Depending on Types

Stephanie Weirich

University of Pennsylvania

Is GHC a dependently-typed language?

YES*

The Story of Dependently-typed Haskell

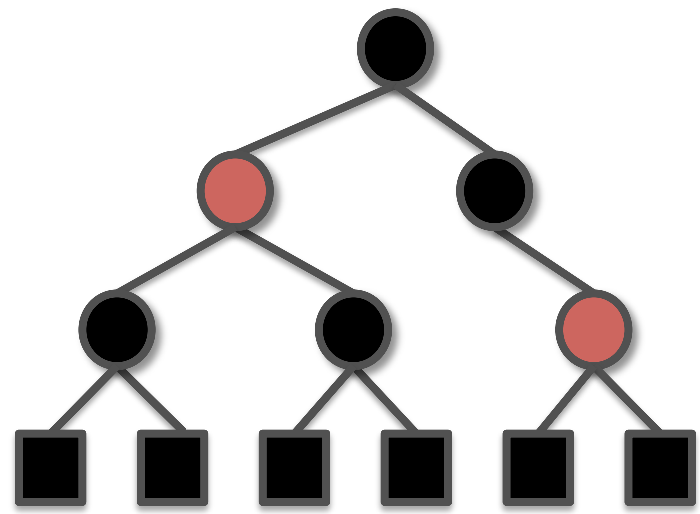
- *The Present*: Show how type system extensions work together to make GHC a dependently-typed language*
- *The Past*: Put those extensions in context, and talk about how they compare to dependent type theory
- *The Future*: Give my vision of where GHC should go and how we should get there

*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

See Conor McBride's talk

"How to keep your neighbours in order" later today

Insertion [Okasaki, 1993]

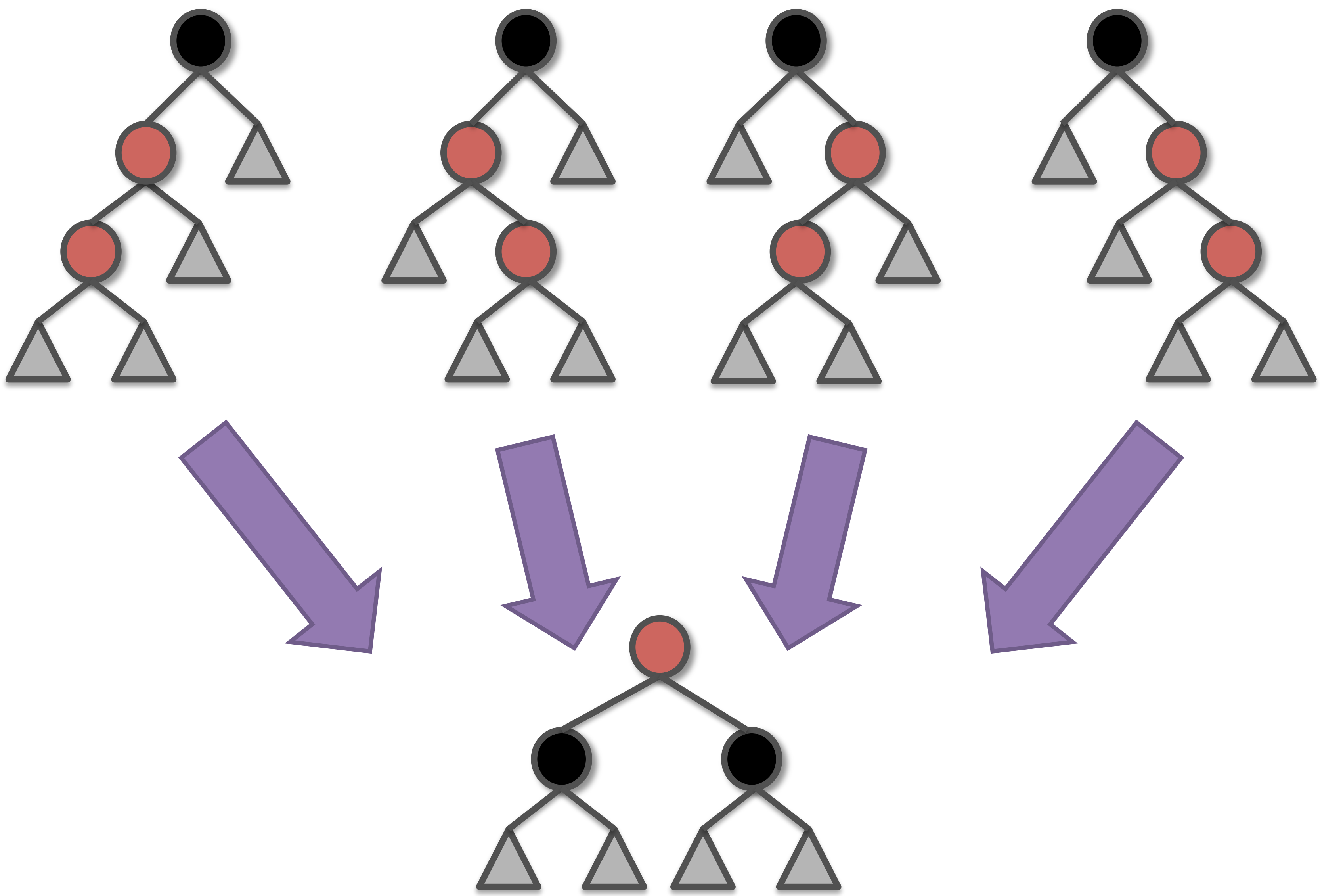
```
data Color = R | B
data Tree  = E | T Color Tree A Tree

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 color (ins a) y b
              | x > y     = balance 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.

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

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)
```

Star
like

Arguments of indexed datatypes vary by data constructor.

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

Indexed datatype

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

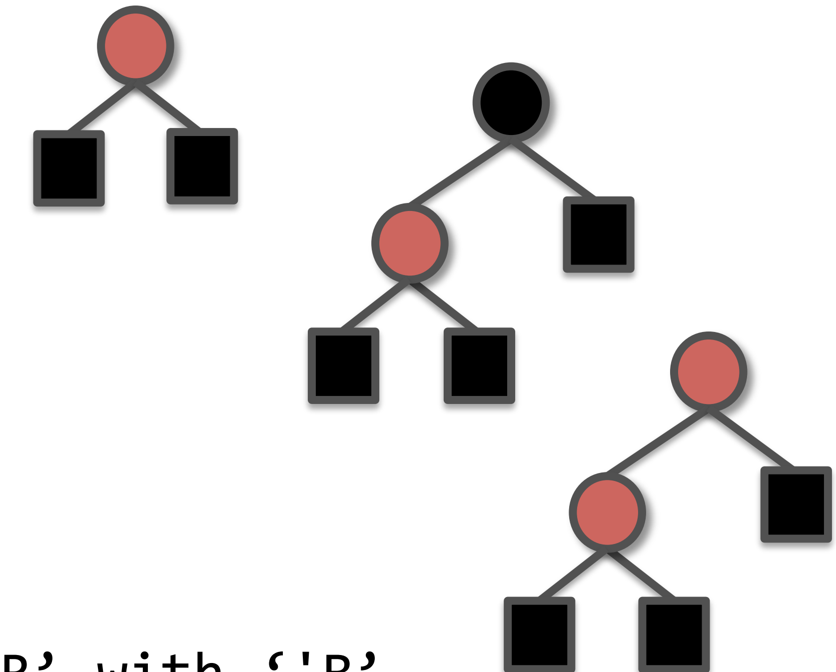
```
data Tree :: Color -> Nat -> * where
   E  :: Tree B Zero
   TR :: Tree B n -> A -> Tree B n -> Tree R n
   TB :: Tree c1 n -> A -> Tree c2 n -> Tree B (Suc n)
```

Haskell

GADTs - datatype arguments may vary by constructor

Datatype promotion – data constructors may be used as types

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
```

Agda and Haskell look similar

- Tree reversal swaps the order of elements in the tree
- Indexed types prove that black height is preserved and root color unchanged

```
rev : {n : ℕ} {c : Color} → Tree c n → Tree c n
rev E = E
rev (TR a x b) = TR (rev b) x (rev a)      -- a, b : Tree B n
rev (TB a x b) = TB (rev b) x (rev a)
```

Agda

Haskell

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

For the application of TR to type check, we must know that (rev b) and (rev a) are black trees of equal height.

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

Haskell

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

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

Where do these features come from?

Datatype promotion

- Recent extension

  [Yorgey, Weirich, Cretin, Peyton Jones, Vytiniotis, Magalhães; TLDI 2012]
  - Inspired by Strathclyde Haskell Extension (SHE) [McBride]
  - Introduced in GHC 7.4 [Feb 2012]

- Makes the type-term separation less brutal
  - Automatically allows data structures to be used in types
  - Includes kind-polymorphism (for promoting lists...)
  - Limitation: GADTs can't be promoted (*more on that later)

"It's crazy how cool the features in new GHC releases are. Other languages get patches to prevent some buffer overflow, we get patches to add an entirely new level of polymorphism." -mbetter on Reddit

GADTs

- Introduced in GHC 6.4 [March 2005]
- Many pre-cursors:
  - [Cheney, Hinze 2003] First-class phantom types (Haskell encoding)
  - [Xi, Chen, Chen 2003] Guarded Recursive Datatypes (ATS)
  - [Sheard, Pasalic 2004] Equality qualified types (Ωmega)
  - [Peyton Jones, Washburn, Weirich 2004] Generalized Algebraic Datatypes (Haskell primitive)
  - [Simonet, Pottier 2005] Guarded Algebraic Types (OCaml)
- Challenge: Integration with Hindley-Milner type inference
  - [Pottier, Régis-Gianis; POPL 2006]
  - [Peyton Jones, Vytiniotis, Washburn, Weirich; ICFP 2006]
  - [Sulzmann, Chakravarty, Peyton Jones; TLDI 2007]
  - [Schrijvers, Peyton Jones, Sulzmann, Vytiniotis; ICFP 2009]
- Could have been added to GHC much earlier…

Silly Type Families*
DRAFT

Lennart Augustsson and Kent Petersson
Department of Computer Sciences
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Email: augustss@cs.chalmers.se, kentp@cs.chalmers.se

September 10, 1994

Abstract

This paper presents an extension to standard Hindley-Milner type checking that allows constructors in data types to have non-uniform result types. We use Haskell as the sample language, [Hud92], but it should work for any language using H-M. It starts with some motivating examples and then shows the type rules for a simple language. Finally, it contains a sketch of how type deduction could be done.

1 Introduction

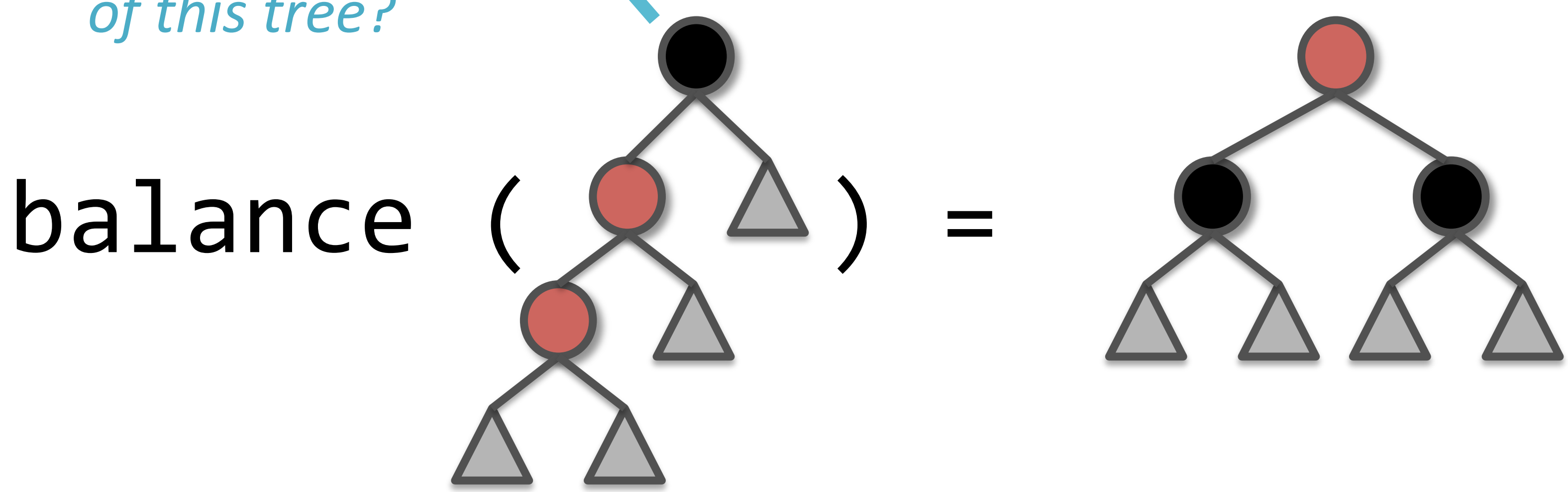
More of the usual ranting should go here.

This extension of H-M type checking has been floating around as a vague suggestion in the FP community for many years, but we do not know of any attempt to work out the details before. It has been inspired by how pattern matching works in ALF [Coq92, Mag], but we want to do type deduction as well as type checking.[1]

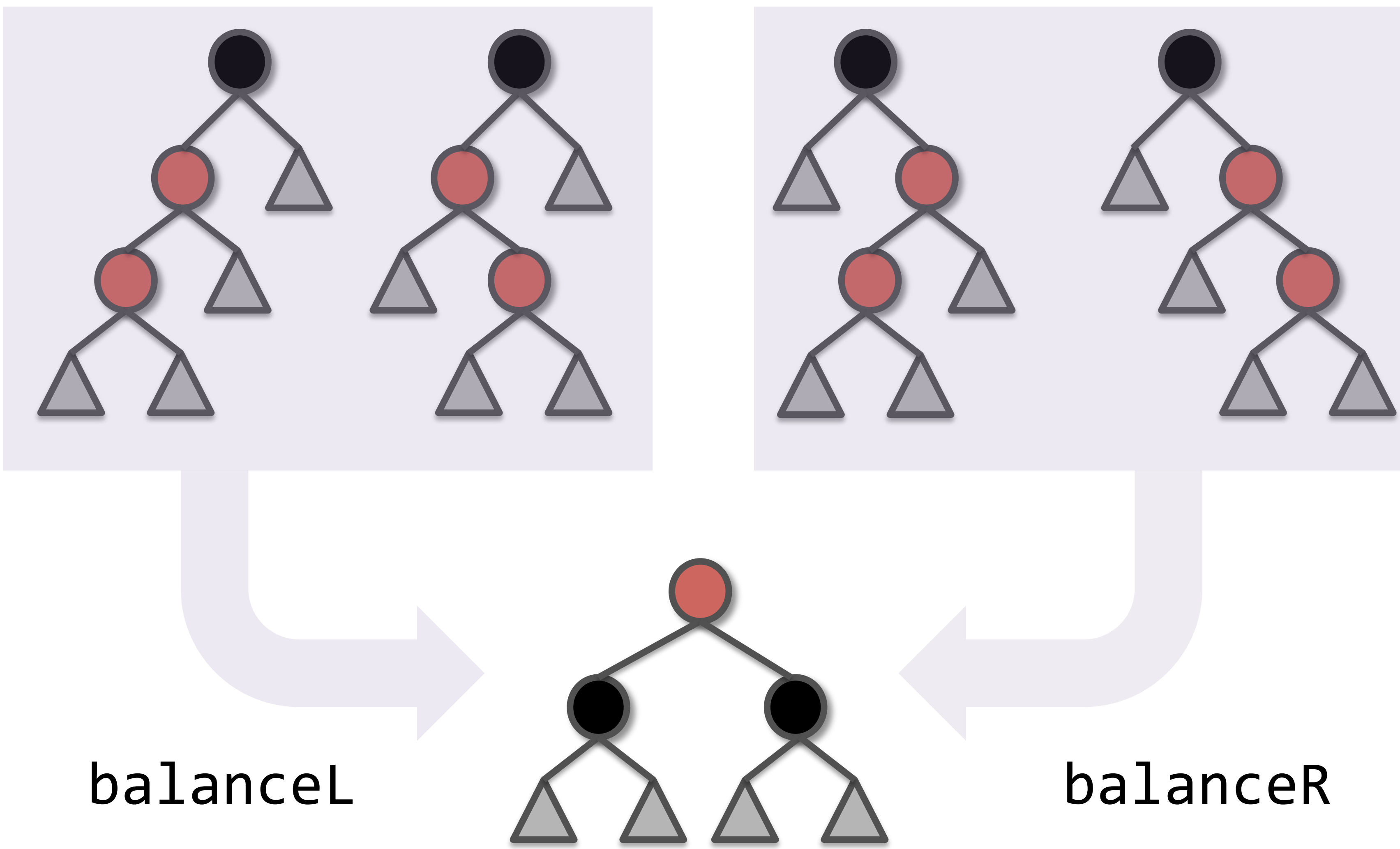
Insertion

How do we temporarily suspend the invariants during insertion?

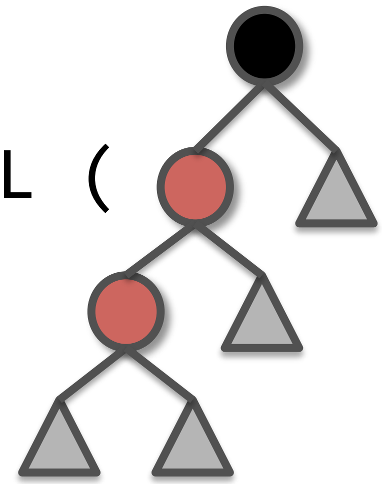
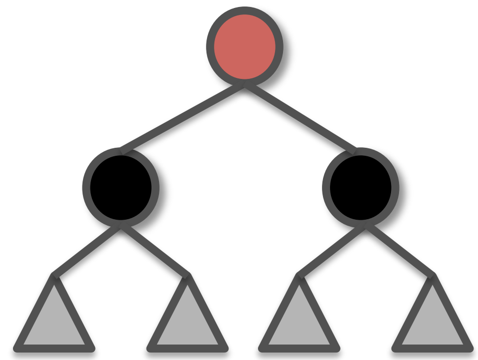
What is the type of this tree?

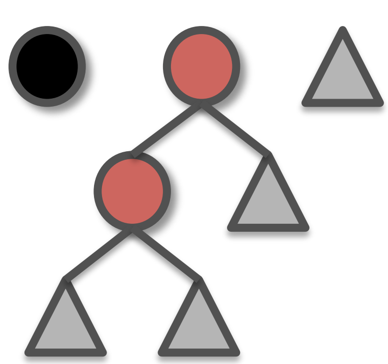
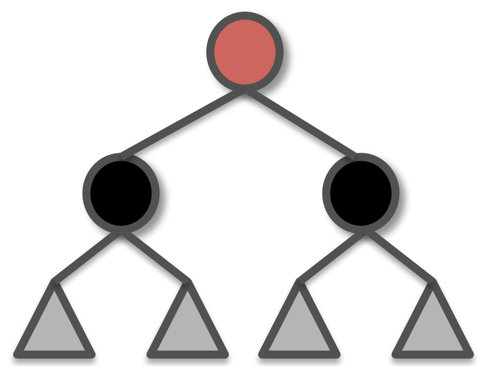


Split balance into two cases

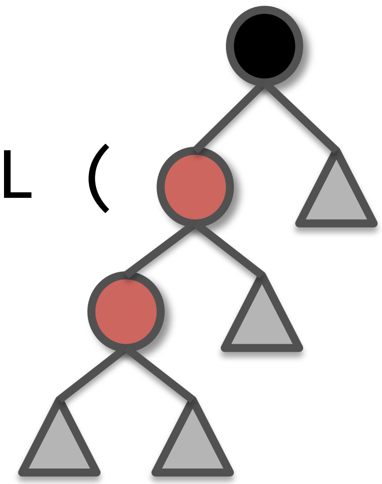
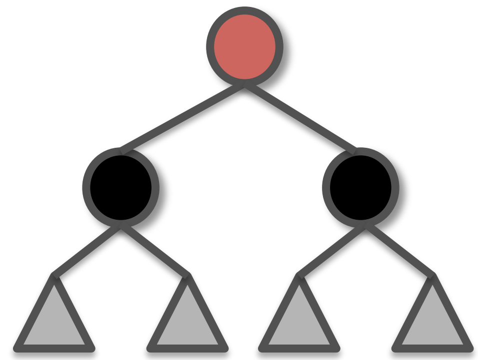


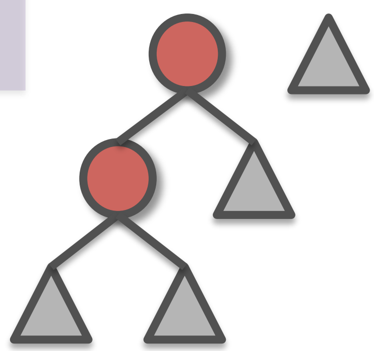
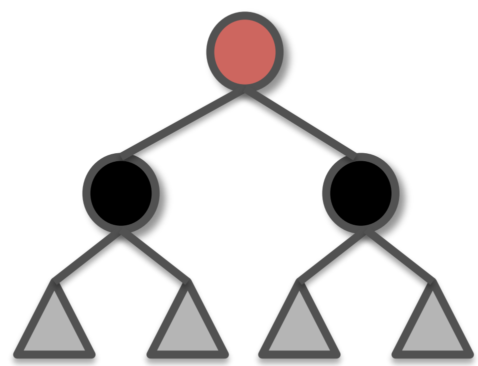
Decompose argument

balanceL () = 

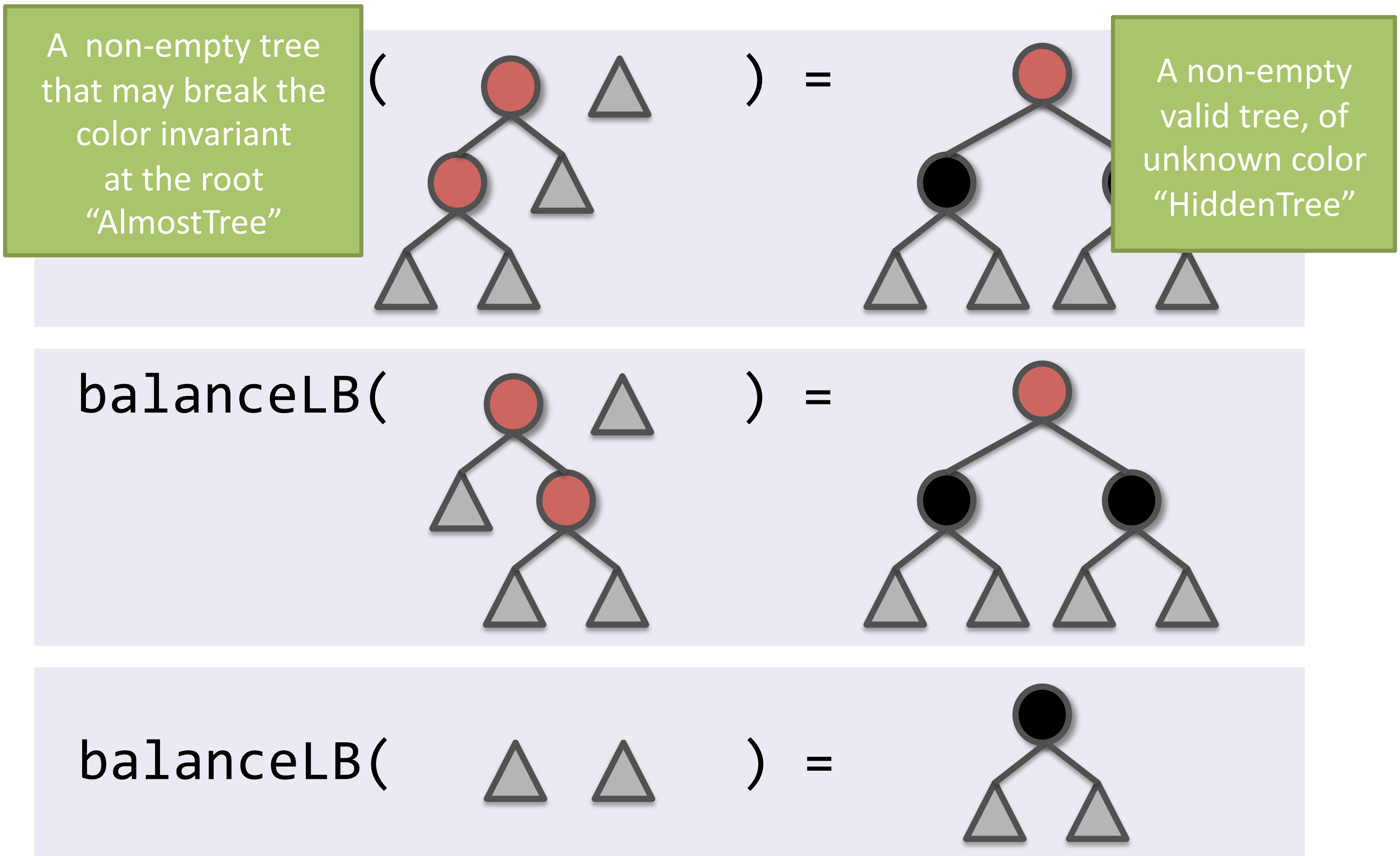
balanceL() = 

Specialize

balanceL () = 

balanceLB() = 

balanceLB : ??? → A → Tree c n → ???



Programming with types (Agda)

- A non-empty valid tree, of unknown color

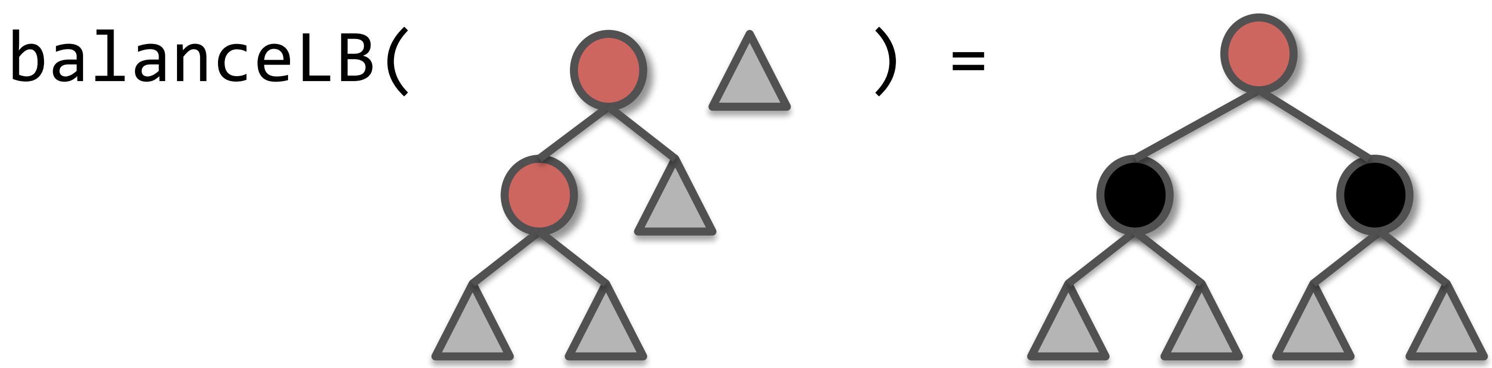
```
  data HiddenTree : ℕ → Set where
     HR : {m : ℕ} → Tree R m       → HiddenTree m
     HB : {m : ℕ} → Tree B (Suc m) → HiddenTree (Suc m)
```

- 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)
```

```
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.

Singleton types

- Standard trick for languages with a type-term distinction

  [Hayashi 1991][Xi, Pfenning 1998]

```
data Sing :: Color -> * where
   SR :: Sing R    -- SR only non-⊥ inhabitant of Sing R
   SB :: Sing B
```

- Can be as expressive as a full-spectrum language

  [Monnier, Haguenauer; PLPV 2010]

```
(x : A) → B
```

➔

```
forall (x :: A). Sing x -> B
```

- In GHC
  - Haskell library (singletons) automates translation, though limited by datatype promotion restrictions* [Eisenberg,Weirich; HS 2012]
  - Extensive use of singletons is painful* [Lindley,McBride; HS 2013]

Type families

- Motivation
  - Highly parameterized library interfaces

```
class IsList l where
   type Item l
   fromList :: [Item l] -> l
   toList :: l -> Item l
```

```
instance IsList Text where
   type Item = Char
   fromList = Text.pack
   toList = Text.unpack
```

  - Generic programming (type-indexed types)
  - Move to replace “logic programming” style of type-level computation (MPTC+FD) with “functional programming” style

- Challenge: Integration with Hindley-Milner type inference

  [Chakravarty, Keller, Peyton Jones, Marlow; POPL 2005]

  [Chakravarty, Keller, Peyton Jones; ICFP 2005]

  [Schrijvers, Peyton Jones, Chakravarty, Sulzmann; ICFP 2008]

  [Eisenberg, Vytiniotis, Peyton Jones, Weirich; POPL 2014]

Type families are not functions

- More restrictive:
  - No lambdas (must be named)
  - Application must be saturated
  - Restrictions on unification
- More expressive:
  - Can pattern match types, not just data
  - Equality testing is available for any kind

```
type family Id (a :: *) where
    Id a = a

instance Monad Id where
    …
```

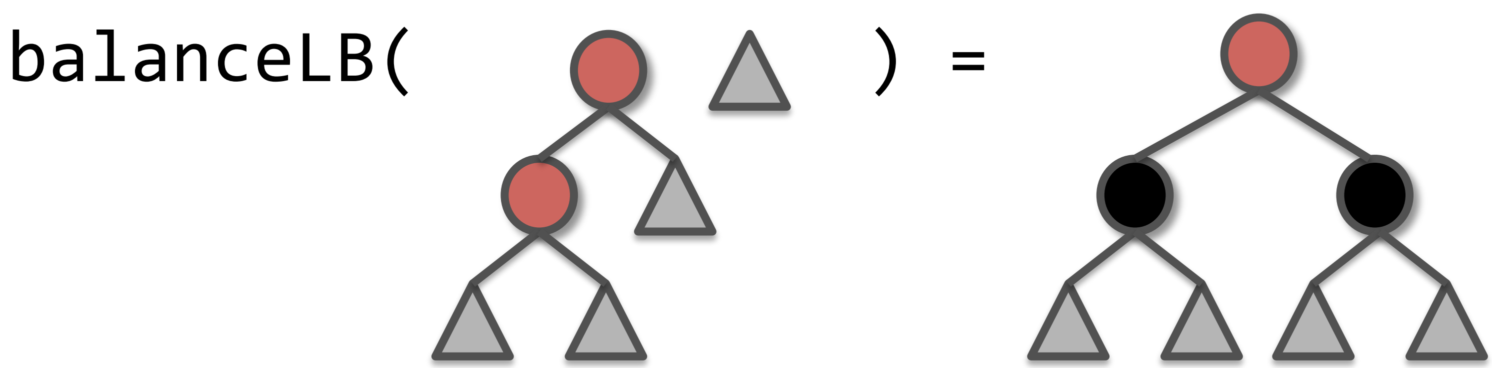
```
type family F (a :: Nat) where
    F Zero = Int

f :: F a -> F a
f x = x
```

```
type family Item (a :: *) where
    Item Text = Char
    Item [a]  = a
```

```
type family Eq (a :: k) (b :: k) :: Bool where
    Eq a a = True
    Eq a b = False
```

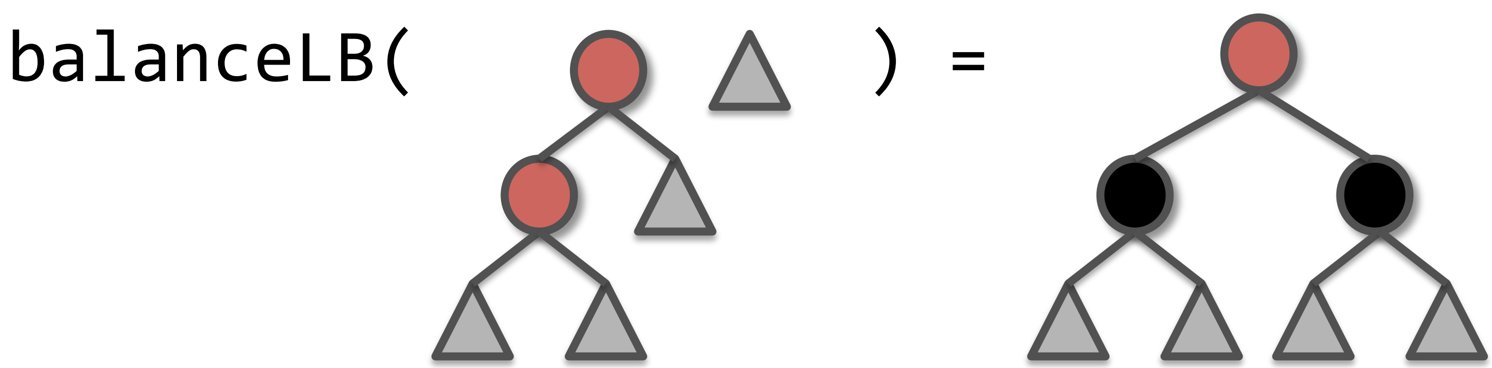
Agda

```
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)
```

Haskell



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.

It is simpler not to prove totality

- 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 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’s next for GHC

Extensions in Progress*

- Datatype promotion only works once
  - Cannot use dependently-typed programming at the type level
  - Some Agda programs have no GHC equivalent
  - Solution for GHC Core [Weirich,Hsu,Eisenberg; ICFP 2013]
  - Current status: Richard has core implementation done, integration with type inference in progress
  - Haskell Implementors Workshop talk "Dependent Haskell"

- GHC should have a real dependent type
  - Plan: Identify a shared subset of types and terms, introduce a new quantifier over that subset
  - Adam Gundry's dissertation provides a road map

Wishlist

TDD

Type-Driven Development

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**.

The ICFP 2015 program?

- (Optional) Totality checking for GHC
  - Pattern match exhaustiveness and termination
  - Language should give programmers the choice [Trellys]
- Extended type inference
  - Unsaturated/injective type families
  - Special purpose constraint solvers [TypeNats, Iavor Diatchki]
  - Programmable error messages
- IDE support
  - Automatic case splitting
  - Automatic code completion and code synthesis

Conclusion

GHC 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