Verified Red-Black Trees: A comparison between Agda and Haskell

Toon Nolten
<toon.nolten@student.kuleuven.be>

***Abstract*—Dependent types are an advanced feature of some type systems that allow properties of programs to be embedded in their types. Originally this was applied in proof assistants, a property is represented as a type and its proof is a function of that type. However this also works the other way around, we can embed proofs in our programs and in doing so prevent certain bugs from passing a static verification, i.e. type checking the program. This correspondence between propositions and types is called the Curry-Howard correspondence and is what gives dependent types their expressiveness. Several languages have such types, most of them are on a spectrum between being a proof assistant and a programming language, Agda leans more towards the latter. In contrast, languages with a Hindley-Milner type system are not dependently typed but in the case of Haskell, extensions to the Glasgow Haskell Compiler have been developed that borrow concepts and fit them into the existing system. In this paper we demonstrate how dependent types can be used to add some verification to the implementation of red-black trees by Chris Okasaki. This is done once for a fully dependently typed language, i.e. Agda, and in comparison, in Haskell with the appropriate GHC extensions. Although these implementations are longer than the original, they prevent us from violating some of the red-black tree invariants which is exactly what they were supposed to accomplish.**

***Index Terms*—Dependent Types, Agda, Haskell, Red-Black Trees, Comparison.**

I. Introduction

DEPENDENT types are a powerful tool to improve software development. Type systems in general are used to prevent certain simple but hard to spot bugs, e.g. a function that returns an integer instead of a floating point number. Dependent types are more expressive than simple types because they can *depend* on values. The canonical introductory example of a dependent type is the type for vectors (lists of fixed length):

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n)
       → Vec A (suc n)
```

For someone with a background in functional programming languages this may be a good example of how dependent types are more expressive but for someone with a background in imperative languages a list with a fixed length is just an array. Hopefully the running example in this paper, i.e. red-black trees [1], will serve to convince those yet unimpressed of the expressiveness of dependent types.

July 15, 2015

A. Red-Black Trees

Red-black trees are (approximately) balanced binary search trees. Relying on color information (essentially an extra bit of data per node) to maintain balance. The approximate balancing is good enough to guarantee $\mathcal{O}(\log n)$ time complexity for search, insertion and deletion. Binary search trees are often used to implement more abstract data structures such as sets and maps. Especially for ordered sets and maps this approach has advantages when compared to a hash table based implementation, as search trees maintain order by definition.

An elegant implementation of red-black trees in a functional programming language is that by Chris Okasaki [2], in Haskell [3], which is what the implementations in the examples are based on:

```
data Color = R | B
data Tree elt =
  E | T Color (Tree elt) elt (Tree elt)

type Set a = Tree a

empty :: Set elt
empty = E

member :: Ord elt => elt -> Set elt -> Bool
member x E = False
member x (T _ a y b) | x <  y = member x a
                     | x == y = True
                     | x >  y = member x b

-- originally without type signature
balance :: Color -> Tree elt -> elt
           -> Tree elt -> Tree elt
balance B (T R (T R a x b) y c) z d =
  T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d =
  T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) =
  T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) =
  T R (T B a x b) y (T B c z d)
balance color a x b = T color a x b

insert :: Ord elt => elt -> Set elt
          -> Set elt
insert x s = makeBlack (ins s)
  where
    ins E = T R E x E
    ins (T color a y b)
      | x <  y = balance color (ins a) y b
      | x == y = T color a y b
      | x >  y = balance color a y (ins b)

    makeBlack (T _ a y b) = T B a y b
```

Okasaki based his implementation on a succinct formulation of the red-black tree invariants:

Invariant 1 No red node has a red parent.
Invariant 2 Every path from the root to an empty node

contains the same number of black nodes.

While this implementation is short and simple enough for most people with some functional programming experience to understand, it is really important for data structures to be correctly implemented. This is usually achieved by writing a suite of tests that exercise the implementation. Even though this is a proven methodology, there is undeniably some maintenance overhead involved: tests need to be written and updated when the interface changes. Another disadvantage of this approach is that it is difficult to ensure that each and every possible case is tested, usually tests are aimed at corner cases that are identified by simply reasoning about the code. Randomized testing tools such as QuickCheck [4] were created for precisely these reasons. In the examples that follow, the expressiveness of dependent types is exploited to provide static verification of the correctness of the implementations.

II. Red-Black Trees in Agda

The code for this section can be found in its entirety in appendix A. No matter how elegant Okasaki's implementation, the `Tree` data type in no way prevents misuse, e.g. `T R (T R E 1 E) 2 E` and `T B (T B E 1 E) 2 E` are both valid values of the type `Tree` while they don't satisfy the invariants on red-black trees. Because of this we have to be careful when implementing a function that produces a `Tree`. Not to mention how we should handle invalid red-black trees? The easiest solution is not to think about it, but then a runtime error could occur because somewhere we defined a function with the implicit assumption it would receive a valid red-black tree and it does not have a matching clause for an invalid tree. The other possible outcome is that it does not cause an error; every function an invalid tree passes through may have done something and all we see is the end result, this could be a debugging nightmare. The expressiveness of dependent types can ensure that only valid red-black trees can be created, in a way the `Tree` type will more accurately represent all possible red-black trees, in Agda [5]:

```
data Color : Set where
  R B : Color
Height = ℕ

data Tree : Color → Height → Set a where
  E : Tree B 0
  R : ∀{h} → Tree B h → A → Tree B h
      → Tree R h
  B : ∀{cl cr h} → Tree cl h → A → Tree cr h
      → Tree B (suc h)
```

Instead of the algebraic data type (ADT) [6] in Haskell we use an inductive family [7] to define the `Tree` type, this time including the `Color` (of the root node) and `Height` (number of black nodes on a path from the root to an empty node) in the type. This type enforces both red-black tree invariants: the constructor for a red node only allows children with a black root, therefore preventing a red node from having a red parent which is invariant 1. The constructors for red and black nodes demand their children be of the same height, thereby enforcing invariant 2. Now the type system guarantees that our functions always produce valid red-black trees. If we define a function that tries to create an invalid `Tree` the code will not type check. As a side effect type checking becomes part of writing code, instead of writing a large part of the program and then type checking at compile time. Agda integrates type checking into the editor which facilitates a more interactive workflow, alternately writing some code and type checking, making use of feedback from the type checker to correct the code.

The biggest advantage of this stronger, more precise type is also somewhat of a disadvantage at the same time. Oftentimes an algorithm will temporarily violate invariants because it is easier to mess up and restore than do everything right immediately, especially when changes can cascade. Okasaki makes use of this in his `ins` function: `ins` may return an invalid `Tree` which is why the calls only occur as part of a call to `makeBlack` or `balance`. Because of how `ins` is implemented it may return a `Tree` with a red root that has one red child, this is also called an *infrared* tree. Transforming an infrared tree into a valid tree can be done by blackening (just changing the color to black) the root node, however this increases the height of the tree by one. During insertion a height increase is undesirable because then the tree has to be rebalanced. Since the tree has to be rebalanced anyway, this is what happens in `ins`. `ins` recurses by deconstructing the node and then deciding which child the new value belongs to. If this child is infrared and the current node is black Okasaki's rebalancing algorithm restores the red-black tree invariants: this is where the elegance of his solution comes from. It is also the only way such an imbalance can occur, because a child can only be infrared, if it had a red root before the insertion which means the node that was just deconstructed is black. So every time we need to rebalance there is a black node with an infrared child; of course this relies on the assumption that the original tree was a valid red-black tree. Since this kind of temporary violation of invariants is not allowed by the dependently typed implementation we need a new data type to represent infrared trees:

```
data IRTree : Height → Set a where
  IRl : ∀{h} → Tree R h → A → Tree B h
    → IRTree h
  IRr : ∀{h} → Tree B h → A → Tree R h
    → IRTree h
```

Because the type for an infrared tree and a red-black tree are now distinct we cannot define `balance` as Okasaki did because the left tree can be infrared and the right red-black or the other way around. However we can easily mimick this behaviour using another data type:

```
data OutOfBalance : Height → Set a where
  _◂_◂_ : ∀{c h} → IRTree h → A → Tree c h
          → OutOfBalance h
  _▸_▸_ : ∀{c h} → Tree c h → A → IRTree h
          → OutOfBalance h
```

Thanks to Agda's flexible mixfix syntax, this Haskell code `balance B infra b c`, turns into this Agda code: `balance (infra ◂ b ◂ c)`. Notice also that we do not require the color argument, we will come back to this. There's still one type missing, some of Okasaki's functions like `ins` can return either a red-black or an infrared tree. Because those are represented by two separate types we need an extra type to act as a disjoint sum:

```
data Treeish : Color → Height → Set a where
  RB : ∀{c h} → Tree c h → Treeish c h
  IR : ∀{h} → IRTree h → Treeish R h
```

Being more precise takes more effort, one data type is replaced by four. The increase in precision goes hand in hand with an increase of verbosity, this will also be apparent in the implementation of the functions. The first function to consider is balance because this is the main source of elegance:

```
balance : ∀{h} → OutOfBalance h → Tree R (suc h)
balance (IRl (R a x b) y c ◂ z ◂ d) =
  R (B a x b) y (B c z d)
balance (IRr a x (R b y c) ◂ z ◂ d) =
  R (B a x b) y (B c z d)
balance (a ▸ x ▸ IRl (R b y c) z d) =
  R (B a x b) y (B c z d)
balance (a ▸ x ▸ IRr b y (R c z d)) =
  R (B a x b) y (B c z d)
```

This is fairly similar to Okasaki's implementation, arguably it is more elegant, but there are subtle differences. The color argument is no longer necessary, it could be replaced by a pattern match on the constructor, but we only balance trees that are out of balance and they only produce red trees. The reason for this as well as the removal of the *catch-all* clause is that the type no longer permits a valid red-black tree. In Okasaki's implementation, when `balance` was called with a valid red-black tree it would be returned unaltered, that way the function can be called indiscriminately. With the more restricted implementation it is necessary to ensure `balance` is only called when necessary. Often this requires more extensive case analysis but this can be a good thing because if there's a bug in Okasaki's implementation, it can be harder to find since every **`Tree`** could go through `balance`, which means we'd have to check that `balance` has the right behaviour even for trees that shouldn't even be rebalanced in the first place.

The local functions Okasaki declared in `insert` have changed a lot more. They are not defined locally because we need multiple clauses for `insert`. The simplest of these is `blacken`, similar to Okasaki's `makeBlack`:

```
blacken : ∀{c h} → (Treeish c h)
          → (if c =ᶜ B then Tree B h
                       else Tree B (suc h))
blacken (RB E) = E
blacken (RB (R l b r)) = (B l b r)
blacken (RB (B l b r)) = (B l b r)
blacken (IR (IRl l b r)) = (B l b r)
blacken (IR (IRr l b r)) = (B l b r)
```

All `blacken` does is take a tree and make the root node black. However, this function has to handle valid red-black trees as well as infrared trees (which corresponds to `Treeish`) and for each of these we have to consider each constructor because we need to access the values used to construct the root node. When a red root is colored black the `Height` of the `Tree` increases by one, that's why we have a return type that is conditional on the color of the original `Tree`.

Because `ins` is no longer defined locally we need an explicit argument for the value to be inserted. `ins` takes a red-black `Tree` and returns a `Tree` if the given `Tree` was black or a possibly infrared `Treeish` if it was not. Either of these would have the same `Height` which can be guaranteed only because the function can return infrared trees. The color of the resulting tree is left undecided by using a dependent pair, this does mean we have to return a pair of a `Color` and a `Tree`. Okasaki's rebalancing algorithm can't balance infrared trees by themselves but it doesn't need to because an infrared tree can be turned into a red-black tree by coloring the root black:

```
ins : ∀{c h} → (a : A) → (t : Tree c h)
      → Σ[ c′ ∈ Color ]
         (if c =ᶜ B then (Tree c′ h)
                    else (Treeish c′ h))
ins a E = R , R E a E
--
ins a (R _ b _) with a ≤ b
ins a (R l _ _) | LT with ins a l
ins _ (R _ b r) | LT | R , t =
  R , IR (IRl t b r)
ins _ (R _ b r) | LT | B , t =
  R , (RB (R t b r))
ins _ (R l b r) | EQ = R , RB (R l b r)
ins a (R _ _ r) | GT with ins a r
ins _ (R l b _) | GT | R , t =
  R , (IR (IRr l b t))
ins _ (R l b _) | GT | B , t =
  R , (RB (R l b t))
--
ins a (B _ b _) with a ≤ b
ins a (B l _ _) | LT with ins a l
ins _ (B {R} _ b r) | LT | c , RB t =
  B , B t b r
ins _ (B {R} _ b r) | LT | .R , IR t =
  R , balance (t ◂ b ◂ r)
ins _ (B {B} _ b r) | LT | c , t =
  B , B t b r
ins _ (B l b r) | EQ = B , B l b r
ins a (B _ _ r) | GT with ins a r
ins _ (B {cr = R} l b _) | GT | c , RB t =
  B , B l b t
ins _ (B {cr = R} l b _) | GT | .R , IR t =
  R , balance (l ▸ b ▸ t)
ins _ (B {cr = B} l b _) | GT | c , t =
  B , B l b t
```

`ins` is a lot longer in this implementation, mostly because we need to be more precise. Whether we return a red-black or an infrared tree and whether we should call `balance` or not, matters. The **`with`** construct enables pattern matching on intermediate results to decide which child a value belongs in and if the intermediate `Tree` needs to be rebalanced or not.

The `insert` function has a more precise type than Okasaki's implementation but it is not as precise as possible, i.e. the resulting `Tree` cannot have just any `Height`, but it is not uniquely defined either. The constructors for the disjoint sum have been aliased to make it more obvious what the resulting type is, either the resulting `Tree` has the same `Height` (h+0) or the `Height` has increased by one (h+1). Returning a value as part of a disjoint sum here is a concession, the problem with being more precise is that it is difficult to say in advance whether a given `Tree` will increase in `Height` upon inserting a given value. A function that decides this can easily be defined and used in the type of `insert` but Agda cannot reason about this function which gives problems because Agda only allows total functions. If such a function is

part of the type Agda cannot infer when certain results of the function are impossible and this combined with the necessity to cover all cases that Agda can't rule out, makes this very difficult. This can probably be remedied by providing Agda with an appropriate proof, but this is a trade-off that has to be made, considering the added effort of a proof against a small increase in precision. In this case simplicity was chosen over precision:

```
insert : ∀{c h} → (a : A) → (t : Tree c h)
         → Tree B h ⊎ Tree B (suc h)
insert {R} a t with ins a t
... | R , t' = h+1 (blacken t')
... | B , t' = h+0 (blacken t')
insert {B} a t with ins a t
... | R , t' = h+1 (blacken (RB t'))
... | B , t' = h+0 (blacken (RB t'))
```

`insert` takes a value and a red-black `Tree` and returns a black `Tree` of the same `Height` or one level higher. The cases for a red `Tree` or a black `Tree` are split up because `ins` respectively returns a `Tree` or a red-black `Treeish`.

III. Red-Black Trees in Haskell

The code for this section can be found in its entirety in appendix B. Haskell as a language is certainly not dependently typed but GHC [8] has several extensions that try to retrofit similar features into the Hindley-Milner type system [9]. One of Haskell's most important features is the type inference of its type system, this has to be preserved by whatever extension may be added. Dependent types in general do not allow for type inference, Agda for example has no type inference. This implies that Haskell will never be a fully dependently typed system. This also means that from time to time we will have to jump through hoops to achieve similar guarantees from the type system. In this section we will see how we can achieve most of the benefits of the implementation in the last section in a not quite dependently typed language. The code in this section relies on the following GHC extensions: GADTs [10], DataKinds [11] and KindSignatures which are analogous to type signatures at the kind level. These extensions are attempts to make the Haskell type system more expressive without giving up on type inference. The **Tree** data type is fairly similar:

```
data Nat = Z | S Nat
  deriving (Show, Eq, Ord)

data Color = R | B
  deriving (Show, Eq)

data Tree :: Color -> Nat -> * -> * where
  ET :: Tree B Z a
  RT :: Tree B  h a -> a -> Tree B  h a
        -> Tree R h a
  BT :: Tree cl h a -> a -> Tree cr h a
        -> Tree B (S h) a
```

Other than syntax and constructor names, the most visible difference is the extra parameter for polymorphism. Agda's type system is not polymorphic, implicit arguments allow a very similar approach. In the previous implementation this problem as well as making sure the elements that are put in trees can be compared, was solved in a different way through Agda's module system, but that would take us too far. In Haskell we can use and **Ord** constraint when we need to be able to compare elements. The **Tree** data type is not an ordinary ADT but a generalised algebraic data type (GADT). A GADT enables the specification of the type signature of each constructor individually, this also permits the definition of constructors that result in a value of a more restricted type. There's also a kind [12] signature on **Tree**, just as a value has a type, a type in Haskell has a kind. Most types in Haskell have kind $\star$ and without extensions it is actually impossible to define new kinds. The kind signature for **Tree** indicates that **Tree** is a type, the resulting kind is $\star$, that takes three parameters, i.e. one of kind **Color**, one of kind **Nat** and one of kind $\star$. The **Color** and **Nat** kinds are not defined explicitly, they are created by data type promotion through the DataKinds extension. The type **Color** is promoted to the kind **Color** and the constructors **R** and **B** are promoted to the types **R** and **B**, any of the promoted objects can be preceded by a single quote, e.g. `'Color` if there is a possibility of ambiguity.

Because the **Tree** type is more specific than in Okasaki's implementation we run into the same problem as in the last section and we need several additional types:

```
data IRTree :: Nat -> * -> * where
  IRl :: Tree R h a -> a -> Tree B h a
         -> IRTree h a
  IRr :: Tree B h a -> a -> Tree R h a
         -> IRTree h a

data OutOfBalance :: Nat -> * -> * where
  (:<:) :: IRTree h a -> a -> Tree c h a
           -> OutOfBalance h a
  (:>:) :: Tree c h a -> a -> IRTree h a
           -> OutOfBalance h a

data Treeish :: Color -> Nat -> * -> * where
  RB :: Tree c h a -> Treeish c h a
  IR :: IRTree h a -> Treeish R h a
```

Except for minor differences, i.e. these are GADTs instead of inductive families, polymorphism and not allowing mixfix notation, these types are the same as the ones in Agda.

The `balance` function is also very similar:

```
balance :: OutOfBalance h a -> Tree R (S h) a
balance ((:<:) (IRl (RT a x b) y c) z d) =
  RT (BT a x b) y (BT c z d)
balance ((:<:) (IRr a x (RT b y c)) z d) =
  RT (BT a x b) y (BT c z d)
balance ((:>:) a x (IRl (RT b y c) z d)) =
  RT (BT a x b) y (BT c z d)
balance ((:>:) a x (IRr b y (RT c z d))) =
  RT (BT a x b) y (BT c z d)
```

`blacken` has a slightly less specific type. Functions that operate on the type level are a lot more restricted in Haskell, precisely because there is a type level, whereas in Agda all types are values. Type families [13] are a GHC extension that provides something like a function at the type level but for simplicity's sake we will just relax the type:

```
blacken :: Treeish c h a
           -> Either (Tree B h a)
                     (Tree B (S h) a)
blacken (RB ET) = Left ET
blacken (RB (RT l b r)) = Right (BT l b r)
```

```
blacken (RB (BT l b r)) = Left  (BT l b r)
blacken (IR (IRl l b r)) = Right (BT l b r)
blacken (IR (IRr l b r)) = Right (BT l b r)
```

The return type is now a disjoint sum, this means the result has to be explicitly tagged as having remained the same height or having increased by one.

The `ins` function also has a slightly relaxed type in that we always return a **`Treeish`**. The disjoint sum with **`Either`** in this case replaces the existential quantification of the dependent pair in the Agda implementation. Of course a disjoint sum can only replace such a dependent pair if the type we want to quantify over has exactly two elements. Because this function needs to decide where an element belongs in the **`Tree`** there is an **`Ord`** constraint on the type variable representing the element type:

```
ins :: Ord a => a -> Tree c h a
       -> Either (Treeish R h a)
                 (Treeish B h a)
ins a ET = Left $ RB (RT ET a ET)
--
ins a (RT l b r)
  | a < b , Left (RB t) <- ins a l =
    Left $ IR (IRl t b r)
  | a < b , Right (RB t) <- ins a l =
    Left $ RB (RT t b r)
  | a == b = Left $ RB (RT l b r)
  | a > b , Left (RB t) <- ins a r =
    Left $ IR (IRr l b t)
  | a > b , Right (RB t) <- ins a r =
    Left $ RB (RT l b t)
--
ins a (BT l b r)
  | a < b , Left (RB t) <- ins a l =
    Right $ RB (BT t b r)
  | a < b , Left (IR t) <- ins a l =
    Left $ RB (balance ((:<:) t b r))
  | a < b , Right (RB t) <- ins a l =
    Right $ RB (BT t b r)
  | a == b = Right $ RB (BT l b r)
  | a > b , Left (RB t) <- ins a r =
    Right $ RB (BT l b t)
  | a > b , Left (IR t) <- ins a r =
    Left $ RB (balance ((:>:) l b t))
  | a > b , Right (RB t) <- ins a r =
    Right $ RB (BT l b t)
```

`ins` makes use of pattern guards [14] to unwrap the **`Tree`** or **`IRTree`** from the tagged **`Treeish`** `ins` now invariably returns. The return type always being a **`Treeish`** would be problematic in Agda. Because Agda is a total language, if a function returns a `Treeish` and you want to pattern match on the result, you *have* to provide patterns that cover each possible value of that type. In the case of `ins` this would mean that you'd have to cover infrared trees as a result of each of the recursive calls even though a call to `ins` on a `Tree` with a black root can never result in an infrared tree. What can we return as a result for a case that can not possibly occur? It doesn't really matter what it is as long as it's a valid value of the return type of the function. However, you can't extract the `Height` from the `Tree` because that information is in the type, not the value. Creating a `Tree` of the right `Height` would have to be done recursively while deconstructing the `Tree`, which would require an entire new function just to cover a case that will never occur. In Agda it clearly makes more sense to restrict the type. Haskell is a partial language so we don't have to worry about cases we know will never occur, of course without the totality check we might miss cases that actually can occur.

The `insert` function is simpler this time because `ins` will always return a **`Treeish`**. The type is the same as for the Agda implementation, glossing over the **`Ord`** constraint and polymorphism. Again a pattern guard unwraps the intermediate result before passing it on to `blacken`:

```
insert :: Ord a => a -> Tree c h a
          -> Either (Tree B h a)
                    (Tree B (S h) a)
insert a t
  | Left t' <- ins a t = blacken t'
  | Right t' <- ins a t = blacken t'
```

IV. Conclusion

Dependent types are expressive enough to allow for the specification of complex properties. These properties are then statically verified by the type system, thereby preventing bugs. Agda because of its design as a fully dependently typed language, allows us to specify arbitrarily complex properties about our code without much friction. Haskell on the other hand does allow us to specify interesting properties but it is clear that the necessary functionality comes from extensions and is not as neatly accommodated by the language.

Appendix A
Agda code

Note that this symbol ↪ is used to indicate a line that was automatically wrapped to fit the narrow column format. Do not confuse this symbol with the arrows used to define function types.

```
{-
  Verified Red-Black Trees
        Toon Nolten

  Based on Chris Okasaki's "Red-Black Trees in
  ↪ a Functional Setting"
  where he uses red-black trees to implement
  ↪ sets.

  Invariants
  ----------

  1. No red node has a red parent
  2. Every Path from the root to an empty node
  ↪ contains the same number of
     black nodes.
     (Empty nodes are considered black)
-}
module RedBlackTree where

open import Data.Bool using (Bool; true;
  ↪ false) renaming (T to So; not to ¬)
_⇒_ : ∀{ℓ₁ ℓ₂} → Set ℓ₁ → Set ℓ₂ → Set _
P ⇒ T = {{p : P}} → T
infixr 3 _⇒_

if_then_else_ : ∀{ℓ}{A : Set ℓ} b → (So b ⇒ A)
  ↪ → (So (¬ b) ⇒ A) → A
if true  then t else f = t
if false then t else f = f
```

```
open import Data.Nat hiding (_<_; _≤_; _≟_;
  ↪ compare)
  renaming (decTotalOrder to ℕ-DTO)
open import Level hiding (suc)
open import Relation.Binary hiding (_⇒_)

open import Data.Sum using (_⊎_) renaming
  ↪ (inj₁ to h+0; inj₂ to h+1)
open import Data.Product using (Σ; Σ-syntax;
  ↪ _,_; proj₁; proj₂)

module RBTree {a ℓ}(order : StrictTotalOrder a
  ↪ ℓ ℓ) where

  open module sto = StrictTotalOrder order
  A = Carrier

  pattern LT = tri< _ _ _
  pattern EQ = tri≈ _ _ _
  pattern GT = tri> _ _ _
  _≤_ = compare


  data Color : Set where
    R B : Color

  _=ᶜ_ : Color → Color → Bool
  R =ᶜ R = true
  B =ᶜ B = true
  _ =ᶜ _ = false

  Height = ℕ

  data Tree : Color → Height → Set a where
    E : Tree B 0
    R : ∀{h} → Tree B h → A → Tree B h →
      ↪ Tree R h
    B : ∀{cl cr h} → Tree cl h → A → Tree cr
      ↪ h → Tree B (suc h)

  data IRTree : Height → Set a where
    IRl : ∀{h} → Tree R h → A → Tree B h →
      ↪ IRTree h
    IRr : ∀{h} → Tree B h → A → Tree R h →
      ↪ IRTree h

  data OutOfBalance : Height → Set a where
    _◂_◂_ : ∀{c h} → IRTree h → A → Tree c h
      ↪ → OutOfBalance h
    _▸_▸_ : ∀{c h} → Tree c h → A → IRTree h
      ↪ → OutOfBalance h

  data Treeish : Color → Height → Set a where
    RB : ∀{c h} → Tree c h → Treeish c h
    IR : ∀{h} → IRTree h → Treeish R h

  -- Insertion

  balance : ∀{h} → OutOfBalance h → Tree R
    ↪ (suc h)
  balance (IRl (R a x b) y c ◂ z ◂ d) = R (B a
    ↪ x b) y (B c z d)
  balance (IRr a x (R b y c) ◂ z ◂ d) = R (B a
    ↪ x b) y (B c z d)
  balance (a ▸ x ▸ IRl (R b y c) z d) = R (B a
    ↪ x b) y (B c z d)
  balance (a ▸ x ▸ IRr b y (R c z d)) = R (B a
    ↪ x b) y (B c z d)

  blacken : ∀{c h} → (Treeish c h)
          → (if c =ᶜ B then Tree B h else
            ↪ Tree B (suc h))
  blacken (RB E) = E
  blacken (RB (R l b r)) = (B l b r)
  blacken (RB (B l b r)) = (B l b r)
  blacken (IR (IRl l b r)) = (B l b r)
  blacken (IR (IRr l b r)) = (B l b r)

  ins : ∀{c h} → (a : A) → (t : Tree c h)
        → Σ[ c′ ∈ Color ] (if c =ᶜ B then
          ↪ (Tree c′ h) else (Treeish c′ h))
  ins a E = R , R E a E
  --
  ins a (R _ b _) with a ≤ b
  ins a (R l _ _) | LT with ins a l
  ins _ (R _ b r) | LT | R , t = R , IR (IRl t
    ↪ b r)
  ins _ (R _ b r) | LT | B , t = R , (RB (R t
    ↪ b r))
  ins _ (R l b r) | EQ = R , RB (R l b r)
  ins a (R _ _ r) | GT with ins a r
  ins _ (R l b _) | GT | R , t = R , (IR (IRr
    ↪ l b t))
  ins _ (R l b _) | GT | B , t = R , (RB (R l
    ↪ b t))
  --
  ins a (B _ b _) with a ≤ b
  ins a (B l _ _) | LT with ins a l
  ins _ (B {R} _ b r) | LT | c , RB t = B , B
    ↪ t b r
  ins _ (B {R} _ b r) | LT | .R , IR t = R ,
    ↪ balance (t ◂ b ◂ r)
  ins _ (B {B} _ b r) | LT | c , t = B , B t b
    ↪ r
  ins _ (B l b r) | EQ = B , B l b r
  ins a (B _ _ r) | GT with ins a r
  ins _ (B {cr = R} l b _) | GT | c , RB t = B
    ↪ , B l b t
  ins _ (B {cr = R} l b _) | GT | .R , IR t =
    ↪ R , balance (l ▸ b ▸ t)
  ins _ (B {cr = B} l b _) | GT | c , t = B ,
    ↪ B l b t

  insert : ∀{c h} → (a : A) → (t : Tree c h)
         → Tree B h ⊎ Tree B (suc h)
  insert {R} a t with ins a t
  ... | R , t′ = h+1 (blacken t′)
  ... | B , t′ = h+0 (blacken t′)
  insert {B} a t with ins a t
  ... | R , t′ = h+1 (blacken (RB t′))
  ... | B , t′ = h+0 (blacken (RB t′))


  -- Simple Set Operations
  set : ∀{c h} → Set _
  set {c}{h} = Tree c h

  empty : set
  empty = E

  member : ∀{c h} → (a : A) → set {c}{h} →
    ↪ Bool
  member a E = false
  member a (R l b r) with a ≤ b
  ... | LT = member a l
  ... | EQ = true
  ... | GT = member a r
```

```
  member a (B l b r) with a ≤ b
  ... | LT = member a l
  ... | EQ = true
  ... | GT = member a r



-- Usage example

open import
  ↪ Relation.Binary.Properties.DecTotalOrder
  ↪ ℕ-DTO
ℕ-STO : StrictTotalOrder _ _ _
ℕ-STO = strictTotalOrder

open module rbtree = RBTree ℕ-STO

t0 : Tree B 2
t0 = B (R (B E 1 E) 2 (B (R E 3 E) 5 (R E 7
  ↪ E)))
       8
       (B E 9 (R E 10 E))

t1 : Tree B 3
t1 = B (B (B E 1 E) 2 (B E 3 E))
       4
       (B (B E 5 (R E 7 E)) 8 (B E 9 (R E 10
         ↪ E)))

t2 : Tree B 3
t2 = B (B (B E 1 E) 2 (B E 3 E))
       4
       (B (R (B E 5 E) 6 (B E 7 E)) 8 (B E 9
         ↪ (R E 10 E)))

open import
  ↪ Relation.Binary.PropositionalEquality
t1≡t0+4 : h+1 t1 ≡ insert 4 t0
t1≡t0+4 = refl

t2≡t1+6 : h+0 t2 ≡ insert 6 t1
t2≡t1+6 = refl
```

Appendix B
Haskell code

```
{-
  Verified Red-Black Trees
        Toon Nolten

  Based on Chris Okasaki's "Red-Black Trees in
  ↪ a Functional Setting"
  where he uses red-black trees to implement
  ↪ sets.

  Invariants
  ----------

  1. No red node has a red parent
  2. Every Path from the root to an empty node
  ↪ contains the same number of
     black nodes.
    (Empty nodes are considered black)
-}
{-# LANGUAGE GADTs, DataKinds, KindSignatures
  ↪ #-}
module RedBlackTree where

data Nat = Z | S Nat deriving (Show, Eq, Ord)

data Color = R | B deriving (Show, Eq)

data Tree :: Color -> Nat -> * -> * where
  ET :: Tree B Z a
  RT :: Tree B  h a -> a -> Tree B  h a ->
    ↪ Tree R h a
  BT :: Tree cl h a -> a -> Tree cr h a ->
    ↪ Tree B (S h) a

instance Eq a => Eq (Tree c h a) where
  ET == ET = True
  RT l a r == RT m b s = a == b && l == m && r
    ↪ == s
  -- Black trees need further pattern matching
    ↪ because of cl and cr
  BT ET a ET == BT ET b ET =
    a == b
  BT ET a r@(RT {}) == BT ET b s@(RT {}) =
    a == b && r == s
  --
  BT l@(RT {}) a ET == BT m@(RT {}) b ET =
    a == b && l == m
  BT l@(RT {}) a r@(RT {}) == BT m@(RT {}) b
    ↪ s@(RT {}) =
    a == b && l == m && r == s
  BT l@(RT {}) a r@(BT {}) == BT m@(RT {}) b
    ↪ s@(BT {}) =
    a == b && l == m && r == s
  --
  BT l@(BT {}) a r@(RT {}) == BT m@(BT {}) b
    ↪ s@(RT {}) =
    a == b && l == m && r == s
  BT l@(BT {}) a r@(BT {}) == BT m@(BT {}) b
    ↪ s@(BT {}) =
    a == b && l == m && r == s
  _ == _ = False


data IRTree :: Nat -> * -> * where
  IRl :: Tree R h a -> a -> Tree B h a ->
    ↪ IRTree h a
  IRr :: Tree B h a -> a -> Tree R h a ->
    ↪ IRTree h a

data OutOfBalance :: Nat -> * -> * where
  (:<:) :: IRTree h a -> a -> Tree c h a ->
    ↪ OutOfBalance h a
  (:>:) :: Tree c h a -> a -> IRTree h a ->
    ↪ OutOfBalance h a

data Treeish :: Color -> Nat -> * -> * where
  RB :: Tree c h a -> Treeish c h a
  IR :: IRTree h a -> Treeish R h a

--Insertion

balance :: OutOfBalance h a -> Tree R (S h) a
balance ((:<:) (IRl (RT a x b) y c) z d) = RT
  ↪ (BT a x b) y (BT c z d)
balance ((:<:) (IRr a x (RT b y c)) z d) = RT
  ↪ (BT a x b) y (BT c z d)
balance ((:>:) a x (IRl (RT b y c) z d)) = RT
  ↪ (BT a x b) y (BT c z d)
balance ((:>:) a x (IRr b y (RT c z d))) = RT
  ↪ (BT a x b) y (BT c z d)
```

```
blacken :: Treeish c h a -> Either (Tree B h
  ↪  a) (Tree B (S h) a)
blacken (RB ET) = Left ET
blacken (RB (RT l b r)) = Right (BT l b r)
blacken (RB (BT l b r)) = Left (BT l b r)
blacken (IR (IRl l b r)) = Right (BT l b r)
blacken (IR (IRr l b r)) = Right (BT l b r)

-- Surprisingly difficult to find the right
  ↪  formulation
-- (ins in a pattern guard)
ins :: Ord a => a -> Tree c h a -> Either
  ↪  (Treeish R h a) (Treeish B h a)
ins a ET = Left $ RB (RT ET a ET)
--
ins a (RT l b r)
  | a < b , Left (RB t) <- ins a l = Left $ IR
    ↪  (IRl t b r)
  | a < b , Right (RB t) <- ins a l = Left $
    ↪  RB (RT t b r)
  | a == b = Left $ RB (RT l b r)
  | a > b , Left (RB t) <- ins a r = Left $ IR
    ↪  (IRr l b t)
  | a > b , Right (RB t) <- ins a r = Left $
    ↪  RB (RT l b t)
--
ins a (BT l b r)
  | a < b , Left (RB t) <- ins a l = Right $
    ↪  RB (BT t b r)
  | a < b , Left (IR t) <- ins a l = Left $ RB
    ↪  (balance ((:<:) t b r))
  | a < b , Right (RB t) <- ins a l = Right $
    ↪  RB (BT t b r)
  | a == b = Right $ RB (BT l b r)
  | a > b , Left (RB t) <- ins a r = Right $
    ↪  RB (BT l b t)
  | a > b , Left (IR t) <- ins a r = Left $ RB
    ↪  (balance ((:>:) l b t))
  | a > b , Right (RB t) <- ins a r = Right $
    ↪  RB (BT l b t)

insert :: Ord a => a -> Tree c h a -> Either
  ↪  (Tree B h a) (Tree B (S h) a)
insert a t
  | Left t' <- ins a t = blacken t'
  | Right t' <- ins a t = blacken t'


-- Simple Set operations
-- Partial Type Signatures might allow
  ↪  'hiding' the color and height
type Set c h a = Tree c h a

empty :: Set B Z a
empty = ET

member :: Ord a => a -> Set c h a -> Bool
member _ ET = False
member a (RT l b r)
  | a < b = member a l
  | a == b = True
  | a > b = member a r
member a (BT l b r)
  | a < b = member a l
  | a == b = True
  | a > b = member a r
```

```
-- Usage example
t0 :: Tree B (S (S Z)) Integer
t0 = BT (RT (BT ET 1 ET) 2 (BT (RT ET 3 ET) 5
  ↪  (RT ET 7 ET)))
        8
        (BT ET 9 (RT ET 10 ET))

t1 :: Tree B (S (S (S Z))) Integer
t1 = BT (BT (BT ET 1 ET) 2 (BT ET 3 ET))
        4
        (BT (BT ET 5 (RT ET 7 ET)) 8 (BT ET 9
          ↪  (RT ET 10 ET)))

t2 :: Tree B (S (S (S Z))) Integer
t2 = BT (BT (BT ET 1 ET) 2 (BT ET 3 ET))
        4
        (BT (RT (BT ET 5 ET) 6 (BT ET 7 ET)) 8
          ↪  (BT ET 9 (RT ET 10 ET)))

-- Would a proof with refl and equality
  ↪  require the entire tree at type
-- level?
t1_is_t0_plus_4 :: Bool
t1_is_t0_plus_4 = t1 == t0_plus_4
  where Right t0_plus_4 = insert 4 t0

t2_is_t1_plus_6 :: Bool
t2_is_t1_plus_6 = t2 == t1_plus_6
  where Left t1_plus_6 = insert 6 t1
```


Acknowledgment

I would like to thank my advisors, Jesper Cockx and Dominique Devriese, for their patient support when writing the code discussed in this paper and their apt suggestions for improvement of the article itself. Furthermore I would like to thank professor Frank Piessens as my promotor without whom I would not have gotten involved in type theory and static program verification. Many thanks also to the wonderful people of the #agda and #haskell irc channels on the freenode network who've helped me better understand both languages.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001, ch. 13: Red-Black Trees, pp. 273–301, ISBN 0-262-03293-7.

[2] C. Okasaki, "Functional Pearls: Red-Black Trees in a Functional Setting," *Journal of Functional Programming*, vol. 9 (04), pp. 471–477, Jan. 1999.

[3] (2015, Jul.) Haskell Language. [Online]. Available: https://www.haskell.org/

[4] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, Sep. 2000.

[5] U. Norell, "Dependently Typed Programming in Agda," in *Proceedings of the 6th International Conference on Advanced Functional Programming*, ser. AFP'08. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 230–266, ISBN-10 3-642-04651-7, ISBN-13 978-3-642-04651-3.

[6] (2015, Jul.) Algebraic data type. [Online]. Available: https://wiki.haskell.org/Algebraic_data_type

[7] P. Dybjer, "Inductive Families," *Formal Aspects of Computing*, vol. 6, pp. 440–465, 1997.

[8] (2015, Jul.) The Glorious Glasgow Haskell Compiler. [Online]. Available: https://www.haskell.org/ghc/

[9] R. Hindley, "The Principal Type-Scheme of an Object in Combinatory Logic," *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, Dec. 1969.

[10] H. Xi, C. Chen, and G. Chen. (2003, Jan.) Guarded Recursive Datatype Constructors. New York, NY, USA.

[11] (2015, Jul.) Datatype Promotion. [Online]. Available: https://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/promotion.html
[12] B. Pierce, *Types and Programming Languages*. MIT Press, 2002, ch. 29: Type Operators and Kinding, ISBN 0-262-16209-1.
[13] (2015, Jul.) Type families. [Online]. Available: https://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/type-families.html
[14] (2015, Jul.) Pattern guard. [Online]. Available: https://wiki.haskell.org/Pattern_guard