

OPLSS'13: Programming and Proving in Agda

Daniel R. Licata Ian Voysey

July 26, 2013

Introduction

In these lectures, we'll discuss programming and proving in the dependently typed programming language Agda [4].

Agda is a pure functional programming language, which means it's like ML/Haskell, except you're not allowed to use any effects, including termination and non-exhaustive matching (or other exceptions). However, this is often not particularly burdensome, because Termination and exhaustiveness are checked “behind the scenes,” and the checkers are pretty good (also, you can always turn them off through a compiler flag if they get in the way).

In the first chapter, we'll see how to use Agda to write functional programs. Then, we'll introduce the idea of *dependent types*, which is where Agda goes beyond ML and Haskell by allowing types that mention programs. We'll see how dependent types can be used for program verification, using Okasaki's red-black trees [5] as an example.

In the second chapter, we'll look at one of the cool things you can do in a dependently typed language, which is using *universes* to build domain-specific programming languages. We'll look at a *data description language* [2, 6] and a *functor library*.

In the third chapter, we'll look at some examples of programming in an extension of Agda called *homotopy type theory*.

Agda is based on similar foundations to Coq, but the way of working with the system is somewhat different. While tactics play a central role in Coq, in Agda the emphasis is on writing proof terms directly—and Agda includes some features that Coq doesn't to make this pleasant. It's best not to go in trying to pick a winner between Agda and Coq, but to try to understand the advantages and disadvantages of both styles.

These notes assume you know simply-typed functional programming (ML or Haskell), and can work with bound variables and substitution. They also assume you have some familiarity with the *propositions-as-types* principle, where propositions are represented as types, where programs inhabiting those types correspond to proofs. I also assume you've seen a type theory with non-trivial *definitional equality*, induced by having some notion of reduction or computation in the type system (like System F^ω or a dependently typed language).

Please keep in mind that these notes are a work in progress, and may be updated as the lectures progress.

Chapter 1

Extrinsic and Intrinsic Verification

This chapter will serve as a first introduction to Agda programming, and to several important ideas in dependently typed programming:

- *Simply-typed functional programming* works much like it does in ML/Haskell, except all functions must be total (no effects, including exceptions and non-termination).
- *Inductive families of types* allow you to define predicates that are useful for program verification. *Dependent pattern matching* is a convenient way to work with inductive families.
- *Extrinsic verification* is a style of program verification where you first write a function, and then separately prove a spec about.
- *Intrinsic verification* is a style of program verification where the specification and verification is baked into the code—the *proof is the code*.
- Finally, *views* are a technique for flexible, user-defined case-analysis in a dependently typed language.

1.1 Simply-typed programming

First, let's see what simply-typed programming looks like in Agda. As an example, we'll use red-black tree insertion, following Okasaki [5]. Red-black trees are a balanced binary search tree.

Red-black trees are parametrized by the types of the keys and values stored in them. So we need a type `Key` of keys, and a type `Value` of values. But we'll also need to be able to test ordering between keys, and to do so we will define our first datatype.

1.1.1 Datatypes

In the companion code for these lectures (`Preliminaries.agda`), we define a datatype

```
data Order : Set where
  Less : Order
  Equal : Order
  Greater : Order
```

This datatype has three constructors (`Less`, `Greater`, `Equal`) that are used as values and in pattern-matching, exactly like you would expect from ML or Haskell.

`Set` is the Agda type classifying (small) types, so you use it where you would use `type` in ML or `*` in Haskell; more on `Sets` later. Note the alignment of the constructors: Agda uses whitespace-sensitive layout syntax, similar to Haskell. For example,

```

data Order : Set where
  Less : Order
  Equal : Order
  Greater : Order

```

won't parse.

1.1.2 Modules

To parameterize the red-black tree implementation by a type of keys with a comparison function and a type of values, we can make a module with three parameters:

```

module RBT (Key : Set) (compare : Key → Key → Order) (Value : Set) where

```

We won't use much of Agda's module system in these lectures. For now, all you need to know is

1. A module starts with

```

module ModuleName <params> where

```

and then the body of the module is indented. The parameters are a list of (identifier : classifier) pairs, and the identifiers are bound in the body.

2. From outside the module, you can refer to components as ModuleName.<component>. When a module has parameters, you can either supply the parameters to the module as a whole, such as

```

module IntStringDict = RBT Int Int.compare String

```

If you don't do this, then each component gets separately abstracted over the parameters. For example, if there is a function insert defined in RBT, you can say things like RBT.insert Int Int.compare String to apply insert to these three arguments, or you can use RBT.insert directly as a function of these three parameters.

We will also sometimes open a module, using a declaration

```

open <ModuleName>

```

which brings all of the names in <ModuleName> into scope.

3. There are a bunch of other bells and whistles in Agda's module system: **abstract** and **private** declarations for information hiding; **using**, **hiding**, **renaming**, **public** modifiers for massaging names; interplay between modules and records; ... We'll discuss those later on if they come up, or you can see the documentation on the Agda Wiki.

1.1.3 Functions

Figure 1.1 shows code for red-black tree insertion.

First, to represent Colors, we define a datatype with two constructors, Red and Black.

Second, we define a datatype representing trees. A Tree is either Empty, or it's a Node, with a left subtree, a color, a key-value pair, and a right subtree. When a datatype constructor has arguments, we give it a (usually curried) function type, which must end in the datatype being defined.

The type $A \times B$ is the type of pairs (a,b), which can be taken apart by pattern-matching, or by projection functions fst : $A \times B \rightarrow A$ and snd : $A \times B \rightarrow B$. Pairs are not built-in in Agda: they are defined in Preliminaries.agda as a datatype—and in fact it's an instance of a more general construction called a *dependent pair type*, but we'll get to that later. We could equivalently have given Node five arguments:

```

Node : Tree → Color → Key → Value → Tree → Tree

```

and avoided product types, but in the code it's sometimes convenient to keep the key and value together (see balance).

Next, we define four functions; let's read them from the bottom up:

- `insert` is the function that clients of the module should call. Functions in Agda are defined by giving clauses

```
<functionName> <pattern1> <pattern2> ...
```

with patterns for as many of the curried function arguments as you want (though this number has to be the same in all of the clauses for a given function). In this case, `insert` has one clause, with two patterns, both of which are variables. The body of the clause says that `insert` is defined by (1) calling the auxiliary function `ins`, which does the real work, and then (2) making the root of the resulting tree black, by calling `blackenRoot`.

- `blackenRoot` takes a tree and colors the root black. It's our first real example of pattern-matching. `blackenRoot` has two clauses. The first covers the case for the constructor `Empty`, in which case we return `Empty` again. The second case covers the constructor `Node` applied to arguments `l` and `r` and `kv` and `color`. The `color` says to ignore the color stored in the `Node`, because we won't use it. The result of the clause is a `Node` with the same key-value pair and left and right subtrees, and `Black` as the color.

Type checking: It almost goes without saying that all of this code is type-checked just like in ML or Haskell; for example, if we wrote

```
blackenRoot (Node l _ kv r) = Node Black kv r
```

Agda will highlight “`Black`” and say

```
Color !=< Tree of type Set
when checking that the expression Black has type Tree
```

That is, the type `Color` is not a subtype of the type `Tree`, which would be necessary to use `Black` as the first argument to `Node`. Subtyping in Agda is essentially for internal use in some aspects of the implementation, so you can read `=<` as type equality.

Exhaustiveness Checking: Agda checks the exhaustiveness of all functions—this is necessary to ensure that functions are total. For example, if we forgot the case for `Empty`, Agda would say

```
Incomplete pattern matching for blackenRoot. Missing cases:
  blackenRoot Empty
when checking the definition of blackenRoot
```

- `ins` does the recursive tree insertion: If the tree is `Empty`, we make a new `Node` with the key-value pair `kv`, `Empty` subtrees, and we color it `Red`. Otherwise, we are trying to insert `(k,v)` into a `Node` that has `(k',v')` at its root—note the use of **deep pattern matching** to break up the key-value pair sitting inside the `Node`. In this case, we compare the keys `k` and `k'`, and recursively insert into the left tree if `k` is less, or the right if `k` is greater, or put the new value at the node if they are the same. This is standard binary tree insertion, except instead of making a new `Node` using the recursive call, we call an auxiliary function `balance`, where `balance l c kv r` puts the arguments together into a `Node` in a clever way.

With: For now, you can think of **with** as being like `case/match` in ML/Haskell, with the following lines giving the clauses of the case-analysis on the comparison. More on **with** later.

Termination checking: Agda checks that every function is terminating. The basic criterion is that recursive calls must be made on structurally smaller terms—i.e. things you find beneath the constructors of inductive datatypes, like `l` and `r` here.

- `balance` performs tree rotations to keep the tree roughly balanced, using the colors to direct traffic. Whenever there is a black node whose child is a red node that itself has a red child, the tree is rotated and the root is colored red and the two children black. See Figure 1.2 for a picture. Using deep pattern matching, it's easy to describe these four cases and the result (which is the same in all four cases—we'll see a way to combine them into one case later on). In all other cases, we simply make a `Node`. Note the use of a catch-all case with variables, which covers all other cases—if we forget this catch-all, the exhaustiveness checker will report all 91 cases that we forgot.

```

data Color : Set where
  Red : Color
  Black : Color

data Tree : Set where
  Empty : Tree
  Node : Tree → Color → (Key × Value) → Tree → Tree

balance : Tree → Color → (Key × Value) → Tree → Tree
balance (Node (Node a Red x b) Red y c) Black z d = Node (Node a Black x b) Red y (Node c Black z d)
balance (Node a Red x (Node b Red y c)) Black z d = Node (Node a Black x b) Red y (Node c Black z d)
balance a Black x (Node (Node b Red y c) Red z d) = Node (Node a Black x b) Red y (Node c Black z d)
balance a Black x (Node b Red y (Node c Red z d)) = Node (Node a Black x b) Red y (Node c Black z d)
balance l c kv r = Node l c kv r

ins : Tree → (Key × Value) → Tree
ins Empty kv = Node Empty Red kv Empty
ins (Node l c (k',v') r) (k,v) with compare k k'
... | Equal = Node l c (k,v) r
... | Less = balance (ins l (k,v)) c (k',v') r
... | Greater = balance l c (k',v') (ins r (k,v))

blackenRoot : Tree → Tree
blackenRoot Empty = Empty
blackenRoot (Node l _ kv r) = Node l Black kv r

insert : Tree → Key × Value → Tree
insert t kv = blackenRoot (ins t kv)

```

Figure 1.1: Red-black tree insertion

1.2 Specification and Verification

OK, so red-black trees make perfect sense now, right? On to the next example!

The thing that's missing from the above simply-typed implementation is much indication of *why the code works*. From typing, we know that the functions always terminate successfully—they don't crash or loop—and that they produce values of the right shapes. But that alone doesn't distinguish a correct implementation of red-black trees from an incorrect one: If we forgot to balance, or to blackenRoot, or inserted into the wrong side, it would still type check. So what else might we want to know?

1. The implementation correctly implements a *dictionary*. E.g. if you lookup a key after inserting it, you get back the value you inserted.
2. `insert t kv` produces a sorted tree, where `Empty` is sorted; and `Node l c (k,v) r` is sorted if every key in `l` is less than or equal to `k`, which is less than or equal to every key in `r`, and `l` and `r` are sorted.
3. The implementation is fast.
4. The operations have good asymptotic complexity—e.g., inserting and looking up are logarithmic in the size of the tree.
5. `insert` preserves the red-black tree invariants:

A tree has **black height** `n` if every path from the root to `Empty` has exactly `n` black nodes (`Empty` is considered black).

A tree is **well-red** if both children of every red node are black.

My point is that there are many different specifications for red-black trees. The first two above are about *behavioral correctness*: The first says that the code implements the abstraction that it is supposed to implement (and this can be

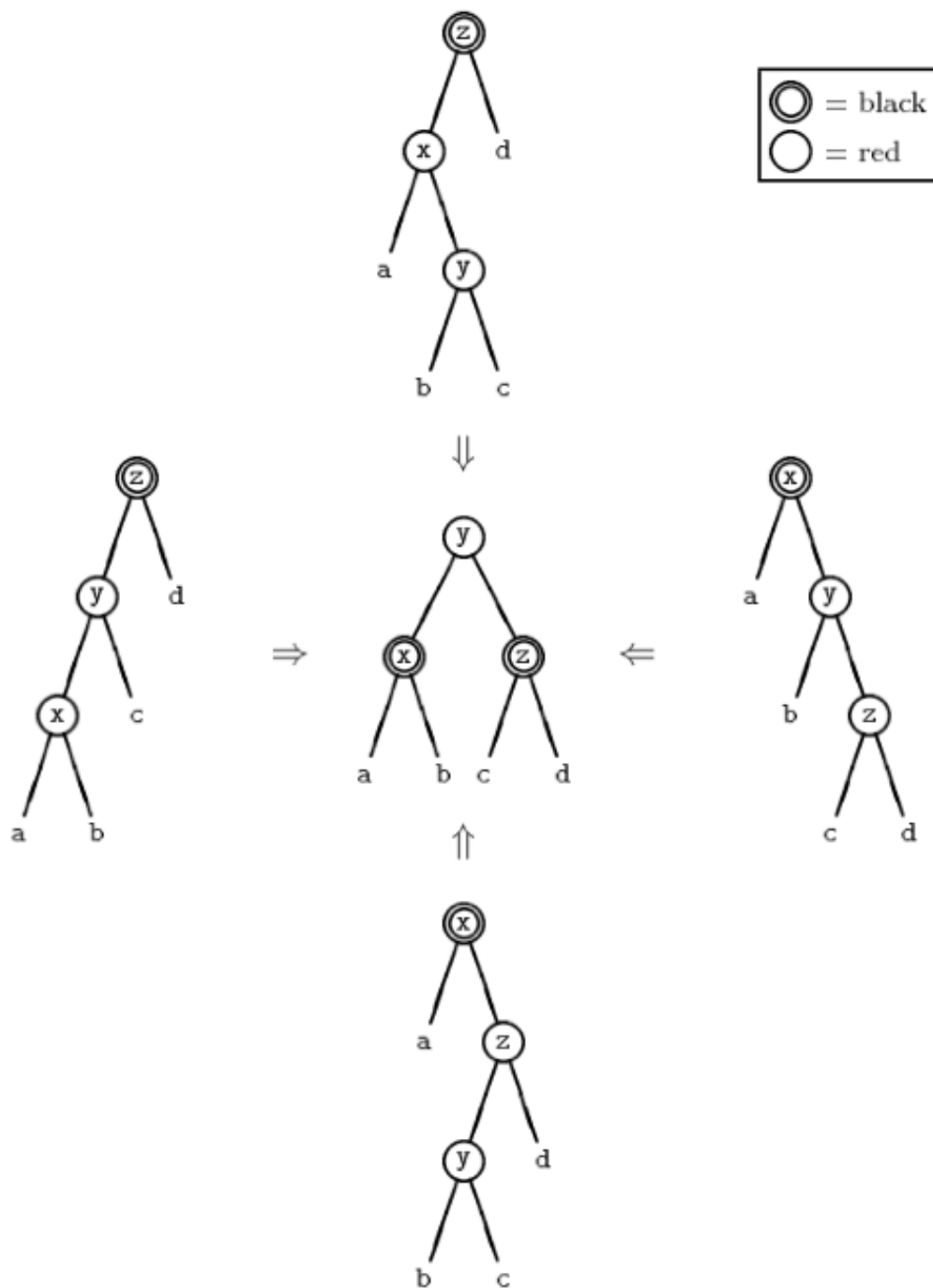


Fig. 1. Eliminating red nodes with red parents.

Figure 1.2: Red-black Tree Balancing, from Okasaki [5]

specified in many different ways—an axiomatic specification relating the operations; equivalence with a reference implementation). The second is an invariant that helps prove this kind of behavioral correctness: the reason you want a tree to be sorted is to justify the lookup function that uses the value at the root of a `Node` to determine whether to go left or right. The last three concern *efficiency*: what we really want is a fast implementation, and asymptotic complexity is often a good predictor of this. A nice property of red-black trees is that the asymptotic complexity can be factored into (1) showing that the operations preserve certain local, structural invariants (being well-red and having a black height) and (2) showing that these structural invariants imply that the tree is (roughly) balanced, and (3) showing that, for balanced trees, the operations (insert, lookup) have logarithmic complexity.

As we will soon see, dependently typed programming languages give you the tools to state and prove specifications about your code. But it’s important to keep in mind that you get to decide how much you want to verify—there is no monolithic notion of absolute correctness—to balance the costs and benefits of verification. There is a spectrum, from using Agda as a slightly more expressive ML/Haskell—where you get a little bit more information out of the type checker with little additional work—all the way to doing complete proofs of every spec you can think of.

We’ll use the asymptotic complexity of red-black tree insert as an example. It’s easy to see that insert (and lookup, defined in the obvious way) take time proportional to the depth of the tree, because they do a constant amount of work around one recursive call. Additionally, a little bit of math shows that if the longest path in a tree is no more than twice the length of the shortest path, then the depth is logarithmic (the tree can’t be that skewed). The red-black invariants imply this: because the tree has a black-height, each path from the root to a leaf has the same number of black nodes. Because no red node has a red child, you can, at most, alternate red and black nodes, so the length of every path from the root to a leaf is at most twice the black-height. Thus, it suffices to show that the operations on red-black trees preserve the black-height and well-red invariants.

To my mind, the above paragraph is a pretty clear and convincing argument that the invariants imply the time complexity, and is a good thing to include in a comment for the person reading your code. If you wanted to be more sure, you use Agda to check some or all of these steps. The tricky part of red-black trees is checking that the operations maintain the invariants. Moreover, if balance seemed mysterious to you before, this is the part that will explain why balance is written the way it is: understanding the invariants will help you understand the code. So, we’ll use Agda to verify these invariants.

1.3 Black Height, Extrinsically

As a first example, let’s verify that `insert t kv` maintains the black-height invariant: if `t` has a black height, then so does `insert t kv`. This example will be a bit slow-going, because we’ll use it to introduce most of Agda’s features. The complete proof is in Figure 1.3 if you want to see where we’re going.

We represent heights by unary natural numbers:

```
data Nat : Set where
  Z : Nat
  S : Nat → Nat
```

Agda has special syntax that allows numerals 0, 1, etc. can be used to write Nats. (E.g. 2 is syntax for `S (S Z)`).

1.3.1 Inductive Families

First, we need to define what it means for a tree to have a black height. In Agda, the nicest way to do this is with an *inductively defined family of types*, or *inductive family*. In this case, we define a family `HasBH t n`, where `t : Tree` and `n : Nat`—`t` and `n` are called the *indices* of the family `HasBH`. In Agda, this is represented by the following datatype definition:

```
data HasBH : Tree → Nat → Set where
  HBH-Empty : HasBH Empty 1
  HBH-Node-Red : (n : Nat) → (l : Tree) → (r : Tree) → (kv : Key × Value)
    → HasBH l n
    → HasBH r n
```



```

→ HasBH (Node l Red kv r) n
HBH-Node-Black : (n : Nat) (l r : Tree) (kv : Key × Value)
→ HasBH l n
→ HasBH r n
→ HasBH (Node l Black kv r) (S n)

```

The first line says that `HasBH` has type `Tree → Nat → Set`; i.e. for every `t` and `n`, `HasBH t n` is a `Set`. Next, we populate the datatype with three constructors, which are named `HBH-Empty` (“has-black-height empty”), `HBH-Node-Red` (“has-black-height node red”), and `HBH-Node-Black`. The type of `HBH-Empty` is `HasBH Empty 1`: this constructor asserts that the empty tree has black-height 1. The type of `HBH-Node-Red` is our first example of a **dependent function** type.

1.3.2 Dependent Function Types

A dependent function type $(x : A) \rightarrow B$ is well-formed when A is a `Set` and B is a `Set` under the assumption $x:A$; this is a generalization of the simple function type $A \rightarrow B$ in which the type B is allowed to depend on the argument. The inhabitants of the type are functions, which can be defined by named clausal function definitions, just like simply-typed functions—as we shall soon see. (Agda also has anonymous functions, but we won’t need those for a bit.) When you apply a dependent function $f : (x : A) \rightarrow B$ to an argument $a:A$, the result has type $B [a/x]$ — B with a substituted for x .

OK, back to the example. The type

```
(n : Nat) → (l : Tree) → (r : Tree) → (kv : Key × Value) → HasBH l n → HasBH r n → HasBH (Node l Red kv r) n
```

consists of 4 dependent function types followed by 2 simple function types. By convention, the body of a dependent function type extends as far to the right as possible, and simple function types are right-associated, so this is really

```
(n : Nat) → ((l : Tree) → ((r : Tree) → ((kv : Key × Value) → (HasBH l n → (HasBH r n → HasBH (Node l Red kv r) n))))))
```

This should be read as “`HBH-Node-Red` is a function, that, given a number `n`, trees `l` and `r`, and a key-value pair `kv`, along with a term of type `HasBH l n` and a term of type `HasBH r n`, returns a term of type `HasBH (Node l Red kv r) n`. Under props as types, you can read this as “for all `n`, `l`, and `kv`, if `HasBH l n` and `HasBH r n`, then `HasBH (Node l Red kv r) n`”—dependent functions are like a universal quantifier. But it’s important to keep in mind that (unlike inductive *props* in Coq) there is no difference in Agda between data and proofs: inhabitants of `HasBH t n` are data, just like trees.

The type of `HBH-Node-Black` is essentially the same, except we use it to illustrate a couple of syntactic simplifications: (1) When you have multiple dependent functions in a row $((x : A) \rightarrow (y : B) \rightarrow C)$, you can omit the \rightarrow : $((x : A) (y : B) \rightarrow C)$. (2) When you have multiple variables of the same type, you can list them together $((l r : Tree))$. Under these assumptions, `Node l Black kv r` has black-height `S n`—i.e. a black node has a black-height that is one more than the black height of its subtrees.

The reason `HasBH` is an inductively defined *family* of types is that all instances of `HasBH t n` must be defined together. In contrast, the usual definition of lists (`List A`) can be viewed as a *family of inductively defined types*: first you fix A , and then you define `List A`—for each A , there is a separate inductive definition of `List A`. Here, for example, `HasBH t (S n)` must be defined mutually with `HasBH t' n`, because to inhabit the former, you need elements of the latter.

1.3.3 Interactive Editing

Let’s prove that a tree has a black height. We’ll use this to introduce Agda’s interactive editor.

To get started, we’ll assume some keys `k1 k2` and some values `v1 v2`, and make a tree named `t`:

```

module Example (k1 k2 : Key) (v1 v2 : Value) where
  t : Tree
  t = Node (Node Empty Red (k1,v1) Empty) Black (k2,v2) Empty

```

Now we’d like to prove that `t` has a black-height, in this case 2. First, we write a line like this, where `t-bh` is defined to be `{!!}`:

```
t-bh : HasBH t 2
t-bh = {!!}
```

The `{!!}` is a *goal* that you can fill in using Agda’s editor (a goal can also be written `?`). First, **load** the file. The goal should turn blue and be labeled with the number 0:

At the bottom of the screen, in the Agda information window, you should see

```
?0 : HasBH t 2
```

which means “goal 0 needs to have type `HasBH t 2`”. If there were more goals, you would see them listed here too. There are various commands for working inside goals. Let’s try some of them.

- **load** (`C-c C-l`) the file first, to ask Agda to type check it.
- **goal-type** (`C-c C-t`) shows you the type of the current goal. For example:

```
HasBH (Node (Node Empty Red (k1,v1) Empty) Black (k2,v2) Empty) 2
```

goal-type normalizes the goal—here Agda has expanded the definition of `t`.

```
HasBH t 2
```

- **goal-and-context** (`C-c C-,`) shows you the type of the current goal, as well as the types of the variables in the context:

```
Goal: HasBH (Node (Node Empty Red (k1,v1) Empty) Black (k2,v2) Empty) 2
```

```
v2   : Value
v1   : Value
k2   : Key
k1   : Key
Value : Set
compare : Key → Key → Order
Key    : Set
```

- Now, type something in the goal, say `t`. **infer-type** will tell you the type of the expression in the goal, in this case `Tree`.
- **goal-and-context-and-inferred** (`C-c C-.`) will show you all of this information: the type of the goal, the context, and the type of the expression currently in the goal.

```
Goal: HasBH (Node (Node Empty Red (k1,v1) Empty) Black (k2,v2) Empty) 2
Have: Tree
```

```
v2   : Value
v1   : Value
k2   : Key
k1   : Key
Value : Set
compare : Key → Key → Order
Key    : Set
```

- **give** (`C-c C-space`) If the type of the expression in the goal matches the type of the goal, then replace the goal with the expression in it. Otherwise, you get a type error.

- **refine** (C-c C-r) Like **give** when the types match. Additionally, suppose the expression in the goal is a function that produces with something matching the goal. Then the goal will be replaced by the expression, applied to new goals for the necessary arguments. (Note: You could always use **give**, because the expression you **give** can contain new goals, but this way Agda infers how many goals to insert for you.) Also, if a solution is forced by typing (e.g. only one datatype constructor applies), Agda will fill it in.
- **make-case** (C-c C-c) case-analyze one of the variables in the function clause, replacing the clause with some number of specialized clauses. We will also call this *splitting* a variable.
- Don't normalize (C-u . . .): for many commands, including **goal-type**, **goal-and-context**, **infer-type**, and **goal-and-context-and-inferred** you can use a prefix argument (C-u) to see the same information, but without normalization. This is sometimes useful when the terms involved get big.

Let's use **give** and **refine** to prove that `t` has black-height 2. `t` is a `Black` node, so we need to start constructing the proof with `HBH-Node-Black`. This constructor has 6 arguments, so we can start writing by writing this in the goal:

```
t-bh : HasBH t 2
t-bh = {! HBH-Node-Black ? ? ? ? ? !}
```

Now **give**, and you will see that Agda makes 6 new goals:

```
t-bh : HasBH t 2
t-bh = HBH-Node-Black 1 (Node Empty Red (k1,v1) Empty) Empty (k2,v2) {!!} {!!}
```

and reports their types as follows:

```
?1 : Nat
?2 : Tree
?3 : Tree
?4 :  $\Sigma (\lambda \_ \rightarrow \text{Value})$ 
?5 : HasBH (Node Empty Red (k1,v1) Empty) 1
?6 : HasBH Empty 1
```

It's clear what the first four goals have to be, so we can fill them in:

```
t-bh : HasBH t 2
t-bh = HBH-Node-Black 1 (Node Empty Red (k1,v1) Empty) Empty (k2,v2) {!!} {!!}
```

Asking Agda for the type of the next goal, we see that it is

```
HasBH (Node Empty Red (k1,v1) Empty) 1
```

so we should start the proof with `HBH-Node-Red`. Rather than writing the six arguments explicitly, we can write `HBH-Node-Red` in the goal and then **refine**, at which point Agda generates the appropriate goals.

It's clear what the first four of the resulting goals should be:

```
t-bh : HasBH t 2
t-bh = HBH-Node-Black 1 (Node Empty Red (k1,v1) Empty)
      Empty (k2,v2)
      (HBH-Node-Red 1 Empty Empty (k1,v1) {!!} {!!})
      {!!}
```

Now all three remaining goals have type

```
?0 : HasBH Empty 1
?1 : HasBH Empty 1
?2 : HasBH Empty 1
```

so we can fill the first one with `HBH-Empty`:

```

t-bh : HasBH t 2
t-bh = HBH-Node-Black 1 (Node Empty Red (k1,v1) Empty) Empty (k2,v2)
      (HBH-Node-Red 1 Empty Empty (k1,v1) HBH-Empty {!!})
      {!!}

```

In fact, HBH-Empty is the only datatype constructor that applies, so if we just **refine** in the goal, Agda will fill it. Finally, we get the following proof:

```

t-bh : HasBH t 2
t-bh = HBH-Node-Black 1 (Node Empty Red (k1,v1) Empty) Empty (k2,v2)
      (HBH-Node-Red 1 Empty Empty (k1,v1) HBH-Empty HBH-Empty)
      HBH-Empty

```

Suppose we had made the left subtree Black instead:

```

t' : Tree
t' = Node (Node Empty Black (k1,v1) Empty) Black (k2,v2) Empty

```

Then we'd get into trouble because

```

t'-bh : HasBH t' 2
t'-bh = HBH-Node-Black 1 (Node Empty Black (k1,v1) Empty) Empty (k2,v2)
      (HBH-Node-Black 0 Empty Empty (k1,v1) {!!} {!!})
      HBH-Empty

```

```

?0 : HasBH Empty 0
?1 : HasBH Empty 0

```

we'd ultimately need to show that Empty has height 0, and there is no way to prove this.

1.3.4 Unification

It's pretty redundant to write the tree, nat, and key-value arguments to HBH-Node-Black and HBH-Node-Red, because *writing anything other than what we've written will result in a type error*: these arguments are forced by typing. Fortunately, Agda can infer them using unification. If part of a term can be inferred by unification, you can replace it with an `_`. Note that unification in Agda succeeds only when there is a unique solution (unlike Coq, which sometimes guesses). So we can simplify the above proof to:

```

t-bh' : HasBH t 2
t-bh' = HBH-Node-Black _ _ _ (HBH-Node-Red _ _ _ HBH-Empty HBH-Empty) HBH-Empty

```

1.3.5 Dependent Pairs

Next, we'll look at insert. The main thing we care about is that a tree has *some* black height; at the outside, we don't particularly care what the height is. To express "a tree with some black-height", we can use a type constructor called a *dependent pair type*. For reasons that we will explain in a bit, a dependent pair type is written $\Sigma (\lambda (x : A) \rightarrow B)$ where B can mention the variable a. The elements of a dependent pair type are pairs (a,b) where $a : A$ and $b : B [a/x]$. Dependent pairs can be taken apart by pattern matching, or by fst and snd projection. Second projection has a dependent type, expressing the dependency on the first projection:

```

fst : (Σ (λ (x : A) → B)) → A
snd : (p : Σ (λ (x : A) → B)) → B [fst p/a]

```

Logically, Σ is like an existential quantifier.

1.3.6 Pattern matching and inversion

Let's look at the spec for `blackenRoot` first: `blackenRoot` takes a tree with black height `n` and produces a tree with some black height. In fact, the black-height will be `n` (if it was already black) or `n+1` (if it was red), but we won't need to know this. So, we can state the following spec:

$$\text{blackenRoot-bh} : (t : \text{Tree}) (n : \text{Nat}) \rightarrow \text{HasBH } t \ n \rightarrow \Sigma (\lambda (m : \text{Nat}) \rightarrow (\text{HasBH } (\text{blackenRoot } t) \ m))$$

For all trees `t` with black height `n`, there exists an `m` such that `blackenRoot t` has black-height `m`. Now let's construct a proof of this fact—i.e. a function with this type. We'll start by making a goal:

$$\text{blackenRoot-bh } t \ n \ h = \{!!\}$$

The goal has type $\Sigma (n \rightarrow \text{HasBH } (\text{blackenRoot } t) \ n)$. How can we fill it in? The most direct way is to apply one of the constructors of `HasBH`. However, the constructors for `HasBH` all require that we know what constructor the tree is built from. Unfortunately, `blackenRoot t` is *stuck*—because `blackenRoot` is defined by pattern matching, it doesn't compute on a variable `t`. To get it to compute, we need to case-analyze `t`. We could fill in the cases ourselves, but it's easier to use Agda to do it. Type `t` in the goal and then **make-case**, and you'll get the following cases:

$$\begin{aligned} \text{blackenRoot-bh } \text{Empty} \ n \ h &= \{!!\} \\ \text{blackenRoot-bh } (\text{Node } t \ x \ t_1) \ n \ h &= \{!!\} \end{aligned}$$

Let's proceed with the `Empty` case first. The *un-normalized* type of the goal is

$$\Sigma (\lambda (m : \text{Nat}) \rightarrow \text{HasBH } (\text{blackenRoot } \text{Empty}) \ m)$$

But the normalized type of the goal is

$$\Sigma (\lambda (m : \text{Nat}) \rightarrow \text{HasBH } \text{Empty} \ m)$$

That is, *because `blackenRoot Empty` computes to `Empty` by the first clause of the definition of `blackenRoot`, Agda replaces one with the other*—you do not need to mark uses of definitional equality in your proof.

In general, when we need to give a term of a Σ -type, we can always introduce a pair of two new goals, and try solving them. We can do this by writing

$$\text{blackenRoot-bh } \text{Empty} \ n \ h = \{!?, ?!\}$$

and then refining to get

$$\text{blackenRoot-bh } \text{Empty} \ n \ h = \{!!\}, \{!!\}$$

Now, we need to fill in the first goal with some number, and to fill in the second goal with a proof that `blackenRoot Empty` has that number as its black-height. So we can fill the goals in with `1` and `HBH-Empty`.

$$\text{blackenRoot-bh } \text{Empty} \ n \ h = 1, \text{HBH-Empty}$$

Of course, we could also have let Agda fill in the `1`, because it's forced by the type of `HBH-Empty`:

$$\text{blackenRoot-bh } \text{Empty} \ n \ h = _, \text{HBH-Empty}$$

Note that we didn't need to use `h` in this case.

Now, we have to do the `Node` case, which we can start by introducing a pair, and then working on the second component of that pair.

$$\text{blackenRoot-bh } (\text{Node } l \ c \ kv \ r) \ n \ h = \{!!\}, \{!!\}$$

The second component has type `HasBH (Node l c kv r) (?? Key compare Value l c kv r n h)`, where `(?? ...)` is the goal for the first component of the pair (the arguments indicate which variables the first component is allowed to depend on). We can start with `HBH-Node-Black`, which is the only constructor that applies.

```
blackenRoot-bh (Node l c kv r) n h = S {!!}, HBH-Node-Black {!!} l r kv {!!} {!!}
```

If you look at the types of the last two goals, you'll see that what we need now is some number `m` that is the black-height of `l` and `r`. Where will we get the black-height of `l` and `r`? It has to come from `h`! `h` has type `HasBH (Node l c kv r) n`, so we don't know whether it was constructed by `HBH-Node-Red` or `HBH-Node-Black`.

At this point, we can proceed in one of two ways. The first is to *case analyze the tree and number enough that the constructor for `h` is forced*. Unfortunately, you can only **make-case** when the right-hand side is a single goal, so let's back out of `HBH-Node-Black` for now. First, we distinguish cases on `c`:

```
blackenRoot-bh (Node l Red kv r) n h = {!!}
blackenRoot-bh (Node l Black kv r) n h = {!!}
```

Now, in the `Red` case, `h` must be constructed with `HBH-Node-Red`, so if we **make-case**, Agda will not make a case for `HBH-Node-Black` or `HBH-Node-Empty`. In an informal proof by rule induction, you might say *by inversion, `h` must be constructed by `HBH-Node-Red`...*

```
blackenRoot-bh (Node l Red kv r) n (HBH-Node-Red .n .l .r .kv hl hr) = {!!}
blackenRoot-bh (Node l Black kv r) n h = {!!}
```

We also see our first example of **dot patterns**. A dot pattern `.e` means “this argument is forced to be the expression `e` by typing.” In this case, the first four arguments of `HBH-Node-Red` are forced, because in order for `HBH-Node-Red` to have the right type, they must be `n l r` and `kv`.

Now if you **goal-and-context**, you'll see that we've uncovered that `l` and `r` have black-height `n`:

```
hr : HasBH r n
hl : HasBH l n
```

So we can finish the case as follows:

```
blackenRoot-bh (Node l Red kv r) n (HBH-Node-Red .n .l .r .kv hl hr) = _, HBH-Node-Black _ _ _ hl hr
blackenRoot-bh (Node l Black kv r) n h = {!!}
```

Or, if you like, you can fill in all the underscores: `S n, HBH-Node-Black n l r kv hl hr`.

Now, the black case. For `HBH-Node-Black` to apply, the height `n` must be of the form `S something`. So let's case-analyze `n`:

```
blackenRoot-bh (Node l Red kv r) n (HBH-Node-Red .n .l .r .kv hl hr) = _, HBH-Node-Black _ _ _ hl hr
blackenRoot-bh (Node l Black kv r) Z h = {!!}
blackenRoot-bh (Node l Black kv r) (S n') h = {!!}
```

In the case for `S n'`, `h` must now be `HBH-Node-Black`, so we can invert `h`, and then fill in the goal:

```
blackenRoot-bh (Node l Red kv r) n (HBH-Node-Red .n .l .r .kv hl hr) = _, HBH-Node-Black _ _ _ hl hr
blackenRoot-bh (Node l Black kv r) Z h = {!!}
blackenRoot-bh (Node l Black kv r) (S n') (HBH-Node-Black .n' .l .r .kv hl hr) = _, HBH-Node-Black _ _ _ hl hr
```

What about the `Z` case? In this case, `h` has type `HasBH (Node l Black kv r) 0`. But there are no constructors inhabiting this type, because the only constructor for black nodes has a height `S _`, which is different than `Z`. So the assumed `h` is contradictory. If we **make-case**, then Agda will fill in an *absurd pattern* `()`, which means “this argument is contradictory.” When there is an absurd pattern, you leave off the right-hand side of the clause:

```
blackenRoot-bh (Node l Red kv r) n (HBH-Node-Red .n .l .r .kv hl hr) = _, HBH-Node-Black _ _ _ hl hr
blackenRoot-bh (Node l Black kv r) Z ()
blackenRoot-bh (Node l Black kv r) (S n') (HBH-Node-Black .n' .l .r .kv hl hr) = _, HBH-Node-Black _ _ _ hl hr
```

1.3.7 Dependent Pattern Matching

In the above proof of `blackenRoot-bh`, we *case-analyzed* simply-typed data (trees, colors, height), and then *inverted* dependently-typed data (`HasBH t n`) once we knew exactly which constructor applied. This is what you would have to do if, for example, your programming language erased the dependently typed data before run-time—e.g. if they were propositions in Coq. However, it makes the proof kind of convoluted: in the `Empty` case, we didn't look at `h` at all, and the `Node` case, we had to decide to case-analyze the color and the height.

Fortunately, in Agda, there is no distinction between data and proofs, and you can directly pattern-match on dependently typed data. This often leads to cleaner proofs, because *case-analyzing dependently typed data tells you information about the indices*.

Here's an alternate way to write `blackenRoot-bh`. Start at the very beginning:

```
blackenRoot-bh2 t n h = {!!}
```

then case-analyze `h`!

```
blackenRoot-bh2 .Empty .1 HBH-Empty = {!!}
blackenRoot-bh2 .(Node l Red kv r) .n (HBH-Node-Red n l r kv hl hr) = {!!}
blackenRoot-bh2 .(Node l Black kv r) .(S n') (HBH-Node-Black n' l r kv hl hr) = {!!}
```

Giving cases for the three constructors for `HasBH` *tells you* that the tree and height must be split into the three cases we had before, for `Empty`, a `Red Node`, and a `Black Node`, and in the last case the height must be `S n'`.

Then we can fill in the right-hand sides as before:

```
blackenRoot-bh2 .Empty .1 HBH-Empty = _, HBH-Empty
blackenRoot-bh2 .(Node l Red kv r) .n (HBH-Node-Red n l r kv hl hr) = _, HBH-Node-Black _ _ _ hl hr
blackenRoot-bh2 .(Node l Black kv r) .(S n') (HBH-Node-Black n' l r kv hl hr) = _, HBH-Node-Black _ _ _ hl hr
```

Case-analyzing dependently typed data often leads to much clearer proofs, because it tells you more.

Note that the above uses of "inversion" are actually special cases of dependent pattern-matching. For example, when we concluded that in the case

```
blackenRoot-bh (Node l Red kv r) n h = {!!}
```

that `HBH-Node-Red` is the *only* constructor that applies, so `h must be (HBH-Node-Red .n .l .r .kv hl hr)`, we were doing a special case of what Agda does all the time:

When you case-analyze a variable `x` whose type is an inductive family, Agda unifies the indices in the type of `x` with the indices of the constructors, and only gives you cases for the ones that match—and in these cases, the goal is specialized with the knowledge gained by unification. Here, `h : HasBH (Node l Red kv r) n`, so it asks

1. Can `HasBH Empty 1` equal `HasBH (Node l Red kv r) n`? No, because `Empty` is different than `Node`. Therefore no case for `HBH-Empty`.
2. Can `HasBH (Node l1 Black kv1 r1) (S n1)` equal `HasBH (Node l Red kv r) n`? No, because `Red` is different than `Black`. Therefore, no case for `HBH-Node-Black`.
3. Can `HasBH (Node l1 Red kv1 r1) n1` equal `HasBH (Node l Red kv r) n`? Yes, but only if `l1 = l`, `kv1 = kv`, `r1 = r`, and `n1 = n`. So we get the case

```
blackenRoot-bh (Node l Red kv r) n (HBH-Node-Red .n .l .r .kv h h1) = {!!}
```

Sometimes, like in the version of the proof where we matched on `h` first, when it has type `HasBH t n`, the indices of the variable are completely general. In this case, you get cases for all possible constructors. In other cases, some constructors are ruled out, but more than one applies—the two proofs we gave here are extremal points (one constructor / all constructors), but anything in between works too.

1.3.8 Implicit Arguments

In the above proof, Agda could always fill in the first four arguments to `HBH-Node-Red` and `HBH-Node-Black` by unification. When you expect a certain argument to often/always be filled in by unification, it's somewhat cumbersome to have to write the `_` each time. This can be solved by using *implicit arguments*. Instead of writing a dependent function type as $(x : A) \rightarrow B$, you can write $\{x : A\} \rightarrow B$ for an *implicit dependent function type*. There is no deep semantic difference between the two; it just changes the syntax.

For example, if we make `n` and `l` and `r` and `kv` implicit arguments to the constructors:

```
data HasBH : Tree → Nat → Set where
  HBH-Empty : HasBH Empty 1
  HBH-Node-Red : {n : Nat} {l r : Tree} {kv : Key × Value}
    → HasBH l n
    → HasBH r n
    → HasBH (Node l Red kv r) n
  HBH-Node-Black : {n : Nat} {l r : Tree} {kv : Key × Value}
    → HasBH l n
    → HasBH r n
    → HasBH (Node l Black kv r) (S n)
```

then

then `t-bh` looks like this:

```
t-bh : HasBH t 2
t-bh = HBH-Node-Black (HBH-Node-Red HBH-Empty HBH-Empty)
      HBH-Empty
```

By default, the arguments of an implicit dependent function are filled in by unification. If you want to supply them explicitly, you can do so using `{ }`; for example:

```
t-bh = HBH-Node-Black {1} { _ } {Empty} (HBH-Node-Red HBH-Empty HBH-Empty)
      HBH-Empty
```

As this example shows, when you have multiple curried implicit functions, you can explicitly apply them to as many arguments as you want, but you have to use `_`s if the arguments you want to supply come after arguments that you don't.

Here's how `blackenRoot-bh` looks if we make `t` and `n` implicit:

```
blackenRoot-bh : {t : Tree} {n : Nat} → HasBH t n → Σ (λ (m : Nat) → (HasBH (blackenRoot t) m))
blackenRoot-bh HBH-Empty = _, HBH-Empty
blackenRoot-bh (HBH-Node-Red hl hr) = _, (HBH-Node-Black hl hr)
blackenRoot-bh (HBH-Node-Black hl hr) = _, HBH-Node-Black hl hr
```

Much cleaner!

Sometimes you want to bind a name to an implicit argument in a function definition, because you need the name to define the function, but you expect it to be inferable at call sites. In this case, you can use `{ }` in the function clause:

```
blackenRoot-bh {t} {n} h = {!!}
```

Because you can explicitly supply implicit arguments when you want to, implicit arguments are pretty flexible—it's really just a question of whether you want an application by default or not.

1.3.9 Revising Definitions

It's a bit annoying that the code for `blackenRoot` has two cases (Empty, and Node of any color), whereas the proof has three cases (Empty, red Node, black Node). The reason is that, in the code, we don't pattern-match on the color `c`, but in the proof, we match on `HBH-Node-Red` and `HBH-Node-Black` to extract `hl` and `hr`. One solution would be to write

a helper function that pulled out these pieces. However, since this will come up again later in the proof, it's worth it to see if we can improve the definition. This is a common theme in formal proofs: *definitions matter a lot*. There are often many equivalent ways to define a concept, and when you're reasoning loosely or informally, you can sometimes be kind of sloppy about things. But for formal proofs, choosing the "right" definition can make a big difference in how much work it is to do your proof. It's usually a good idea to keep the different possibilities in mind and experiment a little as you go, because it's hard to pick the best definition a priori. Moreover, which kinds of definitions work well depends on what proof assistant you're using and how you're doing your proof—there is definitely some art to it.

In this case, what we'd like to do is to abstract over the differences between HBH-Node-Red and HBH-Node-Black. What are the differences? The former constructs a Red node and keeps the height of the result the same; the latter constructs a Black node and increments the height of the result. So, first, we can parametrize by a color c :

```
data HasBH : Tree → Nat → Set where
  HBH-Empty : HasBH Empty 1
  HBH-Node : {n : Nat} {c : Color} {l r : Tree} {kv : Key × Value}
    → HasBH l n
    → HasBH r n
    → HasBH (Node l c kv r) {!!}
```

Now, what we need to express is the relationship between the color c , the height n of the subtrees, and the height of the Node. We can do this using another inductive family:

```
data ColorHeight : Color → Nat → Nat → Set where
  CH-Red : {n : Nat} → ColorHeight Red n n
  CH-Black : {n : Nat} → ColorHeight Black n (S n)
```

ColorHeight gives two possibilities: the color is red, and the two numbers are the same; or the color is black, and the second is one more than the first. Now we can define

```
data HasBH : Tree → Nat → Set where
  HBH-Empty : HasBH Empty 1
  HBH-Node : {n m : Nat} {c : Color} {l r : Tree} {kv : Key × Value}
    → HasBH l n
    → ColorHeight c n m
    → HasBH r n
    → HasBH (Node l c kv r) m
```

Now blackenRoot-bh can be defined with only two clauses, because in the Node case we can choose not to match on the color argument:

```
blackenRoot-bh : {t : Tree} {n : Nat} → HasBH t n → Σ (λ (m : Nat) → (HasBH (blackenRoot t) m))
blackenRoot-bh HBH-Empty = _, HBH-Empty
blackenRoot-bh (HBH-Node hl ch hr) = _, (HBH-Node hl CH-Black hr)
```

1.3.10 More on pattern matching

OK, let's move on and look at a spec for `balance`. `balance l c kv r` is like making a `Node l c kv r`, except it sometimes rotates the tree. But in any case, it has the same black-height that `Node l c kv r` does. So `balance` has the following spec:

```
balance-bh : {l : Tree} {c : Color} {kv : (Key × Value)} {r : Tree} {n m : Nat}
  → HasBH l n → ColorHeight c n m → HasBH r n → HasBH (balance l c kv r) m
```

To prove this, we can match on the implicit arguments, with the same cases used to define `balance`. The first one is:

```

balance-bh { (Node (Node a Red x b) Red y c) } { Black } { z } { d }
  (HBH-Node (HBH-Node ha CH-Red hb) CH-Red hc) CH-Black hd =
  HBH-Node (HBH-Node ha CH-Black hb) CH-Red (HBH-Node hc CH-Black hd)

```

In this case, the call to `balance` in the goal computes to `(Node (Node a Black x b) Red y (Node c Black z d))`, and we can match on `HasBH l n` to extract the heights of `a` and `b` and `c`. Then we can put the heights of `a b c` and `d` together in the new rotation to prove the goal.

Unfortunately, the remaining cases are not so easy:

```

balance-bh { (Node a Red x (Node b Red y c)) } { Black } { z } { d } hl ch hr = {!!}
balance-bh { a } { Black } { x } { (Node (Node b Red y c) Red z d) } hl ch hr = {!!}
balance-bh { a } { Black } { x } { (Node b Red y (Node c Red z d)) } hl ch hr = {!!}
balance-bh hl ch hr = {!!}

```

If you look at the type of the goal in the first one, you'll see

```
HasBH (balance (Node a Red x (Node b Red y c)) Black z d) .m
```

Why didn't `balance` compute? The reason is that *not all function clauses are definitional equalities in Agda*. If you think about it for a second, it's obvious that this has to be true. For example, the final (catch-all) clause of `balance` overlaps with the other four, but gives a different result: so if both clauses held as equations, the behavior would be non-deterministic. Instead, Agda has a *first-match* semantics, like ML or Haskell, where the first matching equation is selected.

However, this means that in order for a function application to compute, Agda has to see that no previous clause applies. In this case, we when call `(balance (Node a Red x (Node b Red y c)) Black z d)` it's not clear whether the first clause applied—a might be a Red Node, or it might not. Thus, the application does not compute, even though the form it matches the second clause. Agda makes these decisions by internally representing pattern-matching as a case-tree, where (generally) matching is sequenced from left to right.

This complicates reasoning about functions like `balance`, which have overlapping clauses. One option is to expand the pattern-match far enough that things reduce, but this gets pretty ugly in this case. We'll see one solution later this lecture, and another next time.

1.3.11 More on with

Let's set aside `balance` for now and look at `ins`. To get the induction to go through, we need to know not only that `ins t kv` has a black-height, but that it's the same as the black-height of `t` (because we insert into only one side of the tree).

We can start by matching on `h`, and filling in the Empty case is easy:

```

ins-bh : { t : Tree } { kv : Key × Value } { n : Nat } → HasBH t n → HasBH (ins t kv) n
ins-bh HBH-Empty = HBH-Node HBH-Empty CH-Red HBH-Empty
ins-bh { _ } { k, v } (HBH-Node { n } { m } { c } { l } { r } { k', v' } hl ch hr) = {!!}

```

The type of the goal in the Node branch (where I've filled in some implicit args to name the variables) is something we haven't seen before:

```

HasBH (ins Key compare Value (Node l Red (k', v') r) (k, v) | compare k k')
  m

```

The first argument of `HasBH` is Agda's way of notating "the function `ins`, applied to the module arguments `Key`, `compare`, and `Value` (the parameters to the module, which are implicitly parameters to the function), and the normal arguments `(Node l Red (k', v') r)` and `(k, v)`, where the **with** case-analysis is on the expression `compare k k'`. You can think of **with** as adding another column to the pattern-matching, by saying what to case-analyze in that column. The bar is Agda's way of displaying what is being case-analyzed by a **with**. (In fact, **with** is implemented by making an anonymous function with an extra argument, and this is Agda's way of notating the application of that anonymous function. But for the most part you don't need to know that.)

The point is that, because `compare k k'` is treated like an extra argument to the function, and the function has separate clauses matching on whether this argument is `Less` or `Equal` or `Greater`, `ins` will not compute until we know which of these results `compare k k'` computes. That is, because the code case-analyzes `compare k k'`, we need to do the analogous case-analysis in the proof. Fortunately, we can use the same mechanism to do it: **with**!

```
ins-bh : {t : Tree} {kv : Key × Value} {n : Nat} → HasBH t n → HasBH (ins t kv) n
ins-bh HBH-Empty = HBH-Node HBH-Empty CH-Red HBH-Empty
ins-bh {._} {k,v} (HBH-Node {n} {m} {c} {l} {r} {k',v'} hl ch hr) with compare k k'
... | compare-result = {!!}
```

If you look at the the goal, you'll see that something slightly magical has happened:

```
Goal: HasBH (ins Key compare Value (Node l Red (k',v') r) (k,v) | compare-result)
      m
```

```
...
compare-result : Order
...
```

When you do a **with**, Agda replaces all occurrences (in the context and goal) of the expression being case-analyzed with the result of the **with**. In this case, this means that the one occurrence of `compare k k'` in the goal is replaced with the variable `compare-result`. (This expression doesn't occur in the context, so no replacement happens there. We didn't notice this feature of **with** when writing `ins` because `compare k k'` didn't appear in any types there, because all the code was simply-typed.) So now, if we case-analyze `compare-result`, `ins` will be applied to a constructor, and will reduce.

```
ins-bh : {t : Tree} {kv : Key × Value} {n : Nat} → HasBH t n → HasBH (ins t kv) n
ins-bh HBH-Empty = HBH-Node HBH-Empty CH-Red HBH-Empty
ins-bh {._} {k,v} (HBH-Node {n} {m} {c} {l} {r} {k',v'} hl ch hr) with compare k k'
... | Less = {!!}
... | Equal = {!!}
... | Greater = {!!}
```

For example, in the `Less` case, the goal is now `HasBH (balance (ins l (k,v)) Red (k',v') r),,` which we can solve by appealing to `balance-bh`, using the recursive call and `hl` and `hr`: The `Greater` case is analogous, the `Equal` case is just making a `Node`.

1.3.12 Tying it all together

With these lemmas in hand, proving the overall theorem is easy: if `t` has a black-height then `insert t kv` does too, by composing the lemmas about `blackRoot` and `ins`. Since the above development involved a bunch of false starts and iteration, to explain the various necessary Agda features, the complete proof (modulo the cases of `balance-bh` that we haven't done yet) is in Figure 1.3.

This development is an example of what we will call *extrinsic verification*: First, we wrote the code for red-black trees, as a traditional simply-typed program. Then, we wrote a separate proof of a correctness property of it, using dependent types to reason about the code.

1.3.13 Propositional Equality

One very important inductive family is *propositional equality*. This a type $M \equiv N$, where $M N : A$, representing equality of `M` and `N`. It is defined with one constructor, `Refl`:

```
data _==_ {A : Set} : A → A → Set where
  Refl : {M : A} → M ≡ M
```

Reflexivity says that $M \equiv M$.

```

data ColorHeight : Color → Nat → Nat → Set where
  CH-Red : {n : Nat} → ColorHeight Red n n
  CH-Black : {n : Nat} → ColorHeight Black n (S n)

data HasBH : Tree → Nat → Set where
  HBH-Empty : HasBH Empty 1
  HBH-Node : {n m : Nat} {c : Color} {l r : Tree} {kv : Key × Value}
    → HasBH l n
    → ColorHeight c n m
    → HasBH r n
    → HasBH (Node l c kv r) m

blackenRoot-bh : {t : Tree} {n : Nat} → HasBH t n → Σ \m → (HasBH (blackenRoot t) m)
blackenRoot-bh HBH-Empty = _, HBH-Empty
blackenRoot-bh (HBH-Node hl ch hr) = _, (HBH-Node hl CH-Black hr)

balance-bh : {l : Tree} {c : Color} {kv : (Key × Value)} {r : Tree} {n m : Nat}
  → HasBH l n → ColorHeight c n m → HasBH r n → HasBH (balance l c kv r) m
balance-bh { (Node (Node a Red x b) Red y c) } { Black } { z } { d }
  (HBH-Node (HBH-Node ha CH-Red hb) CH-Red hc) CH-Black hd =
  HBH-Node (HBH-Node ha CH-Black hb) CH-Red (HBH-Node hc CH-Black hd)
balance-bh { (Node a Red x (Node b Red y c)) } { Black } { z } { d } hl ch hr = {!!}
balance-bh { a } { Black } { x } { (Node (Node b Red y c) Red z d) } hl ch hr = {!!}
balance-bh { a } { Black } { x } { (Node b Red y (Node c Red z d)) } hl ch hr = {!!}
balance-bh hl ch hr = {!!}

ins-bh : {t : Tree} {kv : Key × Value} {n : Nat} → HasBH t n → HasBH (ins t kv) n
ins-bh HBH-Empty = HBH-Node HBH-Empty CH-Red HBH-Empty
ins-bh { _ } { k, v } (HBH-Node { n } { m } { c } { l } { r } { k', v' } hl ch hr) with compare k k'
... | Less = balance-bh (ins-bh hl) ch hr
... | Equal = HBH-Node hl ch hr
... | Greater = balance-bh hl ch (ins-bh hr)

insert-bh : {n : Nat} {t : Tree} {kv : (Key × Value)} → HasBH t n → Σ (λ m → (HasBH (insert t kv) m))
insert-bh h = blackenRoot-bh (ins-bh h)

```

Figure 1.3: Black Height, Extrinsically

Propositional equality can be used to prove equational specifications about functions; you'll see this in lab. We will also sometimes use it to write tests.

Another use of propositional equality is computational definitions. For example, here's another way to define `HasBH t n`: First, we define a function computing the black-height of a tree. Then we say that a tree has a black-height if that function returns `Some`:

```
incr-if-black : Color → Nat → Nat
incr-if-black Red x = x
incr-if-black Black x = S x

black-height : Tree → Maybe Nat
black-height Empty = Some 0
black-height (Node l c _ r) with (black-height l) | (black-height r)
... | Some lh | Some rh with Nat.equal lh rh
... | True = Some (incr-if-black c lh)
... | False = None
black-height (Node l c _ r) | _ | _ = None

HasBH : Tree → Nat → Set
HasBH t n = black-height t ≡ Some n
```

These kinds of definitions can sometimes be useful for automating proofs using computation (like the `ssreflect` library in Coq). However, the style of proofs that we did above, which used dependent pattern matching, work better when properties are defined via inductive families.

1.4 Black Height, Intrinsically

Compare the code in Figure 1.1 with the proof in Figure 1.3. What you should notice is that *the proof has exactly the same structure as the code!* There is one lemma in the proof for each function in the code, and the proof of each lemma exactly mirrors the corresponding function: the proof manipulates the derivations of `HasBH t n` in exactly the same way that the code manipulates trees. This suggests that this example is ripe for *intrinsic verification*: rather than first writing the code, and then doing a proof about it, we can bake the proof into the code.

Instead of first defining trees, and then defining `HasBH t n`, we just define *trees of height n* as the basic data structure. Colors become indexed by the "input" and "output" size, just like `ColorHeight` is. Then, we take the black-height specs that we *proved* above, and make them the *types* of the functions. Then, except for some `_`'s because of the Σ -type in the result of `blackenRoot`, *the exact same code as above type-checks with these fancier types*. Moreover, we avoid the issue where balance was hard to reason about, because of its pattern-matching, because we never reason *about* the reduction behavior of the code. The reason this works out so nicely is that the proof has the same structure as the code, so we can merge the two without changing the code.

There are pros and cons of intrinsic and extrinsic verification:

- **Finding bugs** With intrinsic verification, the type checker will help you find bugs as you write the code. For example, mis-coloring the result of `balance`

```
balance (Node (Node a Red x b) Red y c) Black z d = Node (Node a Black x b) Black y (Node c Black z d)
```

will be a type error

```
S .n != .n of type Nat
when checking that the expression
Node (Node a Black x b) Black y (Node c Black z d) has type
Tree (S .n)
```

Extrinsically, you'll find the same bug if you try to do the proof as you write the code:

```
balance-bh { (Node (Node a Red x b) Red y c) } { Black } { z } { d }
  (HBH-Node (HBH-Node ha CH-Red hb) CH-Red hc) CH-Black hd =
  HBH-Node (HBH-Node ha CH-Black hb) XXX (HBH-Node hc CH-Black hd)
```

```

data Color (n : Nat) : Nat → Set where
  Red : Color n n
  Black : Color n (S n)

data Tree : Nat → Set where
  Empty : Tree 1
  Node : {n m : Nat} → Tree n → (c : Color n m) → (Key × Value) → Tree n → Tree m

balance : {n m : Nat} → Tree n → (c : Color n m) (kv : (Key × Value)) → Tree n → Tree m
balance (Node (Node a Red x b) Red y c) Black z d = Node (Node a Black x b) Red y (Node c Black z d)
balance (Node a Red x (Node b Red y c)) Black z d = Node (Node a Black x b) Red y (Node c Black z d)
balance a Black x (Node (Node b Red y c) Red z d) = Node (Node a Black x b) Red y (Node c Black z d)
balance a Black x (Node b Red y (Node c Red z d)) = Node (Node a Black x b) Red y (Node c Black z d)
balance l c kv r = Node l c kv r

ins : {n : Nat} → Tree n → (Key × Value) → Tree n
ins Empty kv = Node Empty Red kv Empty
ins (Node l c (k',v') r) (k,v) with compare k k'
... | Equal = Node l c (k,v) r
... | Less = balance (ins l (k,v)) c (k',v') r
... | Greater = balance l c (k',v') (ins r (k,v))

blacken-root : {n : Nat} → Tree n → Σ (λ (m : Nat) → Tree m)
blacken-root Empty = _, Empty
blacken-root (Node l _ kv r) = _, Node l Black kv r

insert : {n : Nat} → Tree n → Key × Value → Σ (λ (m : Nat) → Tree m)
insert t kv = blacken-root (ins t kv)

```

Figure 1.4: Black Height, Intrinsically

Here, XXX must have type `ColorHeight Black (S n) (S n)`.

- **How easy is it to write?** Intrinsic verification often leads to a shorter overall artifact. On the other hand, sometimes writing the code and proof at once takes more thought than writing the code, and then banging out the proof.
- **How easy is it to read?** Sometimes, intrinsically verified code is easier to read, because you see the code and its properties at once—especially when the intrinsically verified code is pretty close to the code you would have written anyway, like above. However, sometimes the proof has a very different structure than the code, so when you combine them, the combined artifact is harder to read.
- **Separation of properties** Extrinsically, you can prove separate properties separately—for example, if we want to verify the well-red invariant, or sortedness, we can do a completely separate proof about the same piece of code. Intrinsically, there is only one artifact, so we would have to change the above code to include some additional verification.
- **Efficiency** Extrinsically, you can write the code however you want, and the proofs, or the data necessary to do the proofs (like the black-height), do not influence how it runs. However, even extrinsically, it's often the case that writing code with the proofs in mind will make the overall effort easier—sometimes it's worth making the code a bit less efficient to make it easier to reason about.

Intrinsically, you're running the proof, so the data and control-flow necessary for the proof influences the efficiency of your code. In the example above, there are additional natural numbers (the black height) carried around at run time—unless the compiler optimizes them away [1]. In the above example, we didn't need to change the code, but sometimes the proof requires a slightly different structure, as we'll see below.

Both are important tools to have in your toolbox; picking between them is kind of an art.

1.5 Well-Red, Extrinsically

Next, let's look at the well-red invariant, extrinsically and intrinsically. There are a few lessons from this example: In the extrinsic verification, we'll see an example of a common technique called *views*, and use this to solve the problem with verifying balance that we had above. The intrinsic verification will be a good opportunity to revisit the trade-offs listed above, because the proof will change the structure of the code somewhat.

1.5.1 Views

Recall the issue with balance that we had above: it was hard to reason about because of overlapping patterns. In this section, we'll see a way to get around this problem—which is not perfect, but at least gets the job done. The first step is to make the code for balance not rely on overlapping patterns. We do this by defining a *view*, an auxiliary datatype that describes the relevant cases. The two cases relevant to balance are (1) there is a violation, or (2) there is no violation. In each case, we will record the data that we used to construct the result of balance above: When there is a violation, we need to record the subtrees/key-values $a \times b \ y \ c \ z \ d$. When there is no violation, we record the $l \ c \ kv \ r$ that were passed in:

```
data BalanceView : Set where
  Violation : Tree → (Key × Value) → Tree → (Key × Value) → Tree → (Key × Value) → Tree → BalanceView
  OK : Tree → Color → Key × Value → Tree → BalanceView
```

Now, given $l \ co \ kv \ r$, we can detect whether there was a violation with the same case-analysis that we used above in balance.

```
view : Tree → Color → (Key × Value) → Tree → BalanceView
view (Node (Node a Red x b) Red y c) Black z d = Violation a x b y c z d
view (Node a Red x (Node b Red y c)) Black z d = Violation a x b y c z d
view a Black x (Node (Node b Red y c) Red z d) = Violation a x b y c z d
view a Black x (Node b Red y (Node c Red z d)) = Violation a x b y c z d
view l co kv r = OK l co kv r
```

And, now balance has non-overlapping patterns: if there is a violation, we rotate; otherwise, we make a Node:

```
balance : Tree → (c : Color) (kv : (Key × Value)) → Tree → Tree
balance l co kv r with view l co kv r
... | Violation a x b y c z d =
  Node (Node a Black x b) Red y (Node c Black z d)
... | OK l' co' kv' r' = Node l' co' kv' r'
```

Of course, you might object that we've just squeezed the balloon: balance avoids overlapping patterns, but view uses them! This is true, but this balloon-squeezing is helpful, because it localizes the annoying reasoning to view, which will be easier than doing it directly in balance.

The rest of the code (ins, blackenRoot, insert) is unchanged from above:

1.5.2 Correctness of the view

First we need a spec for view. If you think about the code carefully, it obeys the following spec. Either

- view $l \ co \ kv \ r$ returns Violation $a \times b \ y \ c \ z \ d$, where the node made from $l \ co \ kv \ r$ is one of the red-red violations below a black root (e.g., l is (Node (Node a Red x b) Red y c), co is Black, kv is z and r is d)
- view $l \ Black \ kv \ r$ returns OK $l \ Black \ kv \ r$, and there is no red-red violation at the root of l or r (because such a red-red violation would be flagged as a violation).
- view $l \ Red \ kv \ r$ returns OK $l \ Red \ kv \ r$, and there might be a red-red violation (because such violations at the root are not flagged).

For now, it's OK if it's a bit of a mystery why the cases that `view` divides things into are useful; to really understand them, you need understand how they fit with the spec for `balance` and ins below.

Let's proceed and describe these possibilities using some inductive families. The first, `ViolationType a x b y c z d l co kv r`, has four constructors, one for each possible orientation of the violation. For example, `Violation1` corresponds to the first violation, where `l` is `(Node (Node a Red x b) Red y c)`, `co` is `Black`, `kv` is `z` and `r` is `d`.

```
data ViolationType (a : Tree) (x : Key × Value) (b : Tree) (y : Key × Value) (c : Tree) (z : Key × Value) (d : Tree) :
  Tree → Color → Key × Value → Tree → Set where
  Violation1 : ViolationType a x b y c z d (Node (Node a Red x b) Red y c) Black z d
  Violation2 : ViolationType a x b y c z d (Node a Red x (Node b Red y c)) Black z d
  Violation3 : ViolationType a x b y c z d a Black x (Node (Node b Red y c) Red z d)
  Violation4 : ViolationType a x b y c z d a Black x (Node b Red y (Node c Red z d))
```

Note that the first six arguments are *the same* in each constructor, so we can put them to the left of the colon, which brings them into scope for the constructors. These are called *parameters* of the type family. The next four arguments are *different* in different constructor; these are called *indices*. The fact that the indices can be different in each constructor is what enables one to represent equality constraints with inductive families. For example, case-analyzing a value of type `ViolationType a x b y c z d l co kv r` will give you information about `l`, `co`, `kv`, and `r`.

Next, we need to specify what it means for there to be no red-red violation at the root of a node. First, we define an inductive family specifying the color of a node (`Empty` is considered black):

```
data RootColored : Tree → Color → Set where
  RC-Empty : RootColored Empty Black
  RC-Red : {l : Tree} {kv : Key × Value} {r : Tree} → RootColored (Node l Red kv r) Red
  RC-Black : {l : Tree} {kv : Key × Value} {r : Tree} → RootColored (Node l Black kv r) Black
```

Then, the root of a tree is okay if it's empty; or if it's a black node; or if it's a red node and the subtrees are black:

```
data RootOK : Tree → Set where
  ROK-Empty : RootOK Empty
  ROK-Black : {l : Tree} {kv : Key × Value} {r : Tree} → RootOK (Node l Black kv r)
  ROK-Red : {l : Tree} {kv : Key × Value} {r : Tree} → RootColored l Black → RootColored r Black
    → RootOK (Node l Red kv r)
```

Finally, we define an inductive family `ViewInvariant l co kv r bv`, with one constructor for each of the three possibilities mentioned above: when there is a black root with a red-red violation below it, when there is a black root with no red-red violations, and when there is a red root:

```
data ViewInvariant : (l : Tree) (co : Color) (kv : Key × Value) (r : Tree) → BalanceView → Set where
  ViolationInv : ∀ {l co kv r a x b y c z d} → ViolationType a x b y c z d l co kv r → ViewInvariant l co kv r (Violation a x b y c z d)
  OKBlackInv : ∀ {l kv r} → RootOK l → RootOK r → ViewInvariant l Black kv r (OK l Black kv r)
  OKRedInv : ∀ {l kv r} → ViewInvariant l Red kv r (OK l Red kv r)
```

The syntax $\forall \{l kv r\} \rightarrow \dots$ is shorthand for $\{l : _ \} \{kv : _ \} \{r : _ \} \rightarrow \dots$. It's convenient when there are a bunch of quantifiers in a row and you want to omit the type annotations on all of them.

To prove that `view` satisfies this invariant, we show that `ViewInvariant` holds for all arguments `l`, `co`, `kv`, and `r`:

```
view-ok : (l : Tree) (co : Color) (kv : Key × Value) (r : Tree) → ViewInvariant l co kv r (view l co kv r)
```

The proof is rather ugly (133 cases!), because there is a lot of duplication necessary to expand things enough that `view` computes (see the discussion about `balance` above). But the cases can be generated by Agda, and the right-hand side is a trivial application of constructors, which can be constructed automatically by Agda, using mechanisms that we haven't talked about yet. The benefit of making a view datatype, as opposed to doing the same case-analysis directly in `balance`, is that we need to do this ugly proof only once, and can henceforth reason in terms of the the view's invariant. For example, we will be able to use this proof to prove the well-red invariant, the black-height invariant, and the sortedness invariant.

1.5.3 Insert is Well-Red

Next, we'd like to prove that the result of insert is well-red, assuming the input is. Recall that a tree is well-red if both children of every red node are black. We can specify this more formally as follows: the empty tree is well-red; a black node is well-red if its subtrees are; a red node is well-red if its subtrees are, and the roots of both are black.

```

data WellRed : Tree → Set where
  WR-Empty : WellRed Empty
  WR-Black : {l : Tree} {kv : Key × Value} {r : Tree}
    → WellRed l
    → WellRed r
    → WellRed (Node l Black kv r)
  WR-Red : {l : Tree} {kv : Key × Value} {r : Tree}
    → WellRed l
    → WellRed r
    → RootColored l Black
    → RootColored r Black
    → WellRed (Node l Red kv r)

```

Now, we'd like to prove

```

insert-wellred : {t : Tree} {kv : Key × Value}
  → WellRed t
  → WellRed (insert t kv)

```

But to prove this, we'll need to invent some lemmas about `blackenRoot` and `ins` and `balance`, and these lemmas are somewhat subtle. For example, it's not the case that `ins t kv` always returns a well-red tree! As a counterexample, we can instantiate the module mapping `Nats` to `Unit`, and construct the tree

```

t =
Node
(Node (Node Empty Red (1,<>) Empty)
  Black (2,<>)
  Empty)
Red (3,<>)
(Node Empty Black (4,<>) Empty)

```

(note that this tree is well-red and has a black-height of 2). Then `ins t (0,<>)` returns the following tree:

```

Node
(Node (Node Empty Black (0,<>) Empty)
  Red (1,<>)
  (Node Empty Black (2,<>) Empty))
Red (3,<>)
(Node Empty Black (4,<>) Empty)

```

which has a red-red violation! The tricky part of red-black trees is that rotating the tree to balance it sometimes creates red-red violations—but these then get fixed up.

So we need to figure out the discipline to this process: why do the violations always get fixed? The essence of the idea is that whenever there is a black grandparent with a red child that itself has a red child, we fix the red-red violation (these are the cases of `balance`). But when there is a red node that acquires a new grandchild, we sometimes create a red-red violation. But this violation will be fixed by its parent, which must be black (because the input is well-red). What if it doesn't have a parent, because it's the root? Then it will be fixed by blackening the root. This logic—the clever part of red-black trees—is somewhat obscured in the above code, but the proof really clarifies it.

Because the responsibility for fixing a red-red violation lies with the enclosing black grandparent, these violations will get fixed as the recursion in `ins` percolates up the tree. The only place where they won't get fixed is at the root, because there might not be a black grandparent. Thus, we need to describe a tree that has no red-red violations, except possibly at the root. We say tht such a tree is *almost well-red*:

```

data AlmostWellRed : Tree → Set where
  AWR-Empty : AlmostWellRed Empty
  AWR-Node : {l : Tree} {kv : Key × Value} {r : Tree} {c : Color}
    → WellRed l
    → WellRed r
    → AlmostWellRed (Node l c kv r)

```

The empty tree is almost well-red; a node (of any color) is well-red if its subtrees are well-red.

It's immediate that an almost well-red tree whose root has no violation, in the sense defined above, is in fact well-red; and that any well-red tree is almost well-red:

```

combine : {t : Tree} → RootOK t → AlmostWellRed t → WellRed t
combine ROK-Empty a = WR-Empty
combine ROK-Black (AWR-Node wl wr) = WR-Black wl wr
combine (ROK-Red rl rr) (AWR-Node wl wr) = WR-Red wl wr rl rr

forget : {t : Tree} → WellRed t → AlmostWellRed t
forget WR-Empty = AWR-Empty
forget (WR-Red wl wr _ _) = AWR-Node wl wr
forget (WR-Black wl wr) = AWR-Node wl wr

```

Correctness of ins Based on the above discussion, we can start by trying to prove the following two invariants: First, if you ins into a well-red black tree, then you get a well-red tree, because any red-red violations will ultimately be fixed. Second, if you ins into a red well-red tree, then the result will be almost well-red. We'll need some lemmas about balance to prove this, but we can discover them as we go.

```

ins-black : (t : Tree) (kv : Key × Value) → RootColored t Black → WellRed t → WellRed (ins t kv)
ins-red : (t : Tree) (kv : Key × Value) → RootColored t Red → WellRed t → AlmostWellRed (ins t kv)

```

Let's tackle ins-black first. First, we can split on the proof that t is well-red. Then it's easy to fill in the case for Empty, and the case for WR-Red is impossible because of rt.

```

ins-black .Empty kv rt WR-Empty = WR-Red WR-Empty WR-Empty RC-Empty RC-Empty
ins-black .(Node l Red (k',v') r) kv () (WR-Red {l} {k',v'} {r} wl wr rl rr)
ins-black .(Node l Black (k',v') r) (k,v) rt (WR-Black {l} {k',v'} {r} wl wr) = {!!}

```

In the WR-Black case, we need to show WellRed (ins (Node l Black (k',v') r) (k,v)), but computationa is stuck on compare k k' because of the **with**. So we need to do a **with** in the proof:

```

ins-black .Empty kv rt WR-Empty = WR-Red WR-Empty WR-Empty RC-Empty RC-Empty
ins-black .(Node l Red (k',v') r) kv () (WR-Red {l} {k',v'} {r} wl wr rl rr)
ins-black .(Node l Black (k',v') r) (k,v) rt (WR-Black {l} {k',v'} {r} wl wr) with compare k k'
... | Less = {!!}
... | Equal = WR-Black wl wr
... | Greater = {!!}

```

The Equal case is obvious, because the structure of the tree doesn't change. In the Less case, we need to show that balance (ins l (k,v)) Black (k',v') r is well-red. This will clearly require a lemma about balance, as well as an inductive hypothesis about ins l (k,v).

However, we don't know whether the root of l is black or red, so it's unclear whether the IH will be a call to ins-black or ins-red. Fortunately, it doesn't matter: it will be enough to know that ins l (k,v) is almost well-red, which is true whether the root is red (because that's what ins-red says) or whether the root is black (because we can forget that the tree is well-red, and remember only that it's almost well-red).

```

decide-root : (t : Tree) → Either (RootColored t Black) (RootColored t Red)
decide-root Empty = Inl RC-Empty
decide-root (Node _ Red _ _) = Inr RC-Red

```

```

decide-root (Node _ Black _ _) = Inl RC-Black
ins-any : (t : Tree) (kv : Key × Value) → WellRed t → AlmostWellRed (ins t kv)
ins-any t kv wt with decide-root t
... | Inl rt = forget (ins-black t kv rt wt)
... | Inr rt = ins-red t kv rt wt

```

This suggests the first lemma about balance that we'll need: by ins-any, ins l (k,v) will be almost well-red, and r is well-red, so it suffices to show:

balance-fix-left: If l is almost well-red, and r is well-red, and co is Black, then balance l co kv r is well-red.

That is, when the root is black, and one side is almost well-red, and the other is well-red, then balance will do a tree rotation to fix the violation.

We defer the proof of this lemma, and a symmetric one where r is almost well-red and l is well-red:

```

balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree)
  → AlmostWellRed l
  → WellRed r
  → WellRed (balance l Black kv r)
balance-fix-right : (l : Tree) (kv : Key × Value) (r : Tree)
  → WellRed l
  → AlmostWellRed r
  → WellRed (balance l Black kv r)

```

and complete the proof of ins-black:

```

ins-black .Empty kv rt WR-Empty = WR-Red WR-Empty WR-Empty RC-Empty RC-Empty
ins-black .(Node l Red kv' r) kv () (WR-Red {l} {kv'} {r} y y' y0 y1)
ins-black .(Node l Black (k',v') r) (k,v) rt (WR-Black {l} {k',v'} {r} wl wr) with compare k k'
... | Less = balance-fix-left (ins l (k,v)) (k',v') r (ins-any l (k,v) wl) wr
... | Equal = WR-Black wl wr
... | Greater = balance-fix-right l (k',v') (ins r (k,v)) wl (ins-any r (k,v) wr)

```

On to ins-red. Split on the proof that t is well-red, and then it's easy to fill in the Empty case, and to contradict the Black case. In the Red case we need to look at compare k k' for the same reason as above, and the Equal case is easy.

```

ins-red : (t : Tree) (kv : Key × Value) → RootColored t Red → WellRed t → AlmostWellRed (ins t kv)
ins-red .Empty kv rc WR-Empty = AWR-Node WR-Empty WR-Empty
ins-red .(Node l Black kv' r) kv () (WR-Black {l} {kv'} {r} y y')
ins-red .(Node l Red (k',v') r) (k,v) rc (WR-Red {l} {k',v'} {r} wl wr rl rr) with compare k k'
... | Less = {!!}
... | Equal = AWR-Node wl wr
... | Greater = {!!}

```

In the Less case, we need to show

AlmostWellRed (balance Key compare Value (ins l (k,v)) Red (k',v') r)

```

rr : RootColored r Black
rl : RootColored l Black
wr : WellRed r
wl : WellRed l

```

That is, because the root is red, in this case, we know that l and r are both black. Therefore we can use the inductive call ins-black to conclude that ins l (k,v) is well-red, and we already know that r is well-red. Thus, the final lemma that we need about balance is as follows:

```

balance-break : (l : Tree) (kv : Key × Value) (r : Tree)
  → WellRed l
  → WellRed r
  → AlmostWellRed (balance l Red kv r)
balance-break = {!!}

```

If both l and r are well-red, and the root is red, then `balance` returns an almost well-red tree. Recall the counter-example above, which shows that it doesn't always create a well-red—it's important that we give `ins` the slack to return an almost well-red tree when the root is red!

To complete the proof we just need to apply this lemma to the IH and the assumption about r :

```

ins-black .(Node l Black (k',v') r) (k,v) rt (WR-Black {l} {k',v'} {r} wl wr) with compare k k'
... | Less = balance-fix-left (ins l (k,v)) (k',v') r (ins-any l (k,v) wl) wr
... | Equal = WR-Black wl wr
... | Greater = balance-fix-right l (k',v') (ins r (k,v)) wl (ins-any r (k,v) wr)

```

Now, by `ins-any`, we know that `ins t (k,v)` is always almost well-red. To finish off `insert`, we need to take an almost well-red tree and make it well-red. This is exactly the purpose of `blackenRoot`: if we make the root black, that fixes any possible red-red violation at the root!

```

blackenRoot-fix : {t : Tree} → AlmostWellRed t → WellRed (blackenRoot t)
blackenRoot-fix AWR-Empty = WR-Empty
blackenRoot-fix (AWR-Node wl wr) = WR-Black wl wr

```

Figure 1.5 collects the bits of proof that we've developed so far, and concludes with `insert-wellred`. One detail that we skipped above: because `ins-black` and `ins-red` and `ins-any` all call each other, they are *mutually inductive*. In Agda, this is notated by putting them in a mutual block.

Correctness of `balance` To prove `balance-fix-left`, we will use our invariant about `view` from above. Before we do that, let's see why we need it. Suppose we try to prove the theorem. We clearly need to do a **with** on `view`, because that's what `balance` does: the goal is

```

WellRed (balance Key compare Value l Black kv r | view l Black kv r)

```

```

balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr with view l Black kv r
... | bv = {!!}

```

So far, so good: the goal is now

```

WellRed (balance Key compare Value l Black kv r | bv)

```

so we can match on `bv` and make progress:

```

balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr with view l Black kv r
balance-fix-left l kv r al wr | Violation a x b y c z d = {!!}
balance-fix-left l kv r al wr | OK l' co' kv' r' = {!!}

```

Now, the goal has reduced appropriately, to `WellRed (Node (Node a Black x b) Red y (Node c Black z d))` in the first case, and to `WellRed (Node l' co' kv' r')`. The outer structure of the `Violation` case is obvious:

```

balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr with view l Black kv r
balance-fix-left l kv r al wr | Violation a x b y c z d = WR-Red (WR-Black {!!} {!!}) (WR-Black {!!} {!!}) RC-Black RC-Black
balance-fix-left l kv r al wr | OK l' co' kv' r' = {!!}

```

```

balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left = {!!}

balance-fix-right : (l : Tree) (kv : Key × Value) (r : Tree) → WellRed l → AlmostWellRed r → WellRed (balance l Black kv r)
balance-fix-right = {!!}

balance-break : (l : Tree) (kv : Key × Value) (r : Tree) → WellRed l → WellRed r → AlmostWellRed (balance l Red kv r)
balance-break = {!!}

decide-root : (t : Tree) → Either (RootColored t Black) (RootColored t Red)
decide-root Empty = Inl RC-Empty
decide-root (Node _ Red _ _) = Inr RC-Red
decide-root (Node _ Black _ _) = Inl RC-Black

mutual

ins-black : (t : Tree) (kv : Key × Value) → RootColored t Black → WellRed t → WellRed (ins t kv)
ins-black .Empty kv rt WR-Empty = WR-Red WR-Empty WR-Empty RC-Empty RC-Empty
ins-black .(Node l Red kv' r) kv () (WR-Red {l} {kv'} {r} y y' y0 y1)
ins-black .(Node l Black (k', v') r) (k, v) rt (WR-Black {l} {k', v'} {r} wl wr) with compare k k'
... | Less = balance-fix-left (ins l (k, v)) (k', v') r (ins-any l (k, v) wl) wr
... | Equal = WR-Black wl wr
... | Greater = balance-fix-right l (k', v') (ins r (k, v)) wl (ins-any r (k, v) wr)

ins-red : (t : Tree) (kv : Key × Value) → RootColored t Red → WellRed t → AlmostWellRed (ins t kv)
ins-red .Empty kv rc WR-Empty = AWR-Node WR-Empty WR-Empty
ins-red .(Node l Black kv' r) kv () (WR-Black {l} {kv'} {r} y y')
ins-red .(Node l Red (k', v') r) (k, v) rc (WR-Red {l} {k', v'} {r} wl wr rl rr) with compare k k'
... | Less = balance-break (ins l (k, v)) (k', v') r (ins-black l (k, v) rl wl) wr
... | Equal = AWR-Node wl wr
... | Greater = balance-break l (k', v') (ins r (k, v)) wl (ins-black r (k, v) rr wr)

ins-any : (t : Tree) (kv : Key × Value) → WellRed t → AlmostWellRed (ins t kv)
ins-any t kv wt with decide-root t
... | Inl rt = forget (ins-black t kv rt wt)
... | Inr rt = ins-red t kv rt wt

blackenRoot-fix : {t : Tree} → AlmostWellRed t → WellRed (blackenRoot t)
blackenRoot-fix AWR-Empty = WR-Empty
blackenRoot-fix (AWR-Node wl wr) = WR-Black wl wr

insert-wellred : {t : Tree} {kv : Key × Value} → WellRed t → WellRed (insert t kv)
insert-wellred w = blackenRoot-fix (ins-any _ _ w)

```

Figure 1.5: Correctness of insert

But now we need

```
?0 : WellRed a
?1 : WellRed b
?2 : WellRed c
?3 : WellRed d
```

to finish the proof. What do we have in the context?

```
wr : WellRed r
al : AlmostWellRed l
```

Morally, this is enough information: when `view l Black kv r` returns `Violation a x b y c z d`, the trees `a b c d` are subtrees of `l` or `r`, and the above info is enough to see that they are well-red. However, in the code, we've lost the connection between `l` and `r` and `a b c d`. That's where `ViewInvariant` comes in.

Something similar happens in the second case: we've lost the connection between `l' co' kv' r'` and `l Black kv r`. Moreover, we have no way of knowing that there was no red-red violation in `l`, which is necessary to conclude that it is well-red.

On to take 2: we use `view-ok` from above:

```
balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr with view-ok l Black kv r
... | vok = {! vok!}
```

Now, we want to split on `vok`, but if we try to do so, we get the following error message:

```
Cannot decide whether there should be a case for the constructor
ViolationInv, since the unification gets stuck on unifying the
inferred indices [l2, co, kv2, r2, Violation a x b y c z d] with
the expected indices [l1, Black, kv1, r1, view l1 Black kv1 r1]
```

What gives? Splitting `vok` would require deciding whether there are some `l1 kv1 r1` such that `view l1 Black kv1 r1` equals `Violation a x b y c z d`: if there are, we need to give a case for `ViolationOK`; if there aren't, we don't. However, there's no unique way to solve this equation: it could be any of the red-red violations. So Agda gets stuck. But this seems wrong, since the whole point of `view-ok` is to tell us what happened with `view`!

The solution is to do a *double with*: we need to look at both `view` and `view-ok` simultaneously! The syntax is to separate multiple `with`s with a `|`—think of this as adding *two* columns to the pattern-match.

```
balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr with view l Black kv r | view-ok l Black kv r
... | vw | vok = {! vok!}
```

Now, we're in the following state:

```
Goal: WellRed (balance Key compare Value l Black kv r | vw)
```

```
vok : ViewInvariant l Black kv r vw
```

Because of the first column, Agda replaces all occurrences of `view l Black kv r`, including the one `balance` is stuck on, with the variable `vw`. Because of the second column, we also have `vok : ViewInvariant l Black kv r vw`—which, unlike above, is a general enough instance of the inductive family that we can split on it. Doing so gives the following cases:

```
balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr with view l Black kv r | view-ok l Black kv r
balance-fix-left l kv r al wr | .(OK l Black kv r) | OKBlackInv r l rr = {!!}
balance-fix-left l kv r al wr | .(Violation a x b y c z d) | ViolationInv {l} {l.Black} {kv} {r} {a} {x} {b} {y} {c} {z} {d} vt = {!!}
```

By splitting vok, we learn what vw is—so now we know what view returned, and balance computes. And we have the additional info that we need: in the OKBlackInv case, we have

Goal: WellRed (Node l Black kv r)

wr : WellRed r
al : AlmostWellRed l
rr : RootOK r
rl : RootOK l

So we can finish the case as follows:

balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr **with** view l Black kv r | view-ok l Black kv r
balance-fix-left l kv r al wr | .(OK l Black kv r) | OKBlackInv rl rr = WR-Black (combine rl al) wr
balance-fix-left l kv r al wr | .(Violation a x b y c z d) | ViolationInv { .l } { .Black } { .kv } { .r } { a } { x } { b } { y } { c } { z } { d } vt = { !! }

In the ViolationInv case, we have

Goal: WellRed (Node (Node a Black x b) Red y (Node c Black z d))

wr : WellRed r
al : AlmostWellRed l
vt : ViolationType a x b y c z d l Black kv r

Above, it was a problem that we knew that l and r were (almost) well-red, but needed to prove something about a b c d. Here, we have a proof of ViolationType connecting them! If we split on vt, we learn what l kv r were in each case:

balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr **with** view l Black kv r | view-ok l Black kv r
balance-fix-left l kv r al wr | .(OK l Black kv r) | OKBlackInv rl rr = WR-Black (combine rl al) wr
balance-fix-left .(Node (Node a Red x b) Red y c) .z .d awabc wd | Violation a x b y c z d | ViolationInv Violation1 = { !! }
balance-fix-left .(Node a Red x (Node b Red y c)) .z .d awabc wd | Violation a x b y c z d | ViolationInv Violation2 = { !! }
balance-fix-left .a .x .(Node (Node b Red y c) Red z d) al wr | Violation a x b y c z d | ViolationInv Violation3 = { !! }
balance-fix-left .a .x .(Node b Red y (Node c Red z d)) al wr | Violation a x b y c z d | ViolationInv Violation4 = { !! }

For example, in the first case, we now have:

Goal: WellRed (Node (Node a Black x b) Red y (Node c Black z d))

wd : WellRed d
awabc : AlmostWellRed (Node (Node a Red x b) Red y c)

So we can break up awabc to get the necessary information!

balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr **with** view l Black kv r | view-ok l Black kv r
balance-fix-left l kv r al wr | .(OK l Black kv r) | OKBlackInv rl rr = WR-Black (combine rl al) wr
balance-fix-left .(Node (Node a Red x b) Red y c) .z .d (AWR-Node (WR-Red wa wb ca cb) wc) wd
| Violation a x b y c z d | ViolationInv Violation1 =
WR-Red (WR-Black wa wb) (WR-Black wc wd) RC-Black RC-Black
balance-fix-left .(Node a Red x (Node b Red y c)) .z .d awabc wd | Violation a x b y c z d | ViolationInv Violation2 = { !! }
balance-fix-left .a .x .(Node (Node b Red y c) Red z d) al wr | Violation a x b y c z d | ViolationInv Violation3 = { !! }
balance-fix-left .a .x .(Node b Red y (Node c Red z d)) al wr | Violation a x b y c z d | ViolationInv Violation4 = { !! }

The next case is analogous, and the next two are contradictory, because in Violation3 and Violation4 the violation is in r, but here we know that r is well-red. The final proof is as follows:

```

balance-fix-left : (l : Tree) (kv : Key × Value) (r : Tree) → AlmostWellRed l → WellRed r → WellRed (balance l Black kv r)
balance-fix-left l kv r al wr with view l Black kv r | view-ok l Black kv r
balance-fix-left .(Node (Node a Red x b) Red y c) .z .d (AWR-Node (WR-Red wa wb ca cb) wc) wd
| Violation a x b y c z d | ViolationInv Violation1 =
  WR-Red (WR-Black wa wb) (WR-Black wc wd) RC-Black RC-Black
balance-fix-left .(Node a Red x (Node b Red y c)) .z .d (AWR-Node wa (WR-Red wb wc rb rc)) wd
| Violation a x b y c z d | ViolationInv Violation2 =
  WR-Red (WR-Black wa wb) (WR-Black wc wd) RC-Black RC-Black
balance-fix-left .a .x .(Node (Node b Red y c) Red z d) al (WR-Red _ _ () _) | Violation a x b y c z d | ViolationInv Violation3
balance-fix-left .a .x .(Node b Red y (Node c Red z d)) al (WR-Red _ _ _ ()) | Violation a x b y c z d | ViolationInv Violation4
... | .(OK l Black kv r) | OKBlackInv rl _ = WR-Black (combine rl al) wr

```

balance-fix-right is analogous. For balance-break, the only possible case is OKRedInv, in which case the result follows from the assumptions that l and r are well-red.

The key new technique you should learn from this is the two-column **with**. This is useful when you want to match on something (here, the ViewInvariant), but one of its indices is something specific (like view l Black kv r), which prevents splitting. In this case, you can use a two-column view to replace the specific term with something general, so that you can split.

1.5.4 Discussion

The main thing to take away from this example is that insert is a simple function, with a more complex correctness proof: in the code, it's not apparent that ins is doing different things depending on whether the root is red or black (sometimes it creates violations; sometimes it fixes them), or that balance actually has three different specs. The reasoning necessary to understand the code is much more involved than the actual steps of the algorithm itself.

1.6 Well-Red, Intrinsically

Like before, we can alternatively do an *intrinsic* verification of the well-red invariant, by baking the appropriate invariants into the data structures. Essentially, we "erase" the trees from the above proof, and use that as the algorithm. For the black-height, this kept the code unchanged. Here, the code will change, because, as we discussed, the above proof has a different structure than the code.

First, let's set up the data structures, by erasing the tree arguments from the above definitions of WellRed and AlmostWellRed:

```

data Color : Set where
  Red : Color
  Black : Color

mutual
  data WellRedTree : Set where
    Empty : WellRedTree
    RedNode : (l : WellRedTree)
      → (kv : Key × Value)
      → (r : WellRedTree)
      → (rl : RootColored l Black)
      → (rr : RootColored r Black)
      → WellRedTree
    BlackNode : (l : WellRedTree)
      → (kv : Key × Value)
      → (r : WellRedTree)
      → WellRedTree

  data RootColored : WellRedTree → Color → Set where
    RC-Empty : RootColored Empty Black

```



```

RC-Red  : {l : WellRedTree} {kv : Key × Value} {r : WellRedTree} {cl : RootColored l Black} {cr : RootColored r Black} →
RC-Black : {l : WellRedTree} {kv : Key × Value} {r : WellRedTree} → RootColored (BlackNode l kv r) Black

```

Something a little bit new is going on here: we're mutually defining a datatype (well red trees) along with another inductive family indexed by that datatype (RootColored). This is called an *inductive-inductive* definition [3].

We will also need AlmostWellRedTrees:

```

data AlmostWellRedTree : Set where
  AEmpty : AlmostWellRedTree
  ANode : (l : WellRedTree)
    → (c : Color)
    → (kv : Key × Value)
    → (r : WellRedTree)
    → AlmostWellRedTree

```

Now, the code for balance is similar to the *proof* above: balance is separated into three functions, one which fixes a potential red-red violation on the left (balance-left), on which fixes a potential red-red violation on the right (balance-right), and one which creates a potential violation (balance-break). Here's the code for balance-left:

```

balance-left : AlmostWellRedTree → Key × Value → WellRedTree → WellRedTree
-- these are the two rotation cases
balance-left (ANode (RedNode a x b ra rb) Red y c) z d = RedNode (BlackNode a x b) y (BlackNode c z d) RC-Black RC-Black
balance-left (ANode a Red x (RedNode b y c rb rc)) z d = RedNode (BlackNode a x b) y (BlackNode c z d) RC-Black RC-Black
-- instances of the catch-all
balance-left AEmpty kv r = BlackNode Empty kv r
balance-left (ANode a Black x b) kv r = BlackNode (BlackNode a x b) kv r
balance-left (ANode Empty Red x Empty) kv r = BlackNode (RedNode Empty x Empty RC-Empty RC-Empty) kv r
balance-left (ANode Empty Red x (BlackNode l kv r)) kv' r' =
  BlackNode (RedNode Empty x (BlackNode l kv r) RC-Empty RC-Black) kv' r'
balance-left (ANode (BlackNode a1 x1 a2) Red x Empty) y c =
  BlackNode (RedNode (BlackNode a1 x1 a2) x Empty RC-Black RC-Empty) y c
balance-left (ANode (BlackNode a1 x1 a2) Red x (BlackNode b1 y1 b2)) y c =
  BlackNode (RedNode (BlackNode a1 x1 a2) x (BlackNode b1 y1 b2) RC-Black RC-Black) y c

```

You should think of this as analogous to balance l Black kv r, where only l may have a red-red violation. The first two clauses do the two necessary rotations. The remaining clauses are instances of the catch-all case in the simply-typed balance but here we need to expand them because the *proof* is different in each case.

balance-right is symmetric and balance-break is trivial:

```

balance-right : WellRedTree → Key × Value → AlmostWellRedTree → WellRedTree
-- these are the two rotation cases
balance-right a x (ANode (RedNode b y c _ _) Red z d) = RedNode (BlackNode a x b) y (BlackNode c z d) RC-Black RC-Black
balance-right a x (ANode b Red y (RedNode c z d _ _)) = RedNode (BlackNode a x b) y (BlackNode c z d) RC-Black RC-Black
-- instances of the catch-all
balance-right a x AEmpty = BlackNode a x Empty
balance-right a x (ANode Empty Red kv Empty) = BlackNode a x (RedNode Empty kv Empty RC-Empty RC-Empty)
balance-right a x (ANode Empty Red kv (BlackNode l kv' r)) =
  BlackNode a x (RedNode Empty kv (BlackNode l kv' r) RC-Empty RC-Black)
balance-right a x (ANode (BlackNode l kv r) Red kv' Empty) =
  BlackNode a x (RedNode (BlackNode l kv r) kv' Empty RC-Black RC-Empty)
balance-right a x (ANode (BlackNode l kv r) Red kv' (BlackNode l' kv0 r')) =
  BlackNode a x (RedNode (BlackNode l kv r) kv' (BlackNode l' kv0 r') RC-Black RC-Black)
balance-right a x (ANode l Black kv r) = BlackNode a x (BlackNode l kv r)
balance-break : WellRedTree → Key × Value → WellRedTree → AlmostWellRedTree
balance-break l kv r = ANode l Red kv r

```

Similarly, the code for ins has the same structure as the above proof:

```

decide-root : (t : WellRedTree) → Either (RootColored t Black) (RootColored t Red)
decide-root Empty = Inl RC-Empty
decide-root (RedNode _ _ _ _) = Inr RC-Red
decide-root (BlackNode _ _ _) = Inl RC-Black

forget : WellRedTree → AlmostWellRedTree
forget Empty = AEmpty
forget (RedNode l kv r _ _) = ANode l Red kv r
forget (BlackNode l kv r) = ANode l Black kv r

mutual
ins-red : (t : WellRedTree) (kv : Key × Value) → RootColored t Red → AlmostWellRedTree
ins-red Empty kv ()
ins-red (RedNode l (k',v') r rl rr) (k,v) rc with compare k k'
... | Less = balance-break (ins-black l (k,v) rl) (k',v') r
... | Greater = balance-break l (k',v') (ins-black r (k,v) rr)
... | Equal = ANode l Red (k,v) r
ins-red (BlackNode l kv r) kv' ()

ins-black : (t : WellRedTree) (kv : Key × Value) → RootColored t Black → WellRedTree
ins-black Empty kv rt = RedNode Empty kv Empty RC-Empty RC-Empty
ins-black (RedNode l kv r rl rr) kv' ()
ins-black (BlackNode l (k',v') r) (k,v) rt with compare k k'
... | Less = balance-left (ins-any l (k,v)) (k',v') r
... | Greater = balance-right l (k,v) (ins-any r (k,v))
... | Equal = BlackNode l (k,v) r

ins-any : (t : WellRedTree) (kv : Key × Value) → AlmostWellRedTree
ins-any t kv with decide-root t
... | Inl rt = forget (ins-black t kv rt)
... | Inr rt = ins-red t kv rt

blacken-root : AlmostWellRedTree → WellRedTree
blacken-root AEmpty = Empty
blacken-root (ANode l _ kv r) = BlackNode l kv r

insert : WellRedTree → Key × Value → WellRedTree
insert t kv = blacken-root (ins-any t kv)

```

1.6.1 Discussion

Is this a better or worse piece of code than the extrinsic version? Evaluate in terms of finding bugs, how easy is it to write and read, separation of properties, and efficiency.

Bibliography

- [1] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.
- [2] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2–15. ACM, 2006.
- [3] F. N. Forsberg and A. Setzer. Inductive-inductive definitions. In A. Dawar and H. Veith, editors, *Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 454–468. Springer Berlin / Heidelberg, 2010.
- [4] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
- [5] C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.
- [6] N. Oury and W. Swierstra. The power of pi. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.