Dependent Types in Scala

Yao Li

December 18, 2016

Abstract

Dependent types can prevent bugs and guide programmers to construct correct implementations, by enabling using extremely expressive type system as specifications. While well-known languages with dependent types are Agda, Gallina, Idris, etc., [Eisenberg and Weirich, 2013] have shown that one can also write dependently typed programs in a main-stream functional programming language, *i.e.*, Haskell, by using singleton types.

This report is about my exploration of dependent types in another main-stream programming language, Scala. In contrast with Haskell, Scala is a language with a core based on object-oriented programming. As we will see in this report, this makes a huge difference between the way of implementing dependent types in Scala and in Haskell.

This report consists of two parts: Part 1 shows how to write dependently typed programs in Scala, by using techniques such as subtyping, parametric polymorphism, path dependent types, etc.; In Part 2, I try to talk about how to understand why we can use these completely different techniques comparing with those have been used in Haskell, to implement the same thing.

1 Introduction

A dependent type is a type whose definition depends on a value. For example, in Coq (which has dependent types), one can write down the following function definition:

```
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` represents a vector which contains exactly `n` elements of type `A` (its definition in Coq is omitted). Notice that the return type of this function, *i.e.*, `Vector A n`, depends on one of the values of its parameters, *i.e.*, `n`.

Now, if we change our second case in above the definition to `VCons A O a (rep A O a)`, the Coq compiler will reject this program because the type of

this expression, `Vector A (S 0)`, does not conform the expected type, `Vector A n`. Indeed, a vector of length 1 is different from a vector of length `n`. The compiler rejects the program because it is wrong! Here, dependent types act as an expressive specification of a program, therefore it can prevent bugs and guide programmers to construct the correct implementations.

Unfortunately, dependent types are not directly supported in most mainstream programming languages. For example, the above function definitions cannot be directly written in Haskell, because Haskell (and many other programming languages) enforces a phase separation between runtime values and compile-time values.

However, it has been shown that any constraint expressible with dependent types, can be expressed in a system which does not support dependent types, by using singleton types [Monnier and Haguenauer, 2010]. Using singleton types to write dependently typed programs has been explored in Haskell [Eisenberg and Weirich, 2013; Eisenberg, 2016]. In this report, I try to explore the same techniques in another programming language, Scala.

In contrast with Haskell, Scala is a language with a core based on object-oriented programming, even though it adopts a lot of ideas from the world of functional programming. As we will see in this report, this makes a huge difference between the way of encoding singleton types and implementing dependent types in these two languages.

This report consists of two parts. In Part 1 (Section 2, Section 3, and Section 4), I will talk about how to write dependently typed programs in Scala, by using techniques such as subtyping, parametric polymorphism, path dependent types, etc. In Part 2 (Section 5), I will talk about how to understand the fact that we have used completely different features to implement the same thing in two languages.

2 Basics for Functional Programming in Scala

Scala is a language whose core is based on object-oriented programming. However, it adopts many ideas from the world of functional programming. In this section, we discuss the *alter ego*s of some basic functional programming concepts, namely, algebraic data types and generic algebraic data types (or GADTs), in Scala.

2.1 Product Types

One natural way to represent product types in Scala is using tuples:

```
val book = ("Practical Foundations for Programming Languages",
            "Robert Harper")
```

However, there are two problems with tuples:

1. There can only be at most 22 elements in a tuple.

2. Two tuples with the same structure would have the same type. This is not ideal if you want to, say, distinguish between a Cartesian coordinate and a polar coordinate.

Another way is to use case classes:

```
case class Cartesian(x: Double, y: Double)
case class Polar(r: Double, t: Double)
```

Case classes in Scala are regular classes, but immutable by default and, more importantly, can be decomposed using pattern matching. We will see that in the next section.

We can also define a nullary prodcut by using case objects:

```
case object Empty
```

Case objects are similar to case classes except that it does not take any argument (i.e., a nullary product).

2.2 Sum Types

Sum types can be simulated using subtyping in Scala. The following code snippet describes a binary tree:

```
sealed trait Tree
case object Empty extends Tree
case class Node(l: Tree, v: Int, r: Tree) extends Tree
```

A `trait` in Scala is very similar to an `interface` in Java. That is to say, abstract methods can be defined in it, and a class can inherit from several traits. However, concrete methods can also be defined inside a `trait`, in contrast with Java `interface`.

The `sealed` keyword used in the above code snippet means that this trait can only be extended in the same file where it has been defined.

Here we used a Scala `trait` to define a base class, and then define two case classes which inherit from it. Now, a `Tree` can be either an `Empty` object or a `Node` class, and it can only be one of them because: 1, `Tree` itself cannot be instantiated; 2, `Tree` is sealed, so `Empty` and `Node` are the only subclasses of it. This means that type `Tree` is equivalent to type `Empty + Node`.

Of course, it would not be real sum types without pattern matching, and here is how to do that in Scala:

```
def max(t: Tree): Option[Int] = t match {
  case Empty => None
  case Node(_, v, Empty) => Some(v)
  case Node(_, _, r) => max(r)
}
```

2.3 GADTs

In most cases, GADTs can be encoded in Scala by using subtyping, parametric polymorphism, dispatching, etc. Take the following code snippet from Haskell wiki [1] as example:

```
data Expr a where
    I   :: Int  -> Expr Int
    B   :: Bool -> Expr Bool
    Add :: Expr Int -> Expr Int -> Expr Int
    Mul :: Expr Int -> Expr Int -> Expr Int
    Eq  :: Expr Int -> Expr Int -> Expr Bool

eval :: Expr a -> a
eval (I n) = n
eval (B b) = b
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Eq  e1 e2) = eval e1 == eval e2
```

It can be expressed in Scala in the following way:

```
sealed trait Expr[A] {
  def eval: A
}

case class I(n: Int) extends Expr[Int] {
  def eval = n
}

case class B(b: Boolean) extends Expr[Boolean] {
  def eval = b
}

case class Add(a: Expr[Int], b: Expr[Int]) extends Expr[Int] {
  def eval = a.eval + b.eval
}

case class Mul(a: Expr[Int], b: Expr[Int]) extends Expr[Int] {
  def eval = a.eval * b.eval
}

case class Eq(a: Expr[Boolean], b: Expr[Boolean])
  extends Expr[Boolean] {
  def eval = a.eval == b.eval
}
```

[1] See `https://en.wikibooks.org/wiki/Haskell/GADT`.

2.4 Postscripts

Section 2.1 and Section 2.2 are largely inspired by [Gurnell, 2016], but the words and examples are my own. I came up with the representation of GADTs in Scala, as shown in Section 2.3, by my own. However, I found out later that similar techniques have already been shown and well studied in C# [Kennedy and Russo, 2005].

3 Singleton Types in Scala

Singleton types grant the compiler with the ability to distinguish values at type level, and enable verifications on properties which are not typically verifiable by a type checker. For example, natural number 0 does not have any predecessor while 1 does, even though they share the same type (*i.e.*, `Nat`). A type checker would not be able to deduce that because common type constructs such as `Nat` do not contain such information. Singleton types encode such dependency between values and types by assigning each value with one unique type (*i.e.*, each type has exactly one inhabitant, which is the value).

3.1 Singleton Types for Inductive Types

Before discussing how to define singleton types in Scala, let's see how we can do that in Haskell:

```
{-# LANGUAGE GADTs, DataKinds, KindSignatures #-}
data Nat = Z | S Nat

data SNat :: Nat -> *  where
  SZ :: SNat 'Z
  SS :: SNat n -> SNat ('S n)

pred :: SNat ('S n) -> SNat n
pred (SS n) = n
```

First we need to define a data type called `Nat`. Then we need to define the singleton types for each value in `Nat`, by using kind (or type constructor) in the exact same way of defining `Nat` (but this time, we define a kind `SNat` instead of type `Nat`) [2]. The last two lines show how to define a `pred` function as we expected: given a number greater than 0 (`SS ...`), the function returns its predecessor; given 0 (`SZ`), the compiler will report a type error.

How do we do that in Scala? Here's what it looks like:

```
sealed trait Nat
case object Z extends Nat
```

[2] In practice, one can use the Haskell singleton library to avoid some boilerplates here [Eisenberg and Weirich, 2013]. However, I choose to write the code in this way so we can understand better what is going on in encoding singleton types in Haskell.

```
case class Succ[P <: Nat](pred: P) extends Nat
```

Surprisingly, that's all. Now the compiler knows that `Z` has type `Z.type`, `Succ(Z)` has type `Succ[Z.type]`, and `Succ(Succ(Z))` has type `Succ[Succ[Z.type]]`. If we call `Succ(Z).pred`, we get `Z` (of type `Z.type`), but calling `Z.pred` would result in a type error, because `pred` is not defined on `Z`!

Here is how it works:

1. Defining a `case object` will automatically get a singleton type in Scala. In the same time we defined the case object `Z`, we defined its type `Z.type`.

2. Each case class is at the same time a constructor and a type, In the example given by Section 2.2, `Node(Empty, 0, Empty)` would construct a data of type `Node`.

3. However, adding parametric polymorphism in our example makes it a bit different, for `Succ` is now both a data constructor (like `S` in the Haskell example) and a type constructor, or a kind (like `SS` in the Haskell example), because it now takes a type parameter to make a concrete type.

4. There can only be two sorts of `Nat`: `Z.type` or `Succ` [3]. Because our base case `Z.type` is a singleton type, and `Succ` only takes `Nat` as its parameter, by induction we know that `Succ` constructs singleton types.

5. To construct a natural number `Succ(n)`, we need its predecessor `n`, which is exactly what we want in our `pred` function. In Scala, we can access a field in the constructor of a case class by direcly calling its name: this is why we can just call `Succ(Z).pred` without defining any extra functions.

However, we cannot define a function like, for example, `predpred` in this way, but it would still be very easy:

```
def predpred[N <: Nat](n: Succ[Succ[N]]): N = n.pred.pred
```

or (if we want to use this method in the way of `Succ(Succ(Z)).predpred`):

```
case class SS[N <: Nat](predpred: N)
implicit def toSS[N <: Nat](n: Succ[Succ[N]]): SS[N] = SS(n.pred.pred)
```

3.2 Constructing Values from Singleton Types

Knowing that a value is of some singleton type is only one side of the story. One may wonder if we can know the value held in a singleton type, just by looking at the type, since there is only one inhabitant in this type?

On first thought, this might be impossible or at least very hard, because this requires a pattern matching and a recursive function which run during compile-time at type level! However, it turns out that there is an easy way to implement that in Scala – by using implicits.

[3] Remember that `Nat` itself cannot be instantiated.

In Scala, one can define a parameter of a function to be implicit. Programmers will not need to pass any values to this parameter when calling this function, because the compiler will automatically find an implicit definition of the same type of this parameter within some search scope. For example, we can write the following naive program:

```
implicit def i: Int = 1
def f(implicit x: Int) = x + 1
```

Calling `f` will return 2, even though we did not explicitly pass any arguments to function `f`.

Of course, the above example sounds like just some syntactic sugar with nothing magical, but imagine this: what happened if we declare a function with implicit parameters as an implicit definition? Or as a more concrete example, imagine what will happen if we write down the following code:

```
def get[N <: Nat](implicit nv: NatVal[N]) = nv.v

sealed trait NatVal[N <: Nat] {
  def v: N
}

implicit def zVal = new NatVal[Z.type] {
  def v = Z
}

implicit def sVal[N <: Nat](implicit pv: NatVal[N]) =
  new NatVal[Succ[N]] {
    def v = Succ(pv.v)
  }
```

Let's try to understand this by looking at what will happen step by step when we call `get[Succ[Z.type]]`:

1. The Scala compiler searches for an implicit definition whose type is the same as `get[Succ[Z.type]]`'s implicit parameter, *i.e.*, `NatVal[Succ[Z.type]]`. It will find `sVal[Z.type]`.

2. The Scala compiler searches for an implicit definition whose type is the same as `sVal[Z.type]`'s implicit parameter, *i.e.*, `NatVal[Z.type]`. It will find `zVal`.

3. The Scala compiler tries to pass appropriate arguments to the function call `get[Succ[Z.type]]`, so the actual function call happened here is `get[Succ[Z.type]](sVal[Z.type](zVal))`.

4. The above function call will return `Succ(Z)` at runtime.

And calling `get[Z.type]` will return `Z`, calling `get[Succ[Succ[Z.type]]]` will return `Succ(Succ(Z))`.

What happens here is that the Scala compiler automatically performs some pattern matchings (find `zVal` if `NatVal[Z.type]` is required, `sVal` otherwise) and recursive searches (find another `sVal` when searching for appropriate argument to be passed to `sVal`) to find the appropriate implicit definitions at compile time, according only to type level information.

What this means is that not only we have a way to find the singleton type of a value, we can also find the value of a singleton type. The link between a value and its singleton type is bidirectional.

I have not found any equivalent examples in Haskell.

3.3 Singleton Types for Functions

Now we have our natural numbers, we would like to have some functions on them, and we would want to use our singleton types to verify if the functions are correct. Let's use `plus` as an example. In Haskell, you would need to do this:

```
{-# LANGUAGE GADTs, DataKinds, KindSignatures, TypeFamilies #-}
type family SPlus(n :: Nat)(m :: Nat) :: Nat where
  SPlus 'Z m = m
  SPlus ('S n) m = 'S (SPlus n m)

plus :: SNat n -> SNat m -> SNat (SPlus n m)
plus SZ m = m
plus (SS n) m = SS (plus n m)
```

Notice that the `SPlus` in the type signature of `plus`. It looks like a function, but works at the type level.

Here's how we do that in Scala:

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

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

case class Succ[P <: Nat](pred: P) extends Nat {
  type Plus[M <: Nat] = Succ[pred.Plus[M]]
  override def +[M <: Nat](m: M) = Succ(pred + m)
}
```

The magic starts at the second line: we define a type alias `Plus`, but we don't know anything about this type except that it takes a subclass of `Nat` as its parameter, and it itself is also a subclass of `Nat`. Then we define our method `+` with the return type of `Plus`, even though we don't know what type `Plus` is!

The job of defining what type `Plus` is falls on the subclasses of `Nat`: in the case of `Z`, `Plus[M]` just means `M`; in the case of `Succ[N]`, `Plus[M]` means `Succ[pred.Plus[M]]` (even though we don't know what type `pred.Plus` is!). Now we can just override our `+` methods on `Z` and `Succ` even though they might have different type signatures. These all just work thanks to Scala's unique feature called path-dependent types [4].

Now let's try our `+` method:

```
lastland-scala@ Succ(Z) + Z
res1: Succ[Z.type] = Succ(Z)
lastland-scala@ Succ(Z) + Succ(Succ(Z))
res2: Succ[Succ[Succ[Z.type]]] = Succ(Succ(Succ(Z)))
lastland-scala@ Z + Succ(Succ(Z))
res3: Succ[Succ[Z.type]] = Succ(Succ(Z))
```

Notice that, even though the return type of `+` is `Plus`, it actually returns the singleton type of the return value. This is because the `type` keyword we used is nothing more than just a type alias, so `Plus[Z.type, Z.type]` is actually just `Z.type`.

Suppose we have accidentally defined our `+` method on `Succ` in the following (wrong) way:

```
override def +[M <: Nat](m: M) = Succ(pred + m.pred)
```

We would get a type error because compiler knows that `Succ(pred + m.pred)` does not have type `Plus[M]` (or `Succ[pred.Plus[M]]`).

3.4 Using Singleton Types

Verifying the plus function sounds trivial, so let's verify something more interesting. Let's suppose that we would like to define an append function, and verify that the result vector returned by our implementation does have the length of `n + m`, where `n` and `m` are the lengthes of the input vectors, respectively.

Here's what we would write in Haskell:

```
data Vec :: * -> Nat -> * where
  Nil :: Vec a 'Z
  Cons :: a -> Vec a n -> Vec a ('S n)

app :: Vec a n -> Vec a m -> Vec a (SPlus n m)
app Nil l = l
app (Cons h t) l = Cons h (app t l)
```

[4] For more information, check The Neophyte's Guide to Scala Part 13: Path-dependent Types.

One might think that this would be very straightforward, since we have already defined our `Plus` type alias on `N`:

```
sealed trait Vec[N <: Nat] {
  def app[M <: Nat](l: Vec[M]): Vec[N.Plus[M]]
}

case object Nil extends Vec[Z.type] {
  ... // some implementation
}

case class Cons[N <: Nat](h: Int, t: Vec[N])
  extends Vec[Succ[N]] {
  ... // some implementation
}
```

Unfortunately, this does not work. The problem is that the type alias `Plus` is defined on the values, not on the types! What we need is something like `Vec[A, n.Plus[M]]`, but we cannot do that because we do not have `n`. One might think that we can use the `get` method we have defined in Section 3.2 to get `n`, but that is also infeasible because implicits require the concrete type information to be available at compile-time (so compiler can search for implicit values of that type), but all we know is that this type is a subtype of `Nat`.

We cannot re-define our `Plus` type in our new case classes, either, because we don't know the `pred` of `n` if we do not know `n`, and there is no way to define a `pred` function purely at type level.

Another intuitive approach would be writing down a method like this:

```
def app[N, M](x: Vec[N], y: Vec[M]) = x match {
  case Nil => y
  case Cons(h, t) => Cons(h, app(t, y))
}
```

However, this still does not work because the function is recursive, and, in Scala, we have to specify the return type of a recursive function. In this case, we don't know the return type.

However, we don't really need these methods to define our `app` function. Notice that, we only care about the predecessor of a length in case of `Cons[N]`, which is `Vec[Succ[N]]`. And we do know the predecessor of `Succ[N]`, it's just `N`!

Once again, we use path-dependent types to define the return type of a function in different subclasses separately:

```
sealed trait Vec[N <: Nat] {
  type NM[M <: Nat] <: Nat
  type T[M <: Nat] = Vec[NM[M]]
  def app[M <: Nat](b: Vec[M]): T[M]
}
```

```
case object Nil extends Vec[Z.type] {
  type NM[M <: Nat] = M
  def app[M <: Nat](b: Vec[M]) = b
}

case class Cons[N <: Nat](h: Int, t: Vec[N])
    extends Vec[Succ[N]] {
  type NM[M <: Nat] = Succ[t.NM[M]]
  def app[M <: Nat](b: Vec[M]) = Cons(h, t.app(b))
}
```

3.5 Postscripts

All the Haskell code snippets shown in this section are based on a version Stephanie Weirich has written on a whiteboard. [Ishii, 2014] has also been very useful in helping me understand what is going on in these implemtations.

All the Scala code snippets shown in this section are implemented by myself, but I would not be able to write them down without looking at some other people's code and borrowing ideas from them. In particular: the idea of using subtyping and parametric polymorphism to model singleton types of natural numbers comes from Miles Sabin's implementation of `Nat` in shapeless [5]; the idea of using implicits to generate a value from a singleton type was inspired by [Brady and Sabin, 2013] [6], though the technique was originally used to implement singleton types for functions there; the idea of using path dependent types to implement singleton types for functions was inspired by Miles Sabin's implementation of dependently typed red-black tree in Scala [7].

To the best of my knowledge, this is the first article which summarizes all these techniques used in implementing singleton types in Scala.

4 Case Study: Red-Black Tree

Now we have seen several basic techniques in implementing dependent types in Scala, we can move on for a more exciting example: a red-black tree.

Implementing red-black trees using dependent types is already a complicated subject, and will only be briefly discussed here. Some good in-depth tutorials can be found in [Weirich, 2015; Chlipala, 2013]. In this section, we will follow the implementation found in [Weirich, 2015].

[5] Shapeless: generic programming for Scala. `https://github.com/milessabin/shapeless`. In particular, the idea comes from this file: `https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/nat.scala`. However, I modified the way zero `Z` is defined by using singleton objects in Scala.

[6] Miles Sabin's demo code can be found here: `https://github.com/milessabin/strangeloop-2013`.

[7] Miles Sabin's implementation can be found here: `https://github.com/milessabin/tls-philly-rbtree-2016`

4.1 Data Structures

Red-black trees are a popular functional data structure with the following interesting invariants [8]:

1. A node is either red or black.
2. The root is black.
3. All leaves are black.
4. If a node is red, then both its children are black.
5. Every path from a given node to any of its descendant leaf nodes contains the same number of black nodes.

We can encode all these invariants using dependent types. Here's how we can do that in Haskell:

```
data Color :: * where
    R :: Color
    B :: Color

-- invariants 1, 3, 4, and 5
data Tree :: Color -> Nat -> * where
  E  :: Tree 'B 'Z
  TR :: Tree 'B n  -> Int -> Tree 'B n  -> Tree 'R n
  TB :: Tree c1 n -> Int -> Tree c2 n -> Tree 'B ('S n)

-- invariant 2
data RBT :: * where
  Root :: Tree 'B m -> RBT

-- some example functions
rev :: Tree c n -> Tree c n
rev E = E
rev (TR a x b) = TR (rev b) x (rev a)
rev (TB a x b) = TB (rev b) x (rev a)

maxB :: Tree B (S n) -> Int
maxB (TB _ x E)          = x
maxB (TB _ _ (TR a x b)) = maxR (TR a x b)
maxB (TB _ _ (TB a x b)) = maxB (TB a x b)

maxR :: Tree R n -> Int
maxR (TR _ x E)          = x
maxR (TR _ _ (TB a x b)) = maxB (TB a x b)
```

[8] According to Wikipedia: `https://en.wikipedia.org/wiki/Red%E2%80%93black_tree`

Let's use the techniques we have seen in Section 2 and Section 3 to translate the code into Scala:

1. `Color` is a kind which has two types `R` and `B`. Because a case class is at the same time a constructor and a type, we can just define `Color` as a sum type `Red + Black`, and use the methods we have seen in Section 2.2 to encode sum types.

2. `Tree` is a kind which takes two type parameters `Color` and `Nat`, the equivalent construct in Scala would be `Tree[C <: Color, N <: Nat]`, as we have seen in Section 3.1.

3. Both `TR` and `TB` are inductive types defined using GADTs, therefore, we can use the methods we have seen in Section 2.3. The same technique applies to the `RBT` kind.

4. The return type of `maxB` and `maxR` is fixed, so we can just encode them as ordinary methods defined on `Tree` classes.

5. The return type of function `rev` is dependent on the input parameter, but it does not include any type level functions, so we can just encode them as ordinary methods defined on `Tree` classes, like `maxB` and `maxR`.

So here's what our Scala implementation looks like (with some slight modifications from our analysis):

```
sealed trait Color
case object Red extends Color
case object Black extends Color

// invariant 1
sealed trait Tree[+C <: Color, N <: Nat] {
  def rev: Tree[C, N]
  def max: Option[Int]
  def maxDefault(d: Int): Int
}

// invariant 3
case object Empty extends Tree[Black.type, Z.type] {
  def rev = Empty
  def max = None
  def maxDefault(d: Int) = d
}

// invariant 4 and 5
case class RedNode[N <: Nat]
  (l: Tree[Black.type, N], v: Int, r: Tree[Black.type, N])
    extends Tree[Red.type, N] {
```

```
  def rev = RedNode(r.rev, v, l.rev)
  def max = Some(r.maxDefault(v))
  def maxDefault(d: Int) = r.maxDefault(v)
}


// invariant 5
case class BlackNode[N <: Nat]
  (l: Tree[Color, N], v: Int, r: Tree[Color, N])
    extends Tree[Black.type, Succ[N]] {
  def rev = BlackNode(r.rev, v, l.rev)
  def max = Some(r.maxDefault(v))
  def maxDefault(d: Int) = r.maxDefault(v)
}

// invariant 2
case class RBT(root: Tree[Black.type, _ <: Nat])
```

A few test cases:

- Calling `BlackNode(BlackNode(Empty, 1, Empty), 2, BlackNode(Empty, 3, Empty)).rev` will return `BlackNode(BlackNode(Empty, 3, Empty), 2, BlackNode(Empty, 1, Empty))`.
- Calling `BlackNode(BlackNode(Empty, 1, Empty), 2, BlackNode(Empty, 3, Empty)).max` will return `Some(3)`.
- Trying to construct `RedNode(BlackNode(Empty, 1, Empty), 2, Empty)`, however, would result in a type error because it breaks invariant 5.
- Trying to construct `RedNode(RedNode(Empty, 1, Empty), 2, Empty)` would result in a type error because it breaks invariant 4.

4.2 Balancing Operations

Implementing the balancing functions on a dependently typed red-black tree is tricky, because the above invariants will be violated during the operations. For this reason, we need some new data structures with weaker invariants.

4.2.1 Almost Trees and Hidden Trees

We will need two new data structures for our balancing functions:

1. A data structure to represent a non-empty tree whose own color is hidden. We will call it a `HiddenTree`.
2. A data structure to represent a non-empty tree which has children of arbitrary colors. We will call it an `AlmostTree`.

Here is how we define them in Haskell:

```
data Sing (c :: Color) :: * where
    SR :: Sing R
    SB :: Sing B
deriving instance Show (Sing c)

type family Incr (c :: Color) (x :: Nat) :: Nat where
    Incr R x = x
    Incr B x = S x

data HiddenTree :: Nat -> * where
  HR :: Tree R n     -> HiddenTree n
  HB :: Tree B (S n) -> HiddenTree (S n)
deriving instance Show (HiddenTree n)

data AlmostTree :: Nat -> * where
  AT :: Sing c -> (Tree c1 n) -> Int -> (Tree c2 n) -> AlmostTree (Incr c n)
deriving instance Show (AlmostTree n)
```

Defining `HiddenTree` in Scala is quite straightforward:

```
sealed trait HiddenTree[N <: Nat] {
case class HR[N <: Nat](t: RedNode[N])
  extends HiddenTree[N]
case class HB[N <: Nat](t: BlackNode[N])
  extends HiddenTree[Succ[N]]
```

The definition of `AlmostTree` involves a type family called `Incr`. As shown in Section 3.3 and Section 3.4, we would need path-dependent types. Because the return value of `Incr` depends on the colors, the `Color` trait and its subclasses would be our best place to define our path-dependent types in Scala. We modify the definition of `Color` and its subclasses shown in Section 4.1 as follows:

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

sealed trait Color {
  type Incr[N <: Nat] <: Nat
}

case object Red extends Color {
  type Incr[N <: Nat] = N
  case class AT[N <: Nat]
    (l: Tree[Color, N], v: Int, r: Tree[Color, N])
      extends AlmostTree[Incr[N]]
}

case object Black extends Color {
```

```
  type Incr[N <: Nat] = Succ[N]
  case class AT[N <: Nat]
    (l: Tree[Color, N], v: Int, r: Tree[Color, N])
      extends AlmostTree[Incr[N]]
}
```

Let's see how to construct an `AlmostTree` in these two languages. In Haskell, it would be something similar to this:

```
AT SR l v r
```

In Scala, it would be something similar to this (notice the position of `Red`):

```
Red.AT(l, v, r)
```

4.2.2 Balancing Functions

With our two new data structures defined, we can now write down our balancing functions. For simplicity we only show the `balanceLB` function below, because other functions are quite similar:

```
-- input color is implicitly black
balanceLB :: AlmostTree n -> Int -> Tree c n -> HiddenTree (S n)
balanceLB (AT SR (TR a x b) y c) z d = HR (TR (TB a x b) y (TB c z d))
balanceLB (AT SR a x (TR b y c)) z d = HR (TR (TB a x b) y (TB c z d))
balanceLB (AT SB a x b) kv r = HB (TB (TB a x b) kv r)
balanceLB (AT SR E x E) kv r = HB (TB (TR E x E) kv r)
balanceLB (AT SR (TB a1 x1 a2) x (TB b1 y1 b2)) y c =
  HB (TB (TR (TB a1 x1 a2) x (TB b1 y1 b2)) y c)
balanceLB _ _ _ = error "balanceLB: this is impossible"
```

One might come up with the following direct translation to Scala:

```
sealed trait AlmostTree[N <: Nat] {
  type S <: Nat
  def balanceLB(x: Int, t: Tree[Color, N]): HiddenTree[S]
}

sealed trait Color {
  type Incr[N <: Nat] <: Nat
}

case object Red extends Color {
  type Incr[N <: Nat] = N
  case class AT[N <: Nat]
    (l: Tree[Color, N], v: Int, r: Tree[Color, N])
      extends AlmostTree[Incr[N]] {
    override def balanceLB(z: Int, d: Tree[Color, N]) =
      (l, v, r) match {
```

```
        case (RedNode(a, x, b), y, c) =>
          HR(RedNode(BlackNode(a, x, b), y, BlackNode(c, z, d)))
        case (a, x, RedNode(b, y, c)) =>
          HR(RedNode(BlackNode(a, x, b), y, BlackNode(c, z, d)))
        case (Empty, x, Empty) =>
          HB(BlackNode(RedNode(Empty, x, Empty), z, d))
        case (BlackNode(a1, x1, a2), x, BlackNode(b1, y1, b2)) =>
          HB(BlackNode(RedNode(
            BlackNode(a1, x1, a2), x, BlackNode(b1, y1, b2)), z, d))
      }
  }
}

case object Black extends Color {
  type Incr[N <: Nat] = Succ[N]
  case class AT[N <: Nat]
    (l: Tree[Color, N], v: Int, r: Tree[Color, N])
      extends AlmostTree[Incr[N]] {
    type S = Succ[Succ[N]]
    override def balanceLB(z: Int, d: Tree[Color, Incr[N]]) =
      HB(BlackNode(BlackNode(l, v, r), z, d))
  }
}
```

Unfortunately this does not type check. Why? Let's look at the third and fourth cases inside `Red.AT.balanceLB`:

- `Empty` has type `Tree[Black.type, Z.type]`, but the types of our function parameters are `Tree[Color, N]`! While `Black.type` does match `Color` in the covariant position, there is no way see if that `Z.type` matches `N`.
- `BlackNode(a1, x1, a2)` has type `Tree[Black.type, Succ[N]]`, but the type of our function parameters are `Tree[Color, N]`. We need to construct a tree of type `Tree[Black.type, N]`, but we don't have any way to do that.

Notice that both the third and the fourth cases match two black nodes, so we can merge them into one case. Now we just need to find a way to represent a tree of type `Tree[Black.type, N]`. The trick here is adding an intermediate layer in our definitions:

```
sealed trait TreeB[N <: Nat] extends Tree[Black.type, N]

case object Empty extends TreeB[Z.type] {
  ...
}

case class BlackNode[N <: Nat](
```

```
  l: Tree[Color, N], v: Int, r: Tree[Color, N])
    extends TreeB[Succ[N]] {
  ...
}
```

Now we can define our `Red.AT.balanceLB` as follows:

```
case object Red extends Color {
  type Incr[N <: Nat] = N
  case class AT[N <: Nat]
    (l: Tree[Color, N], v: Int, r: Tree[Color, N])
      extends AlmostTree[Incr[N]] {
    def blacken = RBT(BlackNode(l, v, r))
    def balanceLB(z: Int, d: Tree[Color, N]) = (l, v, r) match {
      case (RedNode(a, x, b), y, c) =>
        HR(RedNode(BlackNode(a, x, b), y, BlackNode(c, z, d)))
      case (a, x, RedNode(b, y, c)) =>
        HR(RedNode(BlackNode(a, x, b), y, BlackNode(c, z, d)))
      case (a: TreeB[N], x, b: TreeB[N]) =>
        HB(BlackNode(RedNode(a, x, b), v, r))
    }
  }
}
```

4.3 Insertions

Now implementing insertions should be quite straightforward, since all the data structures and balancing functions have been defined. For simplicity, we only show one particular function, `insAny`, here.

The implementation of `insAny` in Haskell:

```
insAny :: Tree c n -> Int -> AlmostTree n
insAny (TR l y r) x =  case compare x y of
   LT -> balanceLR (insBlack l x) y r
   GT -> balanceRR l y (insBlack r x)
   EQ -> AT SR l y r
insAny (TB l y r) x = forget (insBlack (TB l y r) x)
insAny E          x = forget (insBlack E x)
```

Since no type-level function is required, the above functions can be implemented directly by using methods in Scala:

```
sealed trait Tree[+C <: Color, N <: Nat] {
  ...
  def insAny(x: Int): AlmostTree[N]
}
```

```
sealed trait TreeB[N <: Nat] extends Tree[Black.type, N] {
  ...
  def insAny(x: Int) = insBlack(x).forget
}

case class RedNode[N <: Nat]
  (l: Tree[Black.type, N], v: Int, r: Tree[Black.type, N])
    extends Tree[Red.type, N] {
  ...
  def insAny(x: Int) =
    if (x < v) {
      l.insBlack(x).balanceLR(v, r)
    } else if (x > v) {
      r.insBlack(x).balanceRR(l, v)
    } else {
      Red.AT[N](l, v, r)
    }
}
```

There is no need to implement `insAny` method on the `Empty` and `BlackNode` subclasses, because they can just inherit the definition from `TreeB`.

4.4 Postscripts

The Haskell implementation used in this section comes directly from [Weirich, 2015] [9], with slight modifications.

The Scala implementation is written by myself, but the idea of adding `TreeB` as an intermediate level to represent black nodes is borrowed from Miles Sabin's dependently typed red-black tree implementation in Scala [10].

My complete implementation of dependently typed red-black tree in Scala can be found in `https://github.com/lastland/DTScala/blob/master/src/main/scala/RBT.scala`.

5 Understanding Dependent Types in Scala

In the previous sections, we have shown how to write dependently typed programs in both Haskell and Scala. As readers might have noticed now, the code written in these two languages look rather different, even though they are implementing the same thing:

- When implementing algebraic types and inductive/recursive types, we used algebraic data types in Haskell, but we used record and subtyping in Scala.

[9] The code can be found in `https://github.com/sweirich/dth/blob/master/depending-on-types/RBT.hs`.

[10] Miles Sabin's implementation can be found here: `https://github.com/milessabin/tls-philly-rbtree-2016`

- When implementing singleton types for inductive types, we used data kind promotions and GADTs in Haskell, but we used subtyping and parametric polymorphism in Scala.
- When implementing singleton types for functions, we used type families in Haskell, but we used subtyping and path-dependent types in Scala.

Trying to understand all these features is difficult. Instead, I will try to focus on one particular feature. It seems that support for subtyping makes a huge difference between these two languages according to the above comparisons, so we are going to try to understand the differences between Haskell and Scala by studying subtyping in more details in this section.

5.1 Recursive Types vs. Subtyping

Before talking about subtyping, we will make some detours: let's start with recursive types. To better discuss the core of recursive types, we will borrow some formal definitions from system FPC described in [Harper, 2016]:

$$\begin{array}{rll} \text{Typ } \tau ::= & t & \text{self-reference} \\ & \text{rec}(t.\tau) & \text{recursive types} \\ \text{Exp } e ::= & \text{fold}\{t.\tau\}(e) & \text{fold} \\ & \text{unfold}(e) & \text{unfold} \end{array}$$

The introduction and elimination forms for recursive types are given below:

$$\frac{\Gamma \vdash e : [\text{rec}(t.\tau)/t]\tau}{\Gamma \vdash \text{fold}\{t.\tau\}(e) : \text{rec}(t.\tau)}$$

$$\frac{\Gamma \vdash e : \text{rec}(t.\tau)}{\Gamma \vdash \text{unfold}(e) : [\text{rec}(t.\tau)/t]\tau}$$

One of the major reasons we want recursive types is that we want some methods to define infinite data types. Infinite data types, in contrast with finite data types such as product types and sum types, are types which have infinite inhabitants. We cannot directly define infinite data types by using finite data types only, so recursive types come to the rescue by providing some way to equate types.

For example, natural numbers can be expressed in the following way using the above definitions of recursive types:

$$\text{rec}(t.[\text{z} \hookrightarrow \text{unit}, \text{s} \hookrightarrow t])$$

If we use ρ to represent the above expression, we can have the following equation:

$$\rho \cong [\text{z} \hookrightarrow \text{unit}, \text{s} \hookrightarrow \rho]$$

because the expression on right hand side of the equation can be folded into the expression on the left hand side, and the expression on the left hand side can also be unfolded into the expression on the right hand side. In this way, $s(e)$ will have the same type as e, provided that e is of type ρ.

Recall how we would define natural numbers (without singleton types) in Haskell:

```
data Nat = Z | S Nat
```

We can express these definitions in Haskell using system FPC (for simplicity, we introduce the unit type and sum types here):

$$\begin{aligned} \rho &= \mathrm{rec}(t.(\mathrm{Unit} + t)) \\ z &= \mathrm{fold}\{\rho\}(\mathrm{inl}[\mathrm{Unit}, \rho](\mathrm{unit})) \\ s(e) &= \mathrm{fold}\{\rho\}(\mathrm{inr}[\mathrm{Unit}, \rho](e)) \end{aligned}$$

where Unit stands for the unit type, and unit stands for the value of unit type.

However, notice that some information has been lost at type level in expression z and $s(e)$. The first piece of information was lost during left and right injections, the introduction form of sum types: after a left or right injection, there is no way to distinguish z and $s(e)$ according only to their types (though it is still possible to distinguish them at runtime by using pattern matchings). The second piece of information was lost during folding, the introduction form of recursive types: after folding, there is no way to distinguish e with $s(e)$ according only to their types (though, again, it is still possible to distinguish them at runtime by unfolding followed by pattern matchings).

The loss of information at type level is usually fine, but becomes a problem in implementing dependent types, where encodings of singleton types are required. Haskell addresses this problem by using data kind promotions [Yorgey *et al.*, 2012]. However, it is not the only way.

There is one rule in systems with subtyping (*e.g.*, system $\mathrm{F}_{<:}$) which provides another way of "equating" types: the rule of subsumption. The rule has many forms depending on how subtyping relations are defined in the system, but one common form can be written as follows:

$$\frac{\Gamma \vdash e : S \qquad S <: T}{\Gamma \vdash e : T}$$

It means that if an expression e is of type S and S is a subtype of T, e is also of type T.

Now, instead of using recursive types, we can define two types Z and S, both of which are subtypes of some type ρ which represents natural numbers. For any expressions with a hole where some expression of type ρ is required, we can just give an expression of either type Z or type S, and the program will type check. More importantly, one can still distinguish zeroes and other values at type level, by just testing if they are of type Z or type S. However, one has to be careful for the other direction of the "equation", because once we convert

a Z or S to ρ, we can no longer tell whether the value is Z or S according only to its type information.

The equivalences of folding (introduction) and unfolding (elimination) forms for recursive types now consist of two phases in a system with subtyping. Folding is a constructor plus an upcast, and unfolding is a downcast plus a destructor [11]. As we have discussed in the previous paragraph, upcasting will still result in loss of information at type level. Fortunately, upcasting is usually not needed in a system with both subtyping and polymorphic types. For example, instead of declaring a function to have type $\rho \to \rho$, one can also declare the function to have type $\forall x <: \rho, x \to x$.

However, subtyping alone is not enough for encoding singleton types. Recall that our natural number is an infinite data type, where its inhabitants are infinite, thus constructing singleton types for natural numbers will also require a way to define infinite amount of types, or in other words, we need a type constructor. Fortunately, there is a way to implement type constructors in systems with subtyping, *i.e.* parametric polymorphism.

Now we come back to our definitions of singleton types of natural numbers in Scala:

```
sealed trait Nat
case object Z extends Nat
case class Succ[P <: Nat](pred: P) extends Nat
```

These definitions directly encode singleton types because subtyping and parametric polymorphism (with subtyping bounds) allow different subtypes of one base type to be considered the same type, without losing any type-level information.

Notice that the above example indeed describes singleton types, but it is hard to formally discuss that using a theoretical system. For example, system $F_{<:}$ is not enough to describe what is going on in the above example, because it cannot describe an abstract class which cannot be instantiated, it cannot describe a singleton object like `Z`, and it cannot describe the *closed* subtyping relations guaranteed by the `sealed` keyword.

In conclusion, we have shown in this section that subtyping is very powerful: like recursive types, it provides a way to equate types and to define infinite data types; but it is better than recursive types in the sense that it is possible to preserve information related to its values at type level. This makes it quite easy to define singleton types in a language with subtyping. In languages which do not support subtyping, other mechanisms such as data kind promotions have to be introduced to be able to define singleton types.

5.2 Type Inference for Systems with Subtyping

In the previous section, we have seen that subtyping is quite powerful. A question comes naturally following our analysis: why Haskell chose not to support

[11] In a system which support records, a constructor can be considered storing given values into some fields, and a destructor is accessing the fields.

subtyping? Is this a mistake, or there is something else to consider?

There is indeed something else to consider when designing a language. It turns out that introducing subtyping to a system makes one thing quite difficult: *type inference*.

In Haskell, a modified version of Hindley-Milner type inference algorithm is used, so programmers usually do not need to explicitly provide any type annotations for their programs. Unfortunately, the same algorithm does not work for Scala or other programming languages with subtyping.

A type inference algorithm usually consists of two parts: constraint generation and unification. In systems without subtyping, the constraints are usually equations between types. In this case, a union-find data structure can be used to effectively solve the constraints, or return a type error when constraints are unsatisfiable.

However, with the introduction of subtyping relations, the type constraints are now usually inclusion relations. Even though some complete algorithms to generate and unify this sort of constraints exists [Aiken and Wimmers, 1993], the large number of constraints usually generated in a program makes them impractical [Pottier, 1996]. For this reason, Scala chose to perform local type inference, instead of total type inference.

The local type inference algorithm was first proposed by [Pierce and Turner, 1998], and later refined by [Odersky *et al.*, 2001] [12]. The idea is to only try to infer types using some local constraints.

The local type inference algorithm [13] will try to give the most informative types according to the local constraints. For example, suppose we have a function f of type $\forall x <: \text{Top}, x \to x$. The type of $f(1)$ will be automatically inferred as Int. Notice that if a total type inference is performed, $f(1)$ should have type $\text{Int} <: x <: \text{Top}$, where x is some type variable. However, the local type inference algorithm will try to be greedy here, and choose the most informative type, Int in this case, as a solution. When types cannot be inferred, the local type inference algorithm will ask programmers to write down more type annotations.

The way local type inference algorithm works is to mark types with three different colors: red for *inherited* types, blue for *synthesized* types, and black for arbitrary types (can be either inherited or synthesized). A type is inherited if its type can be inferred using "outside" information. A type is synthesized if its type can be inferred using "inside" information.

For example, suppose we have function f of type $\forall x, (x \to x) \to x$, and we want to pass a lambda expression for identity function defined on Int to it. Without type inference, we will have to write $f[\text{Int}](\lambda(x : \text{Int}).x)$, where $f[\text{Int}]$ means applying type Int to the type variable in the type of f. However, with local type inference, we can either write $f[\text{Int}](\lambda x.x)$ or $f(\lambda(x : \text{Int}).x)$. In the first case, the type of the lambda expression's parameter is *inherited* from "outside", *i.e.*, $f[\text{Int}]$. In the second case, the type of f is *synthesized* according

[12] I have no direct evidence indicating that this algorithm is the algorithm used in Scala, but the behaviors I tested in Scala conform the descriptions in this paper.

[13] In this report, the local type inference algorithm always refers to the colored local type inference algorithm described in [Odersky *et al.*, 2001].

to its "inside" information, *i.e.*, $\lambda(x : \text{Int}).x$. However, it is not possible to simply write $f(\lambda x.x)$, because in this case none of the type variables can be inherited or synthesized to get its most informative version.

However, if our type of f is changed to $\forall x, (\text{Int} \to x) \to x$, we can write $f(\lambda x.x)$ because now the lambda expression's parameter type can be inherited, its return type can be synthesized, and then the type variable in the type of f can also be synthesized. It is also worth mentioning that if the type of f is changed $\forall x, (x \to \text{Int}) \to x$, we can no longer write $f(\lambda x.x)$, because the algorithm requires that a lambda expression's parameter type must be inherited.

Notice that the above discussion is only about the type inference problems brought by subtyping. Concepts such as classes, methods, abstract classes, etc., which are common in main-stream object-oriented programming languages such as Scala, can only make type inference more difficult. The consequences are, programmer are usually required to write much more type annotations when programming in languages with subtyping, such as Scala, than they would when programming in languages without subtyping, such as Haskell.

However, one might think that this may not be a problem in dependently typed programs, since we want to write down type annotations as specifications anyway. Unfortunately, having to write down type annotations sometimes is not the only problem. The local type inference algorithm is a greedy algorithm and sometimes it can be wrong!

For example, recall the `insAny` method in our dependently typed red-black tree implementation in Scala in Section 4.3:

```
case class RedNode[N <: Nat]
  (l: Tree[Black.type, N], v: Int, r: Tree[Black.type, N])
    extends Tree[Red.type, N] {
  ...
  def insAny(x: Int) =
    if (x < v) {
      l.insBlack(x).balanceLR(v, r)
    } else if (x > v) {
      r.insBlack(x).balanceRR(l, v)
    } else {
      Red.AT[N](l, v, r)
    }
}
```

If we replace `Red.AT[N](l, v, r)` with `Red.AT(l, v, r)`, we will get a type error:

```
[error]  found   : dependent.rbt.Tree[dependent.rbt.Black.type,N]
[error]  required: dependent.rbt.Tree[dependent.rbt.Color,Nothing]
[error] Note: N >: Nothing, but trait Tree is invariant in type N.
[error] You may wish to define N as -N instead. (SLS 4.5)
[error]       Red.AT(l, v, r)
[error]              ^
```

The reason is that the local type inference will always try to avoid maintaining type variables and constraints, by greedily infer a most informative type for an expression. In this case, synthesized type for the type variable in `Red.AT` is `N`, which is another type variable with constraint `Nothing <: N <: Nat`. The most informative version for `N` is `Nothing`, so the Scala compiler will infer that `Red.AT` is instantiated with type `Nothing`. However, this is usually not a big problem, since programmers can provide a type annotation here to make the Scala compiler accept it.

Another example, which is more difficult to solve, is the pattern matching we have seen in the `balanceLB` function in our dependently typed red-black tree implementation in Section 4.2. Here we use a much simpler example, which fails to compile for the same reason, to understand what is going on there:

```
def id[N <: Nat](n: N): N = n match {
  case Z => Z
  case Succ(p) => Succ(p)
}
```

The Scala compiler will report a type error for this obviously correct program:

```
[error]  found   : dependent.nat.Succ[dependent.nat.Nat]
[error]  required: N
[error]     case Succ(p) => Succ(p)
[error]                          ^
```

There is no problem in the first case, but in the second case, the Scala compiler will infer that `p` is of type `Nat`, instead of `N`! The reason is similar to that of the previous example: the local type inference algorithm greedily decides a most informative type to avoid maintaining type variables and constraints.

More specifically, the Scala compiler will first generate a type variable, assuming a, for `p`, and generate constraints similar to $\{a <: \texttt{Nat} \wedge \texttt{Succ}[a] <: \texttt{N}\}$. However, the constraints are not solvable because the type variable `N` is also unknown, so the Scala compiler will now make `N` another type variable, assuming b, and generate constraints similar to $\{a <: \texttt{Nat} \wedge \texttt{Succ}[a] <: b \wedge b <: \texttt{Nat}\}$, which is satisfiable [14]. Because `p` is used later in the clause, the Scala compiler will also need to decide a type for it. Again, because the local type inference algorithm does not try to maintain type variables or constraints, it will try to give `p` a most informative type. Here `p` is a newly binded variable in a contravariant position, like the parameter of a function, so the algorithm will decide that the most informative type of `p` is `Nat`. Clearly, `Succ[Nat]` does not conform to `N`, the return type of our function.

Different from the previous example, we cannot fix this problem by providing more type annotations, because it is not supported by the syntax. However, just

[14] This part is not covered in [Odersky *et al.*, 2001], but some information can be found in `http://www.scala-lang.org/files/archive/spec/2.11/08-pattern-matching.html#type-parameter-inference-in-patterns`.

as what we have seen in implementing the `balanceLB` method for a red-black tree, there are some way to fix this by modifying the program [15]:

```
def id[N <: Nat](n: N): N = n match {
  case Z => Z
  case Succ(p) => n
}
```

or simply:

```
def id[N <: Nat](n: N): N = n
```

5.3 Conclusion

In this section, first we show that subtyping is a powerful feature because it provides a way to equate types without losing type-level information, in contrast with recursive types, which loses some type-level information during folding and unfolding. This feature, combining with parametric polymorphism, makes it very easy to implement singleton types and dependent types in Scala. Haskell, on the other hand, does not support subtyping, so some extra mechanisms such as data kind promotions have to be introduced, to support constructing singleton types.

However, subtyping also makes it much more difficult to perform type inference. For this reason, some object-oriented programming languages like Scala choose to perform local type inference, instead of total type inference which is available in languages without subtyping such as Haskell. As we have shown in this section, local type inference algorithm can result in compiler rejecting correct programs, due to its greedy strategy, especially for dependently typed programs where type variables are common. While there are usually some way to fix these problems, it is unclear that if such a way always exists.

5.4 Postscripts

Though I am certain that I am not even close to be the first one to realize the power of subtyping and the consequences it has brought, I have not yet found an article discussing this topic. However, due to time constraints and limited personal knowledge in the field, there are more things not covered than covered in this section, and more things to be done than have been done in this exploration.

For example, classes and methods were not discussed, though they are quite important in implementing singleton types and dependent types in Scala [16]; parametric polymorphism and its relationship with kinds were not discussed in details; path dependent types were not discussed, etc. It is also worth mentioning that, clearly, the simple well-known theoretical models such as $F_{<:}$ are not enough for us to discuss these feature, and some more complex systems must be

[15] It is not clear to me that if there will always be a fix in this type of situation.

[16] And they can also make type inference even more difficult.

studied to provide a comprehensive understanding of the features we have used in implementing dependent types in Scala [17].

There are also problems that I have met but not mentioned in this section. For example, there is an failed attempt to implement an equivalent version of a dependently typed `zipWithN` function implemented in Haskell [18]. The reason is that Scala compiler, again, rejects the program due to some type errors. At this moment, it is not clear whether this is a bug of Scala compiler, another problem caused by the local type inference algorithm, or simply some mistakes I have made in my implementation.

6 Scala vs. Haskell: Dependent Types

In Section 5, we have discussed the differences between Scala and Haskell in implementing dependent types, on a technical level. In this section, I would like to try to compare writing dependently typed programs in Scala and in Haskell from a user's point of view. Since I did not understand either how dependent types work in Haskell or in Scala when I started this exploration, this section may also be useful for people who are experienced in these languages to understand more about the difficulty a beginner might have in learning these language features [19].

First of all, the implementation of singleton types in Haskell may be very confusing for beginners (at least for me), for the use of advanced features such as GADTs, data kind promotions, and type families. On one hand, there are repeated definitions of `Nat` and `SNat` in different forms (one as data types, the other as kinds); on the other hand, data types are used in kind signatures, which makes it hard to tell whether something is a data, a type, or a kind. Though these concepts become much more clear once I have understood how this works, they are very strange and confusing in the beginning.

The implementation in Scala, however, uses almost nothing fancy [20]: traits can be understood as abstract classes or interfaces, case classes are just normal classes with some pre-defined methods, and parametric polymorphism is no stranger to object-oriented language users. They are features commonly used in practice. And there is no need to repeat definitions at both data and type levels in Scala as in Haskell. However, using implicits to implement type-level pattern matching and recursive calls as shown in Section 3.2 may looks bizarre and requires some thinking to understand how it works.

However, because Scala is an object-oriented programming language, functions are usually implemented as methods inside classes. This makes the function definitions scatter everywhere in Scala: as we have seen in Section 4, `insAny` is defined on `Tree`, `TreeB`, and `RedNode`; `balanceLB` is defined on `Red.AT` and

[17] Dependent object types [Amin *et al.*, 2012, 2014; Rompf and Amin, 2015; Amin *et al.*, 2016] may provide some good tools, but I have not found a chance to fully study them yet.

[18] See `https://github.com/sweirich/dth/blob/master/examples/zipwith/ctf.hs`.

[19] But there might be some bias since I know more about Scala than Haskell when I started.

[20] Arguably with the exception of path dependent types.

`Black.AT`. As a result, the specifications written in dependent types, also scatter everywhere. Certainly these methods can also be implemented as functions in Scala, but then it usually requires to use pattern matching, where local type inference can easily yield wrong type information, as we have shown in Section 5.2.

Subtyping relations can make the specifications even more difficult to read. For example, in our red-black tree example, a black node is defined as follow:

```
case class BlackNode[N <: Nat]
  (l: Tree[Color, N], v: Int, r: Tree[Color, N])
    extends TreeB[Succ[N]]
```

Under this definition, `BlackNode[Z.type]` actually means `Tree[Black.type, Succ[Z.type]]`! The specifications can easily steer away from a programmer's original intention, if she does not pay attention to these relations. For a more complicated example, imagine why the following example fails to type check:

```
sealed trait Color {
  type Incr[N <: Nat] <: Nat
}

sealed trait AlmostTree[N <: Nat] {
  ...
  def balanceRB(a: Tree[Color, N], x: Int): HiddenTree[Succ[N]]
}

case object Black extends Color {
  type Incr[N <: Nat] = Succ[N]
  case class AT[N <: Nat]
    (l: Tree[Color, N], v: Int, r: Tree[Color, N])
      extends AlmostTree[Incr[N]] {
    ...
    def balanceRB(a: Tree[Color, N], x: Int) =
      HB(BlackNode(a, x, BlackNode(l, v, r)))
  }
}
```

The reason is that `Black.AT[N]` is actually of type `AlmostTree[Succ[N]]`, so the type signature of `balanceRB`'s parameter `a` should be `Tree[Color, Succ[N]]`, or `Tree[Color, Incr[N]]` instead. This requires programmers to be careful when implementing these, and when things go wrong, the error information is usually not very helpful.

In contrast, the type annotations in Haskell are usually much more readable. We can easily see that from the type annotation of `balanceRB` written in Haskell:

```
balanceRB :: Tree c n -> Int -> AlmostTree n -> HiddenTree ('S n)
```

Another problem with Scala is when local type inference algorithm fails to infer the expected type, the error information can be confusing, especially for people who do not understand how the local type inference algorithm works (which is probably a lot of people), and the documentations and tutorials [21] on this are not very helpful.

References

Alexander Aiken and Edward L Wimmers. Type Inclusion Constraints and Type Inference. *FPCA*, pages 31–41, 1993.

Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, number EPFL-CONF-183030, 2012.

Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *Acm Sigplan Notices*, volume 49, pages 233–249. ACM, 2014.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World*, pages 249–272. Springer, 2016.

Edwin Brady and Miles Sabin. Scala vs. idris: Dependent types now and in the future. In *Strange Loop*, 2013.

Adam Chlipala. *Certified Programming with Dependent Types*, chapter Library MoreDep. The MIT Press, 2013.

Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2013.

Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, the University of Pennsylvania, 2016.

Dave Gurnell. *The Type Astronaut's Guide to Shapeless*. Underscore Consulting LLP, Brighton, U.K., 2016.

Robert Harper. *Practical Foundations for Programming Languages*, chapter System FPC for Recursive Types. Cambridge University Press, second edition, 2016.

Hiromi Ishii. Prove your haskell for great safety! part 1: Dependent types in haskell. `https://www.schoolofhaskell.com/user/konn/prove-your-haskell-for-great-safety/dependent-types-in-haskell#vectors---avoiding-boundary-error-using-type-system`, 2014.

[21] For example, neither Scala's official documents (`http://docs.scala-lang.org/tutorials/tour/local-type-inference.html`) nor related blog posts (for example, `http://pchiusano.blogspot.com/2011/05/making-most-of-scalas-extremely-limited.html`) are able to provide information to understand why the examples we have shown in Section 5.2 do not type check.

Andrew Kennedy and Claudio Russo. Generalized algebraic data types and object-oriented programming. pages 21–40. ACM New York, NY, USA, October 2005.

Stefan Monnier and David Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, pages 1–8. ACM, 2010.

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.

François Pottier. Simplifying Subtyping Constraints. *ICFP*, pages 122–133, 1996.

Tiark Rompf and Nada Amin. From f to dot: Type soundness proofs with definitional interpreters. *arXiv preprint arXiv:1510.05216*, 2015.

Stephanie Weirich. Depending on types. In *Code Mesh*, London, U.K., 2015.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.