returns a list whose length is not `n`, hence bogus implementations are rejected.

Unfortunately, the above example is not directly available in most mainstream programming languages including Scala. The reason is that these languages usually enforce a phase separation between runtime values and compile-time values.

Nevertheless, I will show you that by using a trick called singleton types and features from Scala such as subtyping and path dependent types, we can encode the above example in Scala, and even more.

More specifically, in this article, we will build a vector in Scala whose length information is encoded in its type. With the information of its length at type-level, we can define the following operations:

- a `rep` function which takes a number, and returns a `Vector` of exactly that length.
- an `app` function that takes two vectors, and returns a list whose length is the sum of their lengths.
- an indexing method with compile-time bounds checking.

Putting Numbers in Your Types

A vector's length ranges from 0 to infinite, so natural numbers are clearly the best representation for it. How do we encode natural numbers in Scala?

```scala
type Nat = Int
```

This is not ideal because we would allow negative numbers to inhabit the type of natural numbers. In this article, we particularly care about the precision of types, so we will not do that.

There are many ways to define natural numbers, here we are particularly interested in the following simple inductive definition:

1. 0 is a natural number.
2. if n is a natural number, then $S(n)$ is also a natural number, where $S(n)$ means the successor of n (later, we can prove that $S(n) = n + 1$).

We are interested in the above inductive definition because we can easily encode an inductive definition in Scala using subtyping:

```scala
trait Nat

case object Z extends Nat

case class S(n: Nat) extends Nat
```

We use `Z` to denote zero, and `S` to denote successor.

Can we use the natural numbers we have just defined in the type of vector? The answer is clearly no. We cannot just use these runtime values as type parameters that are used at compile time in Scala. We have to find a method to represent numbers at type level. Here we use a technique called singleton types.

Singleton Types

Singleton types are types of those each has exactly one inhabitant. If we can define one type for each natural number, and

1. be able to get the type from its value
2. and recover the value from its type,

it seems to be equivalent to having numbers at both type- and value-level.

In our above encoding of natural numbers, `Z` is a value who has a singleton type `Z.type`, because that's what `case object` means in Scala. However, `S(Z)` (i.e. 1) and `S(S(Z))` (i.e. 2) do not have singleton types associated with them, they share a common type `S` instead.

Here is a silly way to define singleton types for natural numbers:

```
trait Nat

case object zero extends Nat

case object one extends Nat

case object two extends Nat
```

With this encoding, we do get singleton types for natural numbers 0, 1, and 2, but there are infinitely many natural numbers, and it's probably unwise to invest a mortal's life in defining the infinite amount of natural numbers in this way.

It turns out that we can change only a little bit of our inductive definitions of natural numbers above to get singleton types:

```
sealed trait Nat

case object Z extends Nat

case class S[N <: Nat](n: N) extends Nat
```

How is this a definition of singleton types for natural numbers? We can show that again by induction:

1. `Z.type` is a singleton type with only one inhabitant: `Z`.
2. If `N` is a singleton type, `S[N]` must also be a singleton type because there is exactly one way to construct an inhabitant of it, that is, from the sole inhabitant of `N`.

We also add the `sealed` keyword in front of the `Nat` trait so it can only be inherited by classes defined in the same file. This prevents others from inhabiting more values in our types.

We can now open a Scala REPL and play with our natural numbers:

```
scala> Z
res1: Z.type = Z

scala> S(Z)
res2: S[Z.type] = S(Z)

scala> S(S(Z))
res3: S[S[Z.type]] = S(S(Z))
```

It seems that we have indeed associated a singleton type with each value. However, I did mention that we should also be able to recover a value from a singleton type, right? Unfortunatly, we cannot do that yet with the code we have.

Let's think for a minute how can we recover a value from a singleton type:

- If the type is `Z.type`, we just return `Z`. This seems easy.
- If the type is `S[N]`, we recover the value `n` of type `N`, and then return `S(n)`. How do we recover the value of type `N`? The same process.

The algorithm seems easy but here are two big problems: we need (1) pattern matching and (2) recursive calls *at the type level*!

This sounds like a very difficult task, if not impossible, in most programming languages, but Scala has one feature that allows us to do exactly that: *implicits*.

I will not go into details to explain what Scala implicits is because it is a complex concept that I myself do not understand all of it[1]. Instead, I will just show you the code we need and explain what happens there:

```
object Nat {
  def get[N <: Nat](implicit v: N): N = v

  implicit val get_z: Z.type = Z
  implicit def get_s[N <: Nat](implicit n: N): S[N] = S(n)
}
```

We can use the above code in a Scala REPL:

[1]If you want to know more about Scala implicits, I would recommend you to read a whole lot of blog posts on the Internet about it. And if you manage to understand the implicit scope, I would be extremely interested to hear about it!

```
scala> Nat.get[S[S[S[Z.type]]]]
res1: S[S[S[Z.type]]] = S(S(S(Z)))
```

Wait, the `get` method is defined with one parameter, how did we manage to call it and get a result without even passing in an argument?

Here's how it works: A parameter of a method in Scala can be declared to be implicit. If so, the programmers will not need to provide a value for this parameter, as long as the compiler can find an implicit value for it according to its type. This may at first sound like just a sugar syntax, but let's see what this does for us:

Our definition of `get` method is very simple here: it simply asks the compiler to find a value of type `N` for us and returns it. But how would the Scala compiler find a value of type `N`? Let's consider two possible cases:

- If the type is `Z.type`, the compiler finds that there is a implicit value `get_z` of type `Z.type` in scope, so it will just fill in `get_z` as the implicit parameter of `get`.
- If the type is `S[N]` for some type `N`, the compiler finds that there is an implicit function that returns a value of type `S[N]`. However, this function requires another implicit parameter `n` of type `N`. How does the compiler find this implicit value of type `N`? Again, by checking the type of `N` and then trying to find an implicit definition from either `get_z` or `get_s`. The Scala compiler will try to find the implicit value recursively!

As you can see, Scala implicits make the Scala compiler do pattern matchings and recursive calls automatically for us at the type level!

Vectors

Now we have numbers at the type level, we can define our vectors. How can we define such a vector? Let's again consider the following inductive definition (for simplicity, let's just consider of the vector of integers):

1. Nil is a vector (more precisely, an empty vector).
2. If v is a vector and x is an arbitrary integer, $\text{cons}(x, v)$ is also a vector.

We can encode the above definition easily using subtyping, but let's also add the length information in addition to that:

```
sealed trait Vec[N <: Nat]

case object Nil extends Vec[Z.type]

case class Cons[N <: Nat](h: Int, t: Vec[N]) extends Vec[S[N]]
```

These definitions above should be fairly straightforward except for one caveat: `Cons[N]` should extend `Vec[S[N]]` instead of `Vec[N]`. This makes sense: the vector's length must be at least 1 once we have used `Cons`. Another way to look at this is that `Cons[N]` means cons a value with a vector of length `N`.

We can play with our definition in the Scala REPL:

```
scala> Nil
res0: Nil.type = Nil

scala> Cons(1, Cons(2, Cons(3, Nil)))
res1: Cons[S[S[Z.type]]] = Cons(1,Cons(2,Cons(3,Nil)))
```

Notice that the type of `Cons(1, Cons(2, Cons(3, Nil)))` is `Cons[S[S[Z.type]]]`, which really is `Vec[S[S[S[Z.type]]]]`. We can also show this fact in the Scala REPL:

```
scala> val l: Vec[S[S[S[Z.type]]]] = Cons(1, Cons(2, Cons(3, Nil)))
l: Vec[S[S[S[Z.type]]]] = Cons(1,Cons(2,Cons(3,Nil)))
```

Replication

Now it's the time to get back to the example we have shown in the introduction: let's define `rep` function which takes a number `n`, and an integer `x`, and returns a vector of `x` whose length is exactly `n`.

Our first problem is that, since the length `n` is given as a function parameter, we must propogate it to the type level. This is easy: just define the signature of our `rep` function as follows:

```
object Vec {
  def rep[N <: Nat](n: N, x: Int): Vec[N] = ???
}
```

It looks like the information of number `n` is repeated twice here. But don't worry. When calling this function, we don't need to provide the type parameter because the type inference algorithm of Scala will do this for us[2].

Our next step is to implement this function. The most intuitive approach is shown below:

[2]Some articles on how the type inference algorithm works may claim that the type information flows from left to right in an expression. That is wrong. Scala's type inference algorithm is based on a technique called bidirectional type checking that was first described by Pierce and Turner [1998], and later adapted by Odersky *et al.* [2001].

```
object Vec {
  def rep[N <: Nat](n: N, x: Int): Vec[N] = n match {
    case Z => Nil
    case S(p) => Cons(x, rep(p, x))
  }
}
```

Unfortunately, the above code snippet would not type check. Here are the errors that compiler would report to us:

```
[error] xxx.scala:24: type mismatch;
[error]  found   : Nil.type
[error]  required: Vec[N]
[error] Note: Z.type <: N (and Nil.type <: Vec[Z.type]),
        but trait Vec is invariant in type N.
[error] You may wish to define N as +N instead. (SLS 4.5)
[error]     case Z => Nil
[error]               ^
[error] xxx.scala:25: type mismatch;
[error]  found   : Cons[Nat]
[error]  required: Vec[N]
[error]     case S(p) => Cons(x, rep(p, x))
[error]                      ^
[error] two errors found
```

To understand these errors, we need a little bit background in how type inference algorithms usually work with polymorphic types: they walk through the whole program or parts of the code[3] and generate some type constraints (which are the relations between types) along the way, and then try to solve those constraints to figure out the exact types.

Let's try to imagine how type inference works by manually walking through the cases inside the pattern matching in our code, we will be able to know two things about the value's type `N`:

1. when `n` is `Z`, `Z.type` is a subtype of `N`.
2. when `n` is `S(p)` for some value `p`, its type `S[M]` is a subtype of `N` for some type `M`.

We will only provide a value to parameter `n` that either has type `Z.type` or `S[M]` for some type `M`. That is, we will never use a super class of them such as `Nat`. However, there is no way to inform the Scala compiler of that. Therefore, the

[3]Scala employs a local type inference algorithm, so it will not walk through the entire program before it tries to solve the constraints. This is because whole-program based type inference algorithms for systems with subtyping are usually too slow to be practical.

best the Scala compiler can do is to infer that when `n` is `Z`, `Z.type` is a subtype of `N`. There is no way to infer if that `Z.type = N`. And because our `Vec` is invariant in its type parameter `N` [4], `Vec[Z.type]` is not a subtype of `Vec[N]` and hence types do not match.

There is an easy fix to this problem: we just define `Vec` to be covariant in its type parameter `N`. However, this is not ideal because we are not precise about what is going on in our types. And even with this fix, the program will not type check because the bigger problem lies in the second error the Scala compiler has reported to us.

We know that when `n` is `S(p)` for some value `p`, there is a type `M` such that `S[M]` is the type of `S(p)` and a subtype of `N`, but Scala's local type inference algorithm will not try to create a new type variable `M` here. It will, instead, try to be greedy to solve all the constraints locally and get an optimal solution for that type. In this case, that type is `Nat`. This is clearly not what we want.

Can we enforce the Scala compiler to create a new type variable inside pattern matching? I don't know that answer, but let's think about our problem again: what do we want here? We want to do pattern matchings and recursive calls with a full awareness of the type-level information. Does that sound familiar? Yes. That's just like what we need for our `get` function of `Nat`.

We again employ Scala implicit to write our `rep` function. A naive approach could be like this:

```
object Vec {
  def rep[N <: Nat](n: N, x: Int)(implicit v: Vec[N]) = v

  implicit val rep_z: Vec[Z.type] = Nil
  implicit def rep_s[N <: Nat](x: Int)(implicit v: Vec[N]): Vec[S[N]] =
    Cons(x, v)
}
```

This would type check but it would not give us what we want. Why? Take a look at our `rep_s` function again, it is function that has an explicit parameter as well as an implicit parameter. The Scala compiler will be able to fill in the implicit parameters of a function automatically, but it will not try to fill in the explicit ones.

What do we do here? Well, we just get rid of the explicit parameter. We can use some functional thinking to get arond this problem: the trick is instead of

[4]Type `Vec` is invariant in type `N` means that no matter what subtyping relations hold for two types `S` and `T`, there is no subtyping relation between `Vec[S]` and `Vec[T]`. Two other related concepts are covariance and contravariance: covariance (declared by defining `Vec` as `sealed trait Vec[+N <: Nat]` means that `S <: T` implies `Vec[S] <: Vec[T]`, while contravariance (declared by defining `Vec` as `sealed trait Vec[-N <: Nat]`) means the opposite: `S <: T` implies `Vec[T] <: Vec[S]`.

returning a implicit definition of type `Vec`, we return a function of type `Int => Vec`:

```
object Vec {
  def rep[N <: Nat](n: N, x: Int)(implicit f: Int => Vec[N]) = f(x)

  implicit val rep_z: Int => Vec[Z.type] = (_: Int) => Nil
  implicit def rep_s[N <: Nat](implicit f: Int => Vec[N]):
      Int => Vec[S[N]] = (x: Int) => Cons(x, f(x))
}
```

And if we run it in a Scala REPL:

```
scala> Vec.rep(S(S(Z)), 0)
res0: Vec[S[S[Z.type]]] = Cons(0,Cons(0,Nil))

scala> Vec.rep(Z, 1)
res1: Vec[Z.type] = Nil

scala> Vec.rep(S(S(S(Z))), 42)
res2: Vec[S[S[S[Z.type]]]] = Cons(42,Cons(42,Cons(42,Nil)))
```

We can also play with our implementation. For example, if we change `rep_s` to:

```
implicit def rep_s[N <: Nat](implicit f: Int => Vec[N]):
    Int => Vec[S[N]] = (x: Int) => f(x)
```

A type error will be reported:

```
[error] xxx.scala:27: type mismatch;
[error]  found   : Vec[N]
[error]  required: Vec[S[N]]
[error]       Int => Vec[S[N]] = (x: Int) => f(x)
[error]                                       ^
[error] one error found
```

Indeed, we need to return a vector of length `S[N]` but our implementation returns a vector of length `N`. The type checker finds a bug for us!

Append

Now let's think about a more challenging problem: how to implement an `app` method for our `Vec`?

What would the type signature of our `app` method be like?

```scala
sealed trait Vec[N <: Nat] {
  def app[M <: Nat](b: Vec[M]): Vec[N + M]
}
```

Defining this method is much more tricky because we need to talk about the sum of two natural numbers at the type level. Or in other words, encode a function in types.

Before we try to solve this problem, let's first think about how the plus operation on natural numbers is defined. Here is one simple way of defining $n + m$:

1. If $n = 0$, then $n + m = m$.
2. If $n = S(n')$ for some n', then $n + m = S(n' + m)$.

The second case has a recursive call. Because n is finite, we can eventually reduce any n to 0 and execute on the base case.

This definition is recursive and depends on the value of n. At the type level, this means we need pattern matchings and recursive definitions on types. You may be tempted to use Scala implicits, but that is used to construct function calls that will be executed at runtime. What we need here is a "function" that runs purely at compile-time.

Can we do that? Well, it happens that there is a feature in Scala that allows us to do exactly that: path-dependent types. Let me first show you the code and then try to explain this concept to you:

```scala
import scala.language.higherKinds

sealed trait Nat {
  type :+[M <: Nat] <: Nat
}

case object Z extends Nat {
  type :+[M <: Nat] = M
}

case class S[N <: Nat](n: N) extends Nat {
  type :+[M <: Nat] = S[n.:+[M]]
}
```

Scala allows us to declare a type in a trait, without giving a specific definition, and each subclass of that trait can give a different definition to it. In this way, the actual definition of this type depends on which subclass we are using, so it is called a path-dependent type.

In this example, we say `:+` is a type that should be defined by each subclass of `Nat`. The concrete type can be anything, as long as it satisfies the type refinement we have defined in `Nat`, that is, it should be a subtype of `Nat` [5]. Notice that I define the plus operation using the symbol `:+` to distinguish it from a value-level plus operation.

The definitions of `:+` can be given by the subclasses of `Nat`, according to our definition of the plus operation. That is, in the case of `Z.type`, returns `M`, the other type parameter; in the case of `S[N]` for some type `N`, we do a recursive call using `n.:+[M]`, and then wrap the result with `S`.

To demonstrate how to use this `:+` type, we also define the plus operation as a run-time method:

```
sealed trait Nat {
  type :+[M <: Nat] <: Nat
  def +[M <: Nat](m: M): :+[M]
}

case object Z extends Nat {
  type :+[M <: Nat] = M
  def +[M <: Nat](m: M): :+[M] = m
}

case class S[N <: Nat](n: N) extends Nat {
  type :+[M <: Nat] = S[n.:+[M]]
  def +[M <: Nat](m: M): :+[M] = S(n + m)
}
```

We are almost there to be able to define an `app` method for vectors. Here's still one small problem: `:+` is a definition on an instance, not a class. This means that we cannot just call `N.:+[M]`. We will need to get an instance `n` of type `N`, and then call `n.:+[M]`. But we already know how to do that, right? Here's what we are going to write down:

```
sealed trait Vec[N <: Nat] {
  def app[M <: Nat](v: Vec[M])(implicit n: N): Vec[n.:+[M]]
}

case object Nil extends Vec[Z.type] {
  def app[M <: Nat](v: Vec[M])(implicit n: Z.type): Vec[n.:+[M]] = v
```

[5]Some reader may realize that I am being a little inprecise here because I don't want to confuse others with the concept of kinds. `:+` is not a type but really a type constructor that takes a type and returns another type. Think about `List`, it is not a type. `List[Int]` is. You need to provide concrete type (e.g. `Int`) to `List` to construct another concrete type (e.g. `List[Int]`). It is similar in our case of `:+`. However, for convenience, I will not try to distinguish them in this article.

```
}

case class Cons[N <: Nat](h: Int, t: Vec[N]) extends Vec[S[N]] {
  def app[M <: Nat](v: Vec[M])(implicit n: S[N]): Vec[n.:+[M]] =
    Cons(h, t.app(v)(n.n))
}
```

Notice that we cannot just write `Cons(h, t.app(v))` in `Cons.app`. The Scala compiler will not be able to find an implicit value of type `N` this time, because there is not enough information to help the compiler to choose from `get_z` or `get_s`. Fortunately, *we know* which value to pass to that parameter, so we just pass in that value explicitly.

Indexing

It would be no point of encoding the length information to a vector, if we do not have bounds checking for indexing. Let's implement that.

Our first step is to define a less than relation between natural numbers. First, let's try to define it mathematically:

1. $0 < S(n)$ for all n.
2. If $n < m$, then $S(n) < S(m)$.

Again, we can encode a inductive definition at type level using subtyping:

```
sealed trait Lt[N <: Nat, M <: Nat]

case class LtZ[M <: Nat]() extends Lt[Z.type, S[M]]

case class LtS[N <: Nat, M <: Nat](lt: Lt[N, M])
    extends Lt[S[N], S[M]]
```

How can we use these classes? Let's look at them in this way: an instance of `Lt[N, M]` is a *proof* that `N` is less than `M`. Everytime we access an index in a vector, we ask the programmer to give a proof that the index is less than the length of this vector. When the index is greater than or equal to the length of the vector, there is no proof and hence no possible value to pass to the method. Therefore, the method can *only* be executed when users *can* provide a proof.

A first attemp may look like this:

```
sealed trait Vec[N <: Nat] {
  def app[M <: Nat](v: Vec[M])(implicit n: N): Vec[n.:+[M]]
  def apply[M <: Nat](m: M)(proof: Lt[M, N]): Int
}

case object Nil extends Vec[Z.type] {
  def app[M <: Nat](v: Vec[M])(implicit n: Z.type): Vec[n.:+[M]] = v
  def apply[M <: Nat](m: M)(proof: Lt[M, Z.type]): Int =
    throw new RuntimeException("This will never happen!")
}

case class Cons[N <: Nat](h: Int, t: Vec[N]) extends Vec[S[N]] {
  def app[M <: Nat](v: Vec[M])(implicit n: S[N]): Vec[n.:+[M]] =
    Cons(h, t.app(v)(n.n))
  def apply[M <: Nat](m: M)(proof: Lt[M, S[N]]): Int =
    m match {
      case Z => h
      case S(p) => ??? // what to do here?
    }
}
```

We stuck in the case of `Cons.apply`. The problem is that when `m` is in the form of `S(p)`, we do not know how to find a proof that the type of `p` is less than `N`?

However, we don't need to do a pattern matching on `m`. We can, instead, do that on the proof! Our definition of less than relations has already contained the fact whether `m` is zero. Furthermore, the proof that $S(a) < S(b)$ contains the proof that $a < b$!

Now we can get our indexing method working:

```
sealed trait Vec[N <: Nat] {
  def app[M <: Nat](v: Vec[M])(implicit n: N): Vec[n.:+[M]]
  def apply[M <: Nat](m: M)(proof: Lt[M, N]): Int
}

case object Nil extends Vec[Z.type] {
  def app[M <: Nat](v: Vec[M])(implicit n: Z.type): Vec[n.:+[M]] = v
  def apply[M <: Nat](m: M)(proof: Lt[M, Z.type]): Int =
    throw new RuntimeException("This will never happen!")
}

case class Cons[N <: Nat](h: Int, t: Vec[N]) extends Vec[S[N]] {
  def app[M <: Nat](v: Vec[M])(implicit n: S[N]): Vec[n.:+[M]] =
    Cons(h, t.app(v)(n.n))
  def apply[M <: Nat](m: M)(proof: Lt[M, S[N]]): Int =
    proof match {
      case LtZ() => h
      case LtS(p) => t.apply(m.n)(p)
    }
}
```

However, it is a quite tedious to provide a proof for the indexing. And it sometimes requires several proofs to construct one proof. The process of constructing a proof is also quite mechanical: it's just pattern matching on the first parameter and some recursive calls at the type level. If only we can automate this!

Well, we can. And we have seen how to do that several times when walking through this article, right? Again, Scala implicits to the rescue!

Eventually, our code for this part looks like this:

```
sealed trait Lt[N <: Nat, M <: Nat]

case class LtZ[M <: Nat]() extends Lt[Z.type, S[M]]

case class LtS[N <: Nat, M <: Nat](lt: Lt[N, M])
    extends Lt[S[N], S[M]]

object Lt {
  implicit def get_ltz[M <: Nat]: Lt[Z.type, S[M]] = LtZ()
  implicit def get_lts[N <: Nat, M <: Nat](implicit lt: Lt[N, M]):
      Lt[S[N], S[M]] = LtS(lt)
}

sealed trait Vec[N <: Nat] {
  def app[M <: Nat](v: Vec[M])(implicit n: N): Vec[n.:+[M]]
  def apply[M <: Nat](m: M)(implicit proof: Lt[M, N]): Int
}

case object Nil extends Vec[Z.type] {
  def app[M <: Nat](v: Vec[M])(implicit n: Z.type): Vec[n.:+[M]] = v
  def apply[M <: Nat](m: M)(implicit proof: Lt[M, Z.type]): Int =
    throw new RuntimeException("This will never happen!")
}

case class Cons[N <: Nat](h: Int, t: Vec[N]) extends Vec[S[N]] {
  def app[M <: Nat](v: Vec[M])(implicit n: S[N]): Vec[n.:+[M]] =
    Cons(h, t.app(v)(n.n))
  def apply[M <: Nat](m: M)(implicit proof: Lt[M, S[N]]): Int =
    proof match {
      case LtZ() => h
      case LtS(p) => t.apply(m.n)(p)
    }
}
```

Try this in our Scala REPL:

```
scala> val l = Cons(1, Cons(2, Cons(3, Nil)))
l: Cons[S[S[Z.type]]] = Cons(1,Cons(2,Cons(3,Nil)))

scala> l(Z)
res1: Int = 1

scala> l(S(Z))
res2: Int = 2

scala> l(S(S(S(Z))))
<console>:16: error: could not find implicit value for parameter proof:
              Lt[S[S[S[Z.type]]],S[S[S[Z.type]]]]
       l(S(S(S(Z))))
        ^
```

Notice that the last error is a compile-time error. The compiler complaints to us that it cannot find a proof to show that $3 < 3$. Indeed, there is no such proof!

Further Reading

I want to conclude this article by listing a few reading materials.

To be filled.

References

Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored Local Type Inference. *POPL*, pages 41–53, 2001.

Benjamin C Pierce and David N Turner. Local Type Inference. *POPL*, pages 252–265, 1998.