Depending on Types

Stephanie Weirich
University of Pennsylvania

TDD

Type-Driven Development

with Dependent Types

The Agda Experience

On 2012-01-11 03:36, Jonathan Leivent wrote on the Agda mailing list:

> Attached is an Agda implementation of Red Black trees [..]
> The dependent types show that the trees have the usual
> red-black level and color invariants, are sorted, and
> contain the right multiset of elements following each function. [..]
>
> However, one interesting thing is that I didn't previously know or
> refer to any existing red black tree implementation of delete - I
> just allowed the combination of the Agda type checker and
> the exacting dependent type signatures to do their thing [..]
> **making me feel more like a facilitator than a programmer**.

Is Haskell a dependently-typed language?

YES*

Dependently-typed Haskell

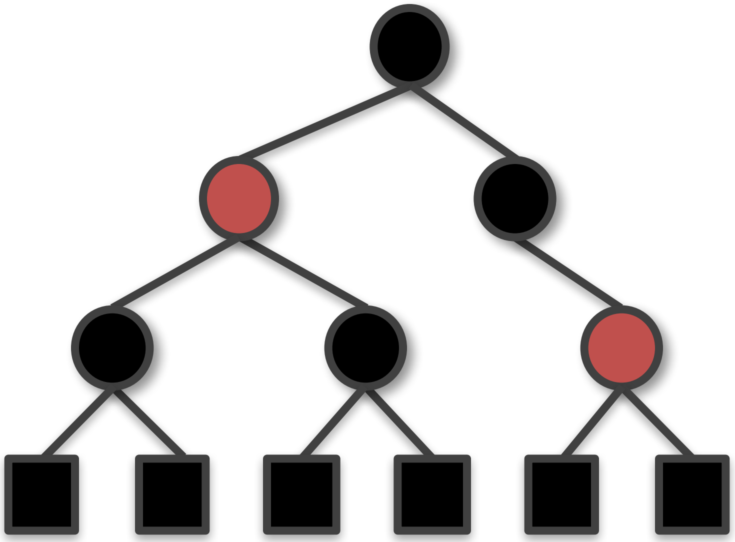
- Show how type system extensions work together to make GHC a dependently-typed language*
- *The Past*: Put those extensions in context, and talk about how they compare to dependent type theory
- *The Future*: Give my vision of where GHC should go and how we should get there

*we cannot port *every* Agda/Coq/Idris program to GHC, but what we can do is impressive

Example: Red-black Trees

Running example of a data structure with application-specific invariants

- *Root is black*
- *Leaves are black*
- *Red nodes have black children*
- *From each node, every path to a leaf has the same number of black nodes*



All code available at

http://www.github.com/sweirich/dth

Insertion [Okasaki, 1993]

```
data Color = R | B
data Tree  = E | T Color Tree A Tree


insert :: Tree -> A -> Tree
insert s x =          ins s
   where ins E = T R E x E
         ins s@(T color a y b)
             | x < y     =          T color (ins a) y b
             | x > y     =          T color a y (ins b)
             | otherwise = s
```

Fix the element type to be A for this talk

Temporarily suspend invariant: Result of ins may create a red node with a red child or red root.

Insertion [Okasaki, 1993]

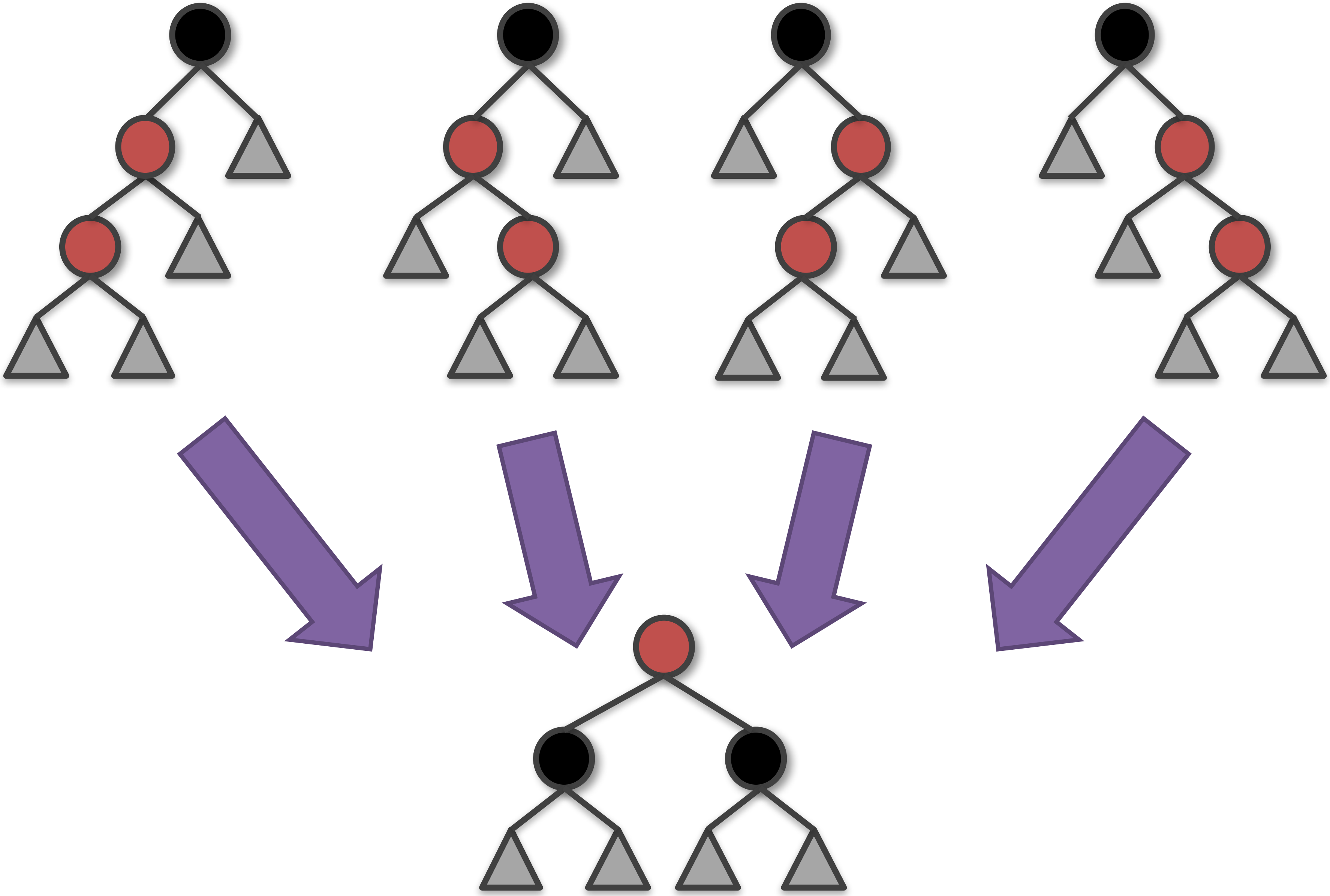
```
data Color = R | B
data Tree  = E | T Color Tree A Tree


insert :: Tree -> A -> Tree
insert s x = blacken (ins s)
   where ins E = T R E x E
         ins s@(T color a y b)
             | x < y     = balance (T color (ins a) y b)
             | x > y     = balance (T color a y (ins b))
             | otherwise = s
         blacken (T _ a x b) = T B a x b
```

Fix the element type to be A for this talk

Temporarily suspend invariant: Result of ins may create a red node with a red child or red root.

Two fixes:
- blacken if root is red at the end
- rebalance two internal reds

balance



How do we know insert preserves Red-black tree invariants?

Do it with types

```
insert :: RBT -> A -> RBT
```

Red-black Trees in Agda [Licata]

```
data ℕ : Set where
  Zero : ℕ
  Suc  : ℕ → ℕ

data Color : Set where
  R : Color
  B : Color
```

Star
like

Arguments of indexed datatypes vary by data constructor.

Indexed datatype

Data constructors have dependent types. The types of later arguments depend on the values of earlier arguments.

```
data Tree : Color → ℕ → Set where
  E  : Tree B Zero
  TR : {n : ℕ} → Tree B n → A → Tree B n → Tree R n
  TB : {n : ℕ} {c₁ c₂ : Color} →
       Tree c₁ n → A → Tree c₂ n → Tree B (Suc n)
```

Agda doesn't distinguish between types and terms. Curly braces indicate inferred arguments.

Red-black Trees in GHC

```
data Tree : Color → ℕ → Set where
  E  : Tree B Zero
  TR : {n : ℕ} → Tree B n → A → Tree B n → Tree R n
  TB : {n : ℕ} {c₁ c₂ : Color} →
           Tree c₁ n → A → Tree c₂ n → Tree B (Suc n)
```

Agda

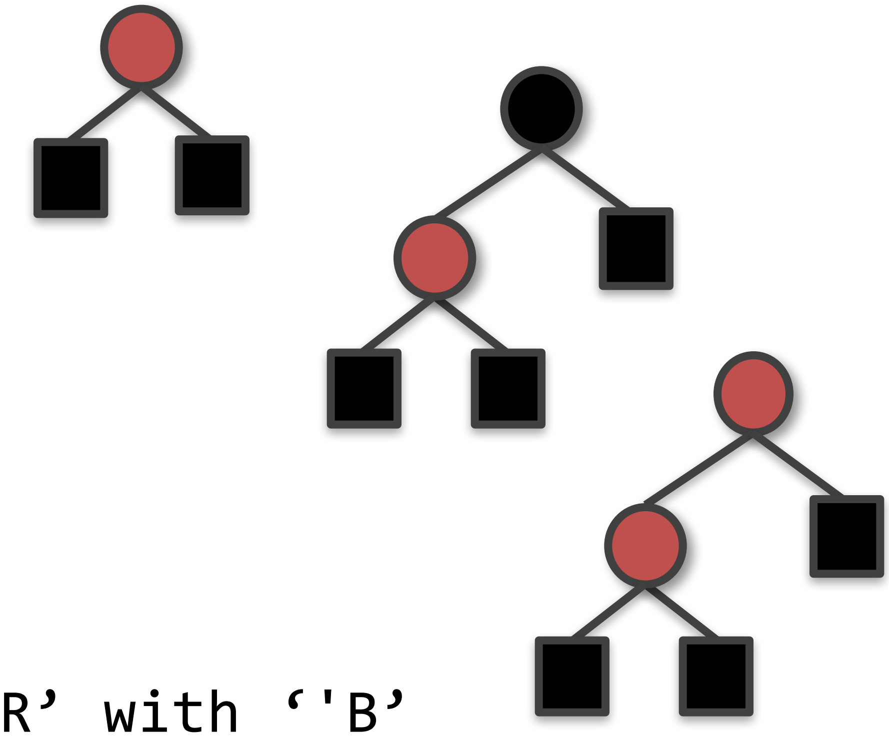
```
data Tree :: Color -> Nat -> * where
   E  :: Tree B Zero
   TR :: Tree B n -> A -> Tree B n -> Tree R n
   TB :: Tree c1 n -> A -> Tree c2 n -> Tree B (Suc n)
```

Haskell

GADTs - datatype arguments may vary by constructor

Datatype promotion – data constructors may be used in types (which are naturally dependent)

Static enforcement



```
ghci> let t1 = TR E a1 E
ghci> :type t1
t1 :: Tree 'R 'Zero
ghci> let t2 = TB t1 a2 E
ghci> :type t2
t2 :: Tree 'B ('Suc 'Zero)
ghci> let t3 = TR t1 a2 E
<interactive>:38:13:
    Couldn't match type ‘'R’ with ‘'B’
    Expected type: Tree 'B 'Zero
      Actual type: Tree 'R 'Zero
    In the first argument of ‘TR’, namely ‘t1’
    In the expression: TR t1 A2 E
```

Static enforcement

RBT: Top-level type for red-black trees

Hides the black height and forces the root to be black

```
data RBT : Set where
  Root  : {n : ℕ} → Tree B n → RBT

insert : RBT → A → RBT
insert (Root t) x = …
```

Agda

Haskell

```
data RBT :: * where
  Root :: Tree B n -> RBT

insert :: RBT -> A -> RBT
insert (Root t) x = …
```

How are Agda and Haskell different?

Haskell distinguishes types from terms

Agda does not

Types are special in Haskell:

1. Type arguments are always inferred (HM type inference)
2. Only types can be used as indices to GADTs
3. Types are always erased before run-time

GADTs: *Type* indices only

- Both Agda and GHC support indexed datatypes, but GHC syntactically requires indices to be *types*
- Datatype promotion automatically creates new data*kinds* from data*types*

```
data Color :: * where  -- Color is both a type and a kind
   R :: Color          -- R and B can appear in both
   B :: Color          -- expressions and types

data Tree :: Color -> Nat -> * where
   E  :: Tree B Zero
   TR :: Tree B n -> A -> Tree B n -> Tree R n
   TB :: Tree c1 n -> A -> Tree c2 n -> Tree B (Suc n)
```

Types are erased

RBT: Top-level type for red-black trees

Hides the black height and forces the root to be black

```
data RBT : Set where
  Root   : {n : ℕ} → Tree B n → RBT


bh : RBT -> ℕ
bh (Root {n} t) = n
```

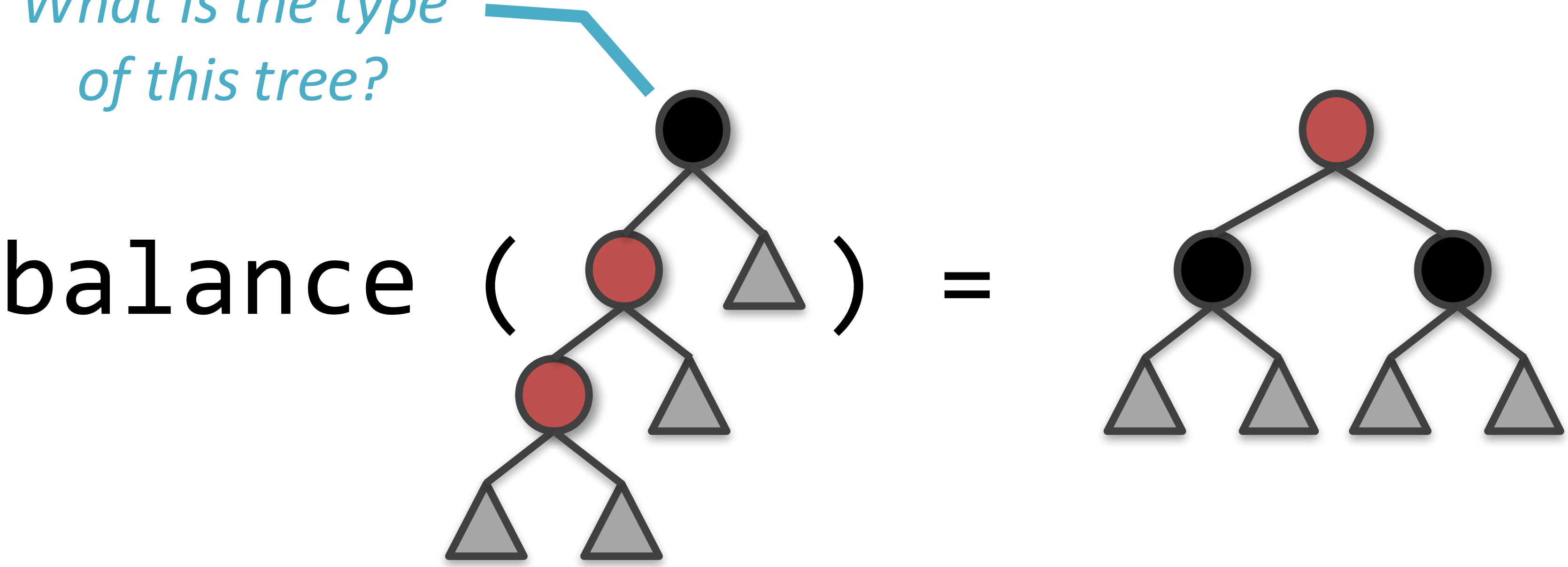
Agda

```
data RBT :: * where
  Root :: Tree B n -> RBT


bh :: RBT -> Nat
bh (Root t) = ???            -- No runtime access to black height
```

Haskell

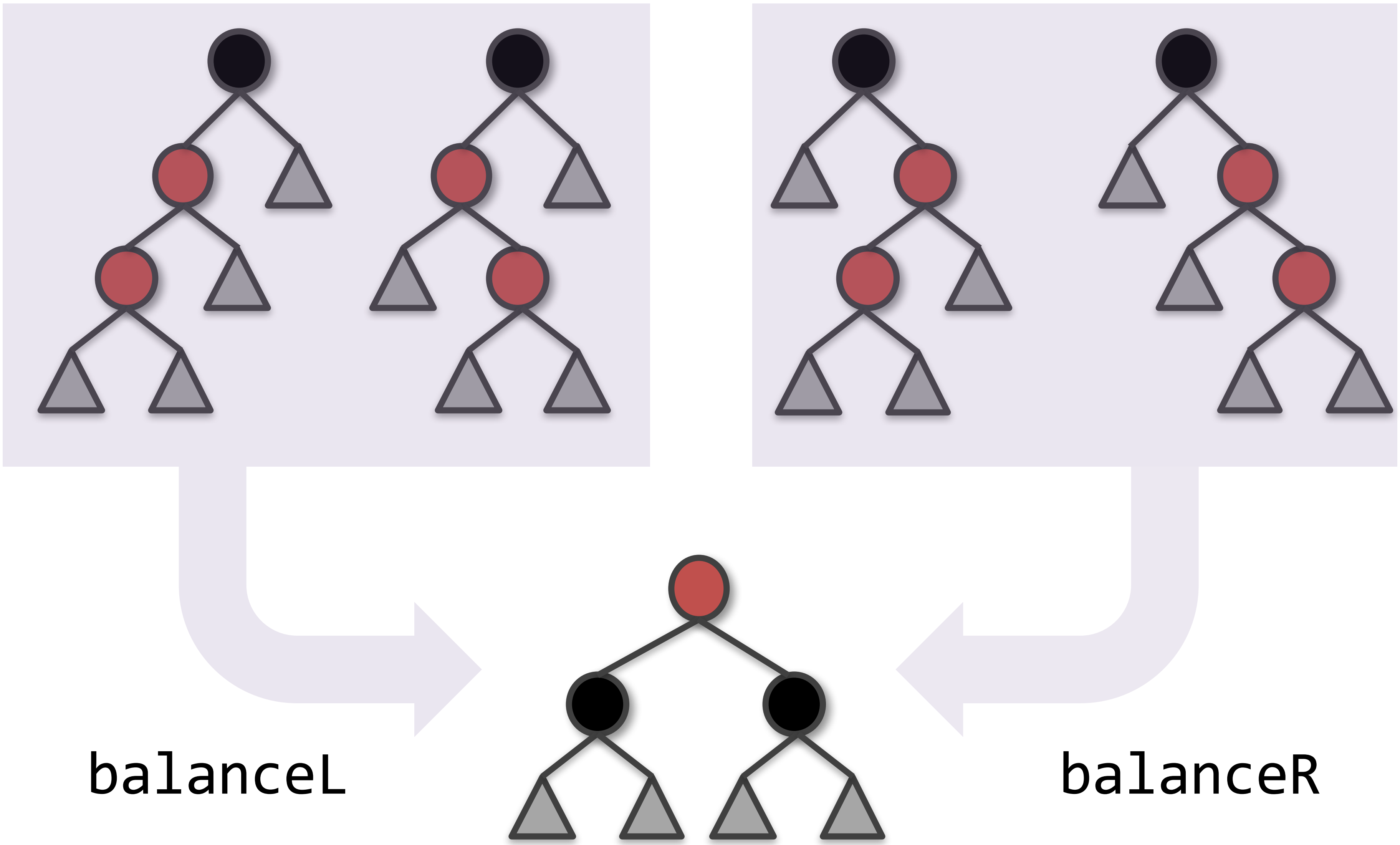
Insertion

How do we temporarily suspend the invariants during insertion?



What is the type of this tree?

Split balance into two cases

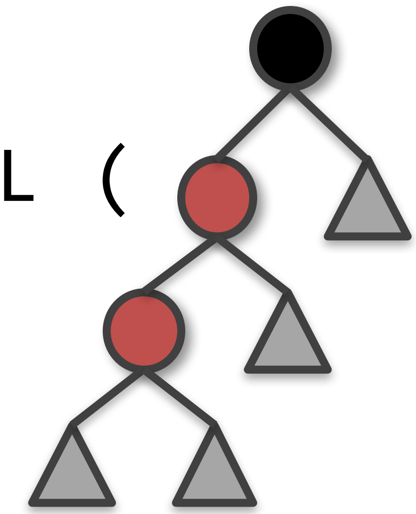


`balanceL`

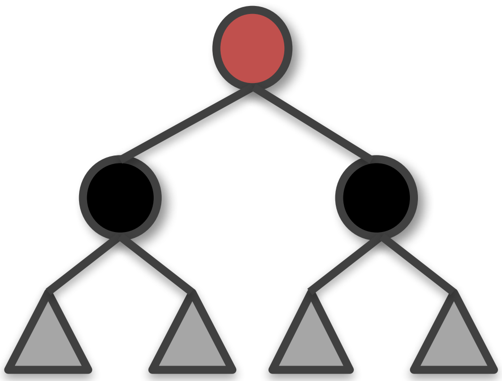
`balanceR`

Decompose argument

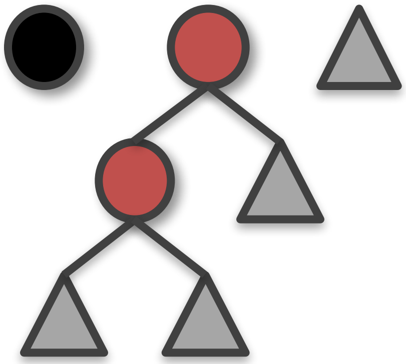
`balanceL (`



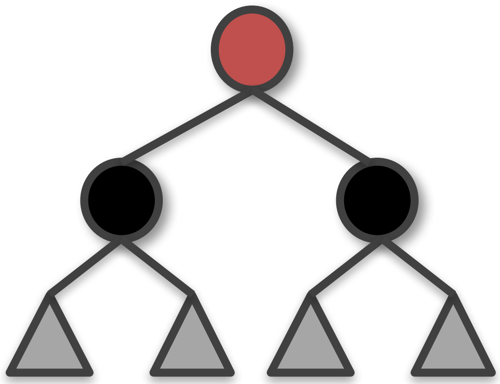
`) =`



`balanceL(`

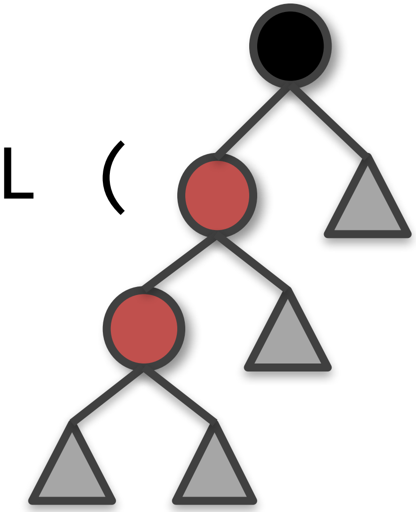


`) =`

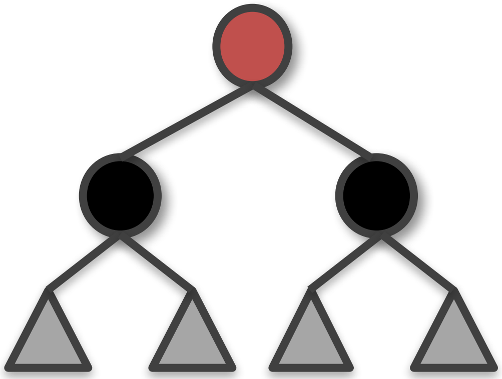


Specialize Color

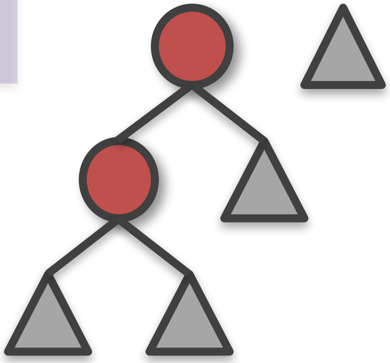
`balanceL (`



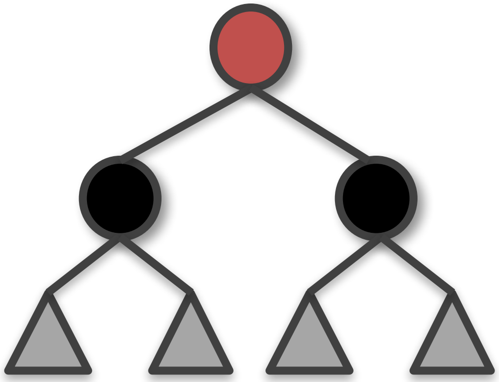
`) =`



`balanceLB(`



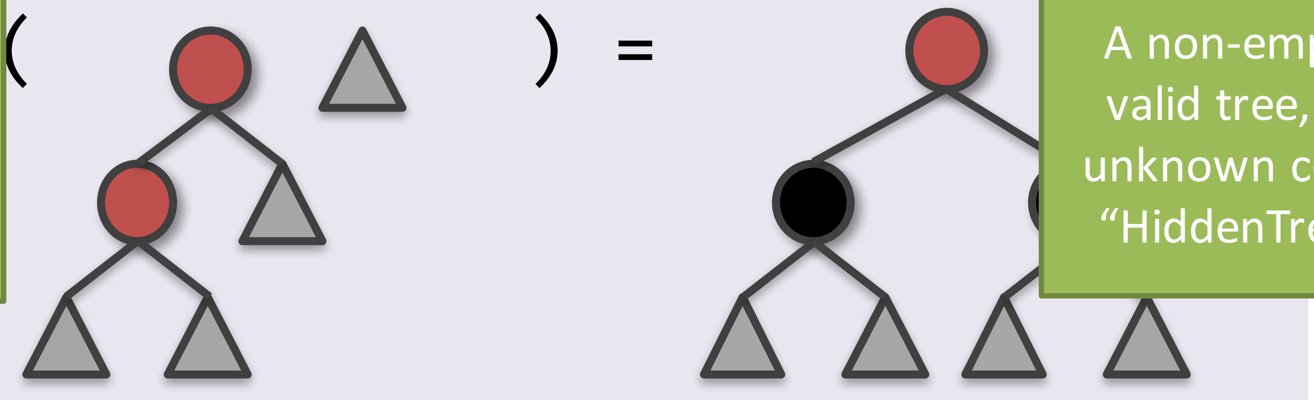
`) =`

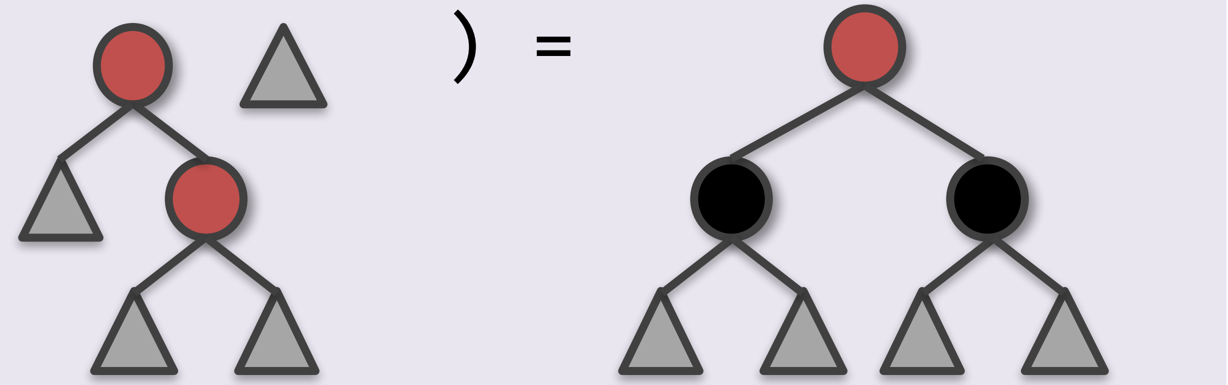


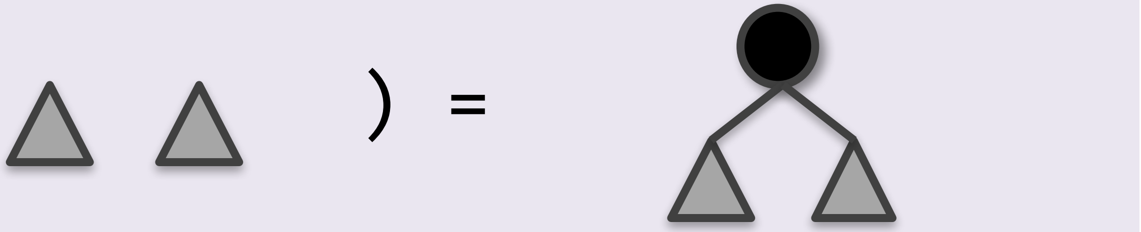
`balanceLB : ??? → A → Tree c n → ???`

A non-empty tree that may break the color invariant at the root "AlmostTree"

A non-empty valid tree, of unknown color "HiddenTree"


`( ) =`


`balanceLB( ) =`


`balanceLB( ) =`

Programming with types (Agda)

- A non-empty valid tree, of unknown color

```
  data HiddenTree  : ℕ → Set  where
     HR : {m : ℕ} → Tree R m         → HiddenTree  m
     HB : {m : ℕ} → Tree B (Suc  m) → HiddenTree  (Suc  m)
```

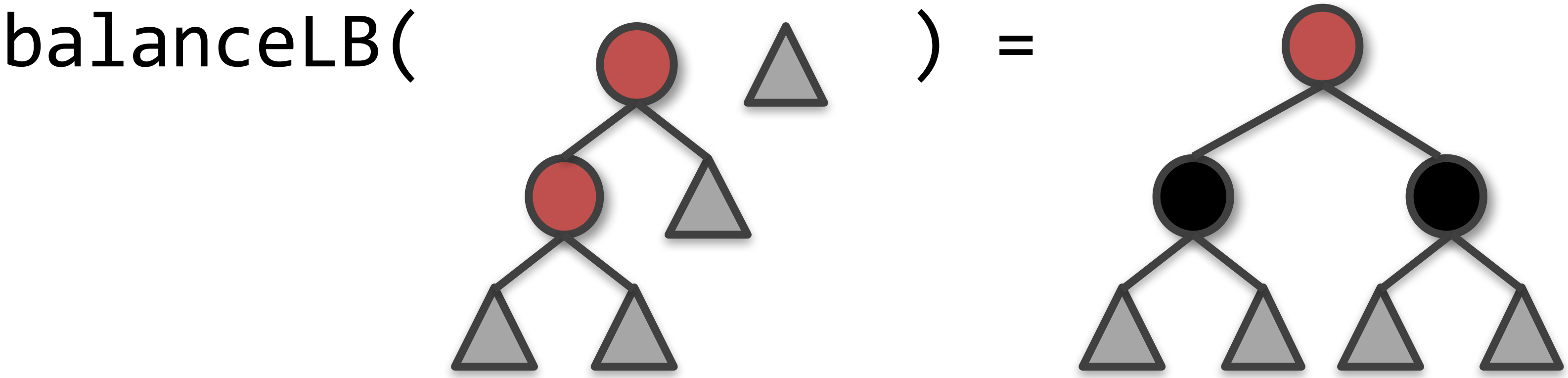
- A non-empty tree that may break the invariant at the root

Use a function to calculate the black height from the color

```
incr  : Color → ℕ → ℕ
incr  B = Suc
incr  R = id


data  AlmostTree  : ℕ → Set  where
   AT  : {n  : ℕ}{c₁  c₂  : Color} → (c  : Color) →
          Tree  c₁  n → A → Tree  c₂  n → AlmostTree  (incr  c  n)
```

Agda

```
balanceLB : {n : ℕ}{c : Color} →
      AlmostTree n → A → Tree c n → HiddenTree (Suc n)
balanceLB (AT R (TR a x b) y c) z d =
  HR (TR (TB a x b) y (TB c z d))
balanceLB (AT R a x (TR b y c)) z d =
  HR (TR (TB a x b) y (TB c z d))
balanceLB (AT B a x b) y r = HB (TB (TB a x b) y r)
balanceLB (AT R E x E) y r = HB (TB (TR E x E) y r)
balanceLB (AT R (TB a w b) x (TB c y d)) z e =
  HB (TB (TR (TB a w b) x (TB c y d)) z e)
```



GHC version of AlmostTree

```
type family Incr (c :: Color) (n :: Nat) :: Nat where
   Incr R n = n
   Incr B n = Suc n

data Sing :: Color -> * where
   SR :: Sing R
   SB :: Sing B

data AlmostTree :: Nat -> * where
   AT :: Sing c -> Tree c1 n -> A -> Tree c2 n ->
        AlmostTree (Incr c n)
```

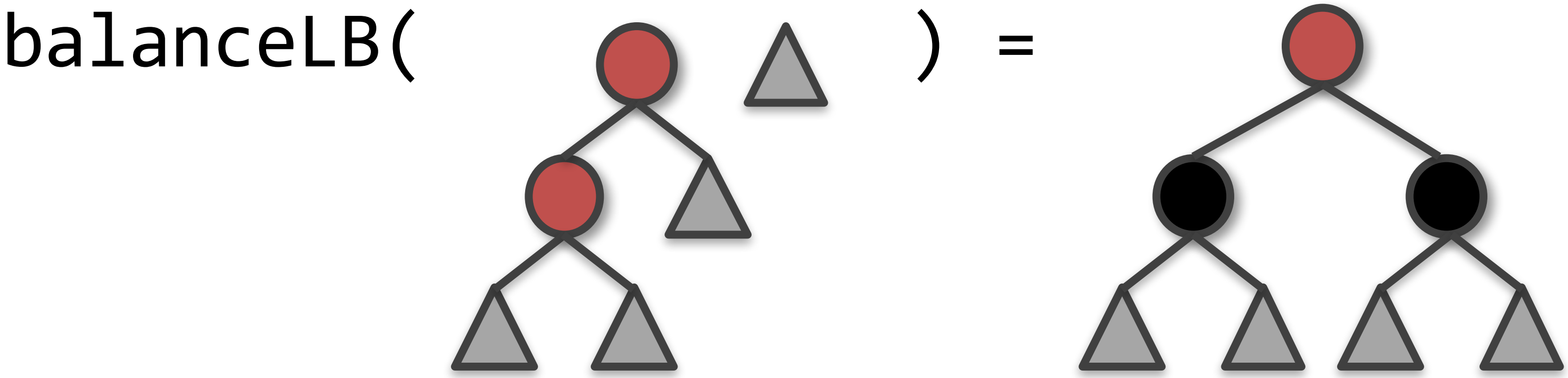
Type family

Singleton type

Type-term separation:
Singleton types provides runtime access to the color of the node in GHC.

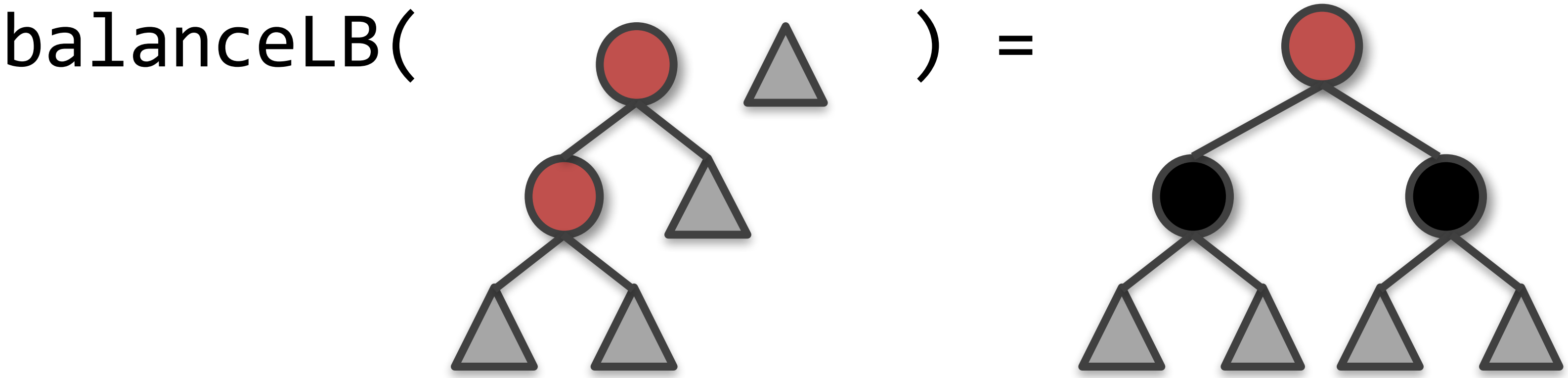
Agda

```
balanceLB : {n : ℕ}{c : Color} →
      AlmostTree n → A → Tree c n → HiddenTree (Suc n)
balanceLB (AT R (TR a x b) y c) z d =
  HR (TR (TB a x b) y (TB c z d))
balanceLB (AT R a x (TR b y c)) z d =
  HR (TR (TB a x b) y (TB c z d))
balanceLB (AT B a x b) y r = HB (TB (TB a x b) y r)
balanceLB (AT R E x E) y r = HB (TB (TR E x E) y r)
balanceLB (AT R (TB a w b) x (TB c y d)) z e =
  HB (TB (TR (TB a w b) x (TB c y d)) z e)
```



Haskell

```
balanceLB ::
      AlmostTree n -> A -> Tree c n -> HiddenTree (Suc n)
balanceLB (AT SR (TR a x b) y c) z d =
  HR (TR (TB a x b) y (TB c z d))
balanceLB (AT SR a x (TR b y c)) z d =
  HR (TR (TB a x b) y (TB c z d))
balanceLB (AT SB a x b) y r = HB (TB (TB a x b) y r)
balanceLB (AT SR E x E) y r = HB (TB (TR E x E) y r)
balanceLB (AT SR (TB a w b) x (TB c y d)) z e =
  HB (TB (TR (TB a w b) x (TB c y d)) z e)
```



Implementation of insert

- The Haskell version of insert is in lock-step with Agda version!
- But, are they the same? Not quite…

Agda:

```
insert : RBT → A → RBT
```

given a (valid) red-black tree and an element,
*insert **will** produce a valid red-black tree*

Haskell:

```
insert :: RBT -> A -> RBT
```

given a (valid) red-black tree and an element,
***if** insert produces a red-black tree, **then** it will be valid*

Difference: Totality

Adga requires all functions to be proved total

Haskell does not

- On one hand, Agda provide stronger guarantees about execution.
- On the other hand, totality checking is inescapable. Sometimes *not* reasoning about totality simplifies dependently-typed programming.

Not proving things is simpler

- Okasaki’s version of insert (simply typed): 12 lines of code
- Haskell version translated from Agda
  - 49 loc (including type defs & signatures)
  - precise return types for balance functions

```
balanceLB :: AlmostTree n -> A -> Tree c n -> HiddenTree (Suc n)
balanceLR :: HiddenTree n -> A -> Tree c n -> AlmostTree n
```

- Haskell version from scratch (see git repo)
  - 32 loc (including type defs & signatures)
  - more similar to Okasaki’s code
  - less precise return type for balance functions

```
balanceL :: Sing c ->
          AlmostTree n -> A -> Tree c n -> AlmostTree (Incr c n)
```

What is Dependently-Typed Haskell, really?

Dependently-Typed Haskell

- Flow sensitive type checking (e.g. GADTs)
  - Types influenced by pattern matching
  - "Singleton types" encode "dependent types"
  - Improvements to coverage checker improve TDD

- Rich type-level language enabling application specific invariants
  - Promoted datatypes
  - Type families (i.e. functions)
  - Type-level symbols & numbers
  - Pluggable constraint solvers
  - …

Ivory: Safe bit-level programming

```
-- | Convert an array of four 8-bit integers into a 32-
   bit integer.
test2 :: Def ('[Ref s (Array 4 (Stored Uint8))]
                :-> Uint32)
test2 = proc "test2" $ \arr -> body $ do
  a <- deref (arr ! 0)
  b <- deref (arr ! 1)
  c <- deref (arr ! 2)
  d <- deref (arr ! 3)
  ret $ ((safeCast a) `iShiftL` 24) .|
        ((safeCast b) `iShiftL` 16) .|
        ((safeCast c) `iShiftL` 8)  .|
        ((safeCast d) `iShiftL` 0)
```

https://github.com/GaloisInc/ivory

Length-preserving Convolution

```
convolve :: ∀ a b n. Vec n a -> Vec n b -> Vec n (a,b)
convolve xs ys =
    case walk xs of
        (r, Nil) -> r
        -- precondition [1]: ∀ n ∈ ℕ, m = n implies n - m = 0
        -- therefore this is an exhaustive match
    where
        walk :: ∀ m a. (m <= n) => Vec m a ->
                                    (Vec m (a,b), Vec (n - m) b)
        walk Nil = (Nil, ys)
        walk (a :. as) =
            case walk as of
                (r, b :. bs) -> ((a,b) :. r, bs)
                -- precondition [2]: ∀ n, m ∈ ℕ, n - m + 1 > 0
                -- therefore the list is non-empty
```

[Kenny Foner, Compose 2016]

Safe Database Access

```
type NameSchema  = [ Col "first" String, Col "last" String ]

printName :: Row NameSchema  -> IO ()
printName (first ::> last ::> _) = putStrLn (first ++ " " ++ last)

readDB classes_sch students_sch  = do
   classes_tab   <- loadTable "classes.table"  classes_sch
   students_tab  <- loadTable "students.table"  students_sch

   putStr "Whose students do you want to see? "
   prof <- getLine

   let joined  =
      Project
       (Select (field @"id" @Int `ElementOf`  field @"students")
          (Product
             (Select (field @"prof" :== Literal prof) (Read classes_tab))
             (Read students_tab)))
   rows <- query joined
   mapM_ printName rows
```

Haskell infers what rows need to be in the two different schemas. If these rows are not present, then the program will fail (at either compiletime or runtime).

[Richard Eisenberg, 2016]

What’s coming in GHC

Extensions in Progress (Eisenberg)

- Datatype promotion only works once
  - Cannot use dependently-typed programming at the type level
  - Some Agda structures have no GHC equivalent
  - Solution: Combine type and kind language together (-XTypeInType)
  - Current status: Merged into GHC HEAD, release coming soon!
- Type inference doesn't work well for type-level programming
  - Solution: Explicit type application
  - Nice interaction with HM, see ESOP 2016 paper
  - Current status: Merged into GHC HEAD, release coming soon!

- Singletons required
  - Solution: Add a PI type
  - Current status: planning stage, see Richard's dissertation draft

Conclusion

Haskell programmers can use dependent types*

… and we're actively working on the *

… but it is exciting to think about how *dependent*-type structure can help design programs

Thanks to: Simon Peyton Jones, Dimitrios Vytiniotis, Richard Eisenberg, Brent Yorgey, Geoffrey Washburn, Conor McBride, Adam Gundry, Iavor Diatchki, Julien Cretin, José Pedro Magalhães, David Darais, Dan Licata, Chris Okasaki, Matt Might, NSF

http://www.github.com/sweirich/dth