Dependently-Typed Programming in GHC

Stephanie Weirich

University of Pennsylvania

Acknowledgements

- **Simon Peyton Jones**
- **Dimitrios Vytiniotis**
- Brent Yorgey
- Richard Eisenberg
- Justin Hsu
- Julien Cretin
- José Pedro Magalhães
- Iavor Diatchki
- Conor McBride

My (Overly Ambitious) Abstract

Is Haskell a dependently-typed programming language?

The Glasgow Haskell Compiler (GHC) type-system extensions, such as Generalized Algebraic Datatypes (GADTs), multiparameter type classes and type families, give programmers the ability to encode domain-specific invariants in types. Clever functional programmers have used these features to enhance the reasoning capabilities of static type checking. But really, how far have we come?

In this talk, I will (attempt to) answer the question ”Is it Dependent Types Yet?”, through examples, analysis and comparisons with modern full-spectrum dependently-typed languages, such as Agda and Coq. What sorts of dependently-typed programming can be done? What sorts of programming do these languages support that Haskell cannot? What should GHC learn from these languages, and conversely, what lessons can GHC offer in return?

Why Dependent Types?

- *Verification*: Dependent types express application-specific program invariants that are beyond the scope of existing type systems
- *Expressiveness*: Dependent types enable flexible interfaces, allowing more programs to be statically checked
- *Uniformity*: The same syntax and semantics is used for computations, specifications and proofs

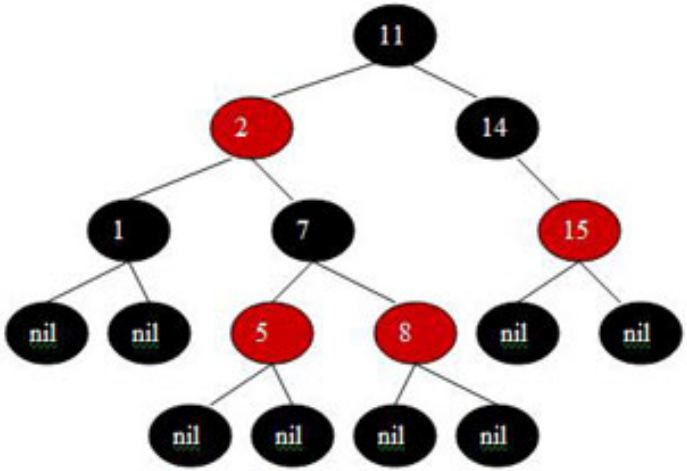
Program verification is “just programming”

Is GHC dependently-typed?

YES*

Example: Red Black Trees

- Application-specific invariants:
  - Binary search tree
  - *Root is black*
  - *Empty nodes at the leaves are black*
  - *Red nodes have black children*
  - From each node, every path to a leaf has the same number of black nodes
- As with all verification, choose trade-off between correctness and complexity
- [cf. Kahr's Coq version]



Red/Black Trees in Haskell

```
data Color = R | B
data T a = E | N Color (T a) a (T a)

color :: T a -> Color
color E = B
color (N c _ _ _) = c

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

cf. Okasaki 1998

Insert

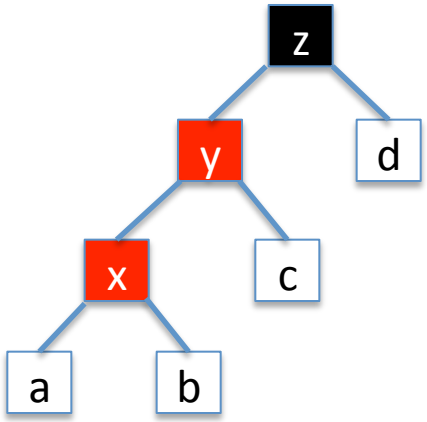
```
ins :: Ord a => a -> T a
ins x E = N R E x E
ins x s@(N c a y b)
  | x < y = balanceL c (ins x a) y b
  | x > y = balanceR c a y (ins x b)
  | _     = s

insert :: Ord a => a -> T
insert x t = blacken (ins x t) where
   blacken E = error “erk”
   blacken (N _ a x b) = N B a x b
```

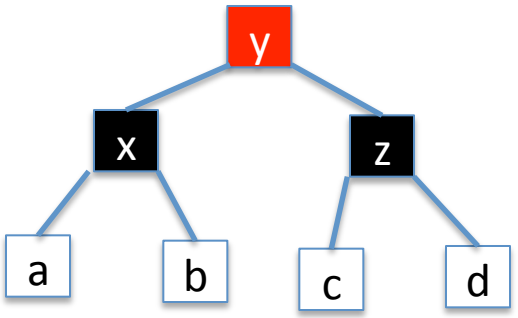
Temporarily suspend invariant:
result may have a red node
with a red child.

Two fixes:
- rebalance if Black on top
  of two Reds
- blacken if root is Red at the end

Rebalancing

```
balanceL :: Ord a =>
    Color -> T a -> a -> T a -> T a
balanceL B (N R (N R a x b) y c) z d =
    (N R (N B a x b) y (N B c z d))
balanceL B (N R a x (N R b y c)) z d =
    (N R (N B a x b) y (N B c z d))
balanceL c a x b = N c a x b
```





Can we statically enforce Red/Black tree invariants?

Root is black

Empty nodes at the leaves are black

Red nodes have black children

Alternative syntax for ADTs

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

data T (a :: *) where
  E :: T a
  N :: Color -> T a -> a -> T a -> T a
```

With normal datatypes, result of each constructor is always the same

GADT: Index by the root color

```
data T (c :: Color) (a :: *) where
  E :: T B a
  N ::
       T c1 a -> a -> T c2 a -> T c3 a
```

- *Indexed datatypes (GADTs)*
- *Datatype promotion*

```
data RBT (a :: *) where
     Root :: T B a -> RBT a
```

GADT: Index by the root color

```
data T (c :: Color) (a :: *) where
  E :: T B a
  N :: Valid c1 c2 c3 =>
       T c1 a -> a -> T c2 a -> T c3 a

class Valid (c1::Color) (c2::Color) (c3::Color)
instance Valid B B R
instance Valid c1 c2 B

data RBT (a :: *) where
     Root :: T B a -> RBT a
```

- *Indexed datatypes (GADTs)*
- *Datatype promotion*
- *Multiparameter type classes*

Runtime access to the node color

```
data Sing (c :: Color) where
     SR :: Sing R
     SB :: Sing B
```

"Singleton" GADTs provides a runtime witness to type-level data. Sing c isomorphic to Color but with more informative type

```
data T (c :: Color) (a :: *) where
  E :: T B a
  N :: Valid c1 c2 c3 => Sing c3
    -> T c1 a -> a -> T c2 a -> T c3 a

color :: T c a -> Sing c
color E = SB
color (N c _ _ _) = c
```

Static enforcement

```
ghci> let t1 = N SR E 1 E
ghci> :t t1
T R Integer
ghci> let t2 = N SB t1 2 (N SR E 3 E)
ghci> :t t2
T B Integer
ghci> let t3 = N SR t1 2 E
<interactive>:23:10:
    No instance for (Valid R B R)
      arising from a use of `N'
    Possible fix: add an instance
  declaration for (Valid R B R)
```

Insertion

```
ins :: Ord a => a -> T c a -> ???
ins x E = N SR E x E
ins x (N c a y b)
   | x < y = balanceL c (ins x a) y b
   | x > y = balanceR c a y (ins x b)
   | _     = N c a y b
```

Insertion produces a tree that "slightly" violates the invariant.
This tree is not representable.

```
insert :: Ord a => a -> RBT a -> RBT a
insert (Root t) = blacken (ins x t) where
   blacken :: ??? -> RBT a
```

Safe insertion?

- Data type preserves invariants too strongly

```
data T (c :: Color) (a :: *) where
  E :: T B a
  N :: Valid c1 c2 c3 => Sing c3
    -> T c1 a -> a -> T c2 a -> T c3 a
```

- A new data type to hold slightly wrong, nonempty trees that hides the top-level color

```
data Node (a :: *) where
  Node :: Sing c
    -> T c1 a -> a -> T c2 a -> Node a
```

How to insert into tree?

```
ins :: Ord a => a -> T c a -> Node a
ins x E = Node SR E x E
ins x (N c a y b)
   | x < y = balanceL c (ins x a) y b
   | x > y = balanceR c a y (ins x b)
   | _     = Node c a y b

insert :: Ord a => a -> RBT a -> RBT a
insert (Root t) = blacken (ins x t) where
    blacken :: Node a -> RBT a
    blacken (Node _ a x b) =
        Root (T SB a x b)
```

Rebalancing

```
balanceL :: Ord a => Sing c
      -> Node a -> a -> T c1 a -> Node a
balanceL SB (Node SR (N SR a x b) y c) z d =
    (Node SR (N SB a x b) y (N SB c z d))
balanceL SB (Node SR a x (N SR b y c)) z d =
    (Node SR (N SB a x b) y (N SB c z d))
balanceL c a x b = (Node c a x b)
```

Rebalancing

```
balanceL :: Ord a => Sing c
      -> Node a -> a -> T c1 a -> Node a
balanceL SB (Node SR (N SR a x b) y c) z d =
    (Node SR (N SB a x b) y (N SB c z d))
balanceL SB (Node SR a x (N SR b y c)) z d =
    (Node SR (N SB a x b) y (N SB c z d))
balanceL c a x b = (Node c a x b)
balanceL c (Node SR a'@(N SB _ _ _) x'
                    b'@(N SB _ _ _)) x b =
    (Node c (N SR a' x' b') x b)
balanceL c (Node SR a'@E x' b'@E) x b =
    (Node c (N SR a' x' b') x b)
balanceL c (Nod
    (Node c (N
```

Nontrivial exhaustiveness check:
`balanceL SR (Node SR (N SR a x b) y c) …`
impossible.

BlackHeight

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

Uses promotion again

```
data T (c :: Color) (n :: Nat) (a :: *)
  where
  E :: T B Z a
  N :: Valid c1 c2 c3 => Sing c3
      -> T c1 n a -> a -> T c2 n a
      -> T c3 ??? a
```

BlackHeight

```
data Nat = S Nat | Z
type family Inc (c::Color) (n::Nat) :: Nat
type instance (Inc B n) = S n
type instance (Inc R n) = n
```

Type families
i.e. Type-level functions

```
data T (c :: Color) (n :: Nat) (a :: *)
  where
  E :: T B Z a
  N :: Valid c1 c2 c3 => Sing c3
      -> T c1 n a -> a -> T c2 n a
      -> T c3 (Inc c3 n) a
```

Top-level tree

```
data T (c :: Color) (n :: Nat) (a :: *)
  where
  E :: T B Z a
  N :: Valid c1 c2 c3 => Sing n
      -> T c1 n a -> a -> T c2 n a
      -> T c3  (Inc c3 n) a
```

```
data RBT a where
  Root :: T B n a -> RBT a
```

Existential Datatype
type argument n
does not appear in result

Why is this dependent types?

- Informative case analysis
  - Pattern matching GADTs refines type checking
  - Singletons connect types and terms
- Type-level computation (i.e. “large eliminations”)
  - Type families
  - Datatype promotion
  - Kind polymorphism
- First-class polymorphism
  - Existential data constructors
  - Higher-rank types

How does GHC compare?

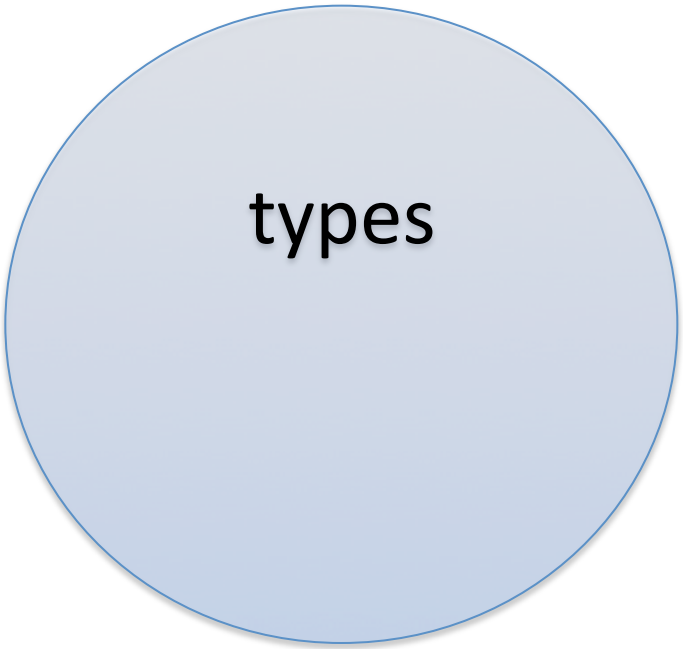
Coq

Agda

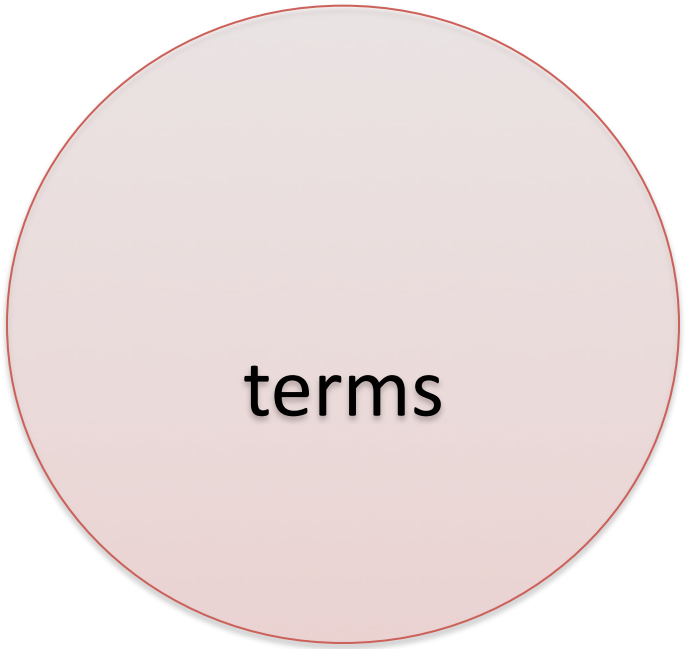
- Phase distinction
- “Partial” correctness
- Type inference & Qualified types

GHC: Phase distinction

- Terms can never appear in types
- Complete distinction between types and terms



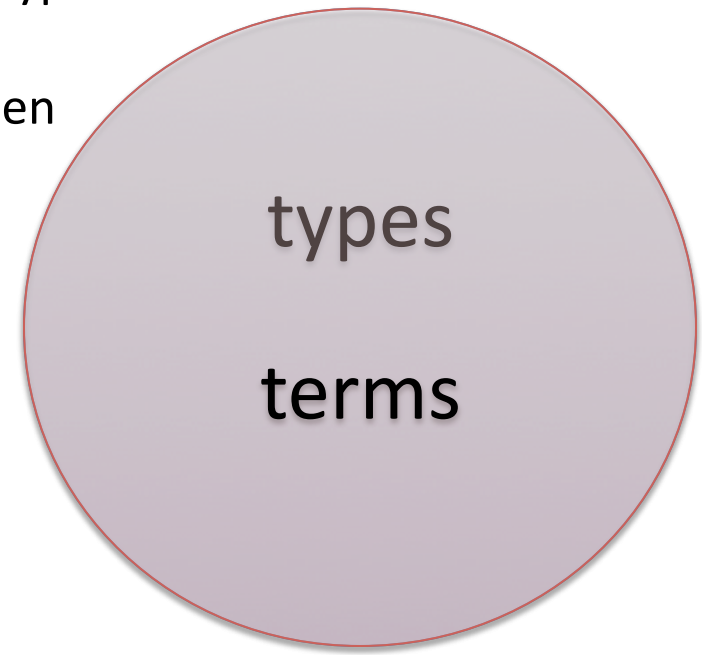
Can show equivalent



Can compile to machine code

Coq/Agda: No Phase Distinction

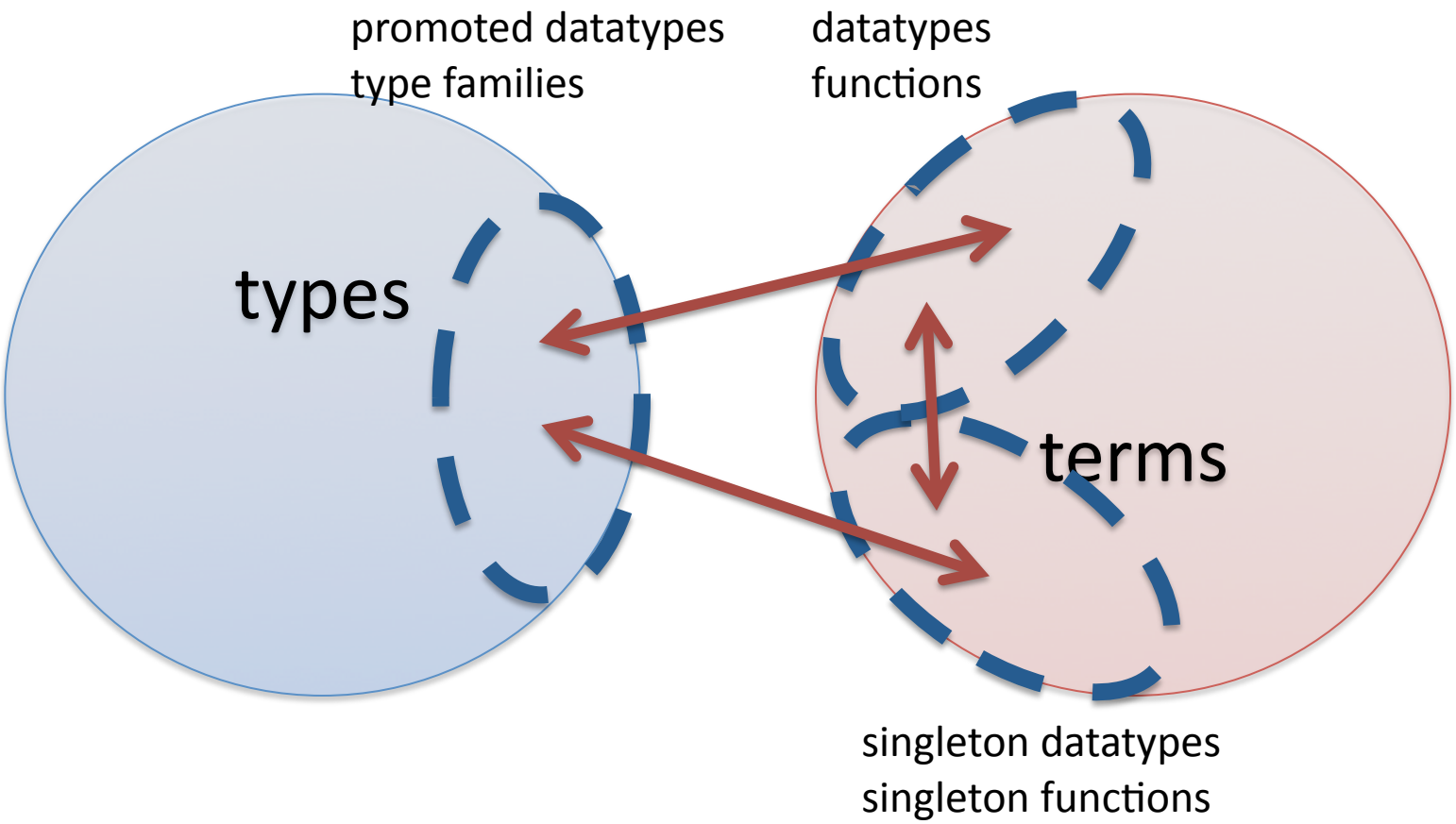
- Proof assistants based on Type Theory
- Terms can appear in types (trivially)
- No distinction between types and terms



Can show equivalent

Can compile to machine code

Getting around the phase distinction



Benefits of the phase distinction

- Prevents types from being "infected" by terms.
  - Don't have to decide arbitrary program equivalence
  - Don't have to reason about effectful code (esp. IO monad)

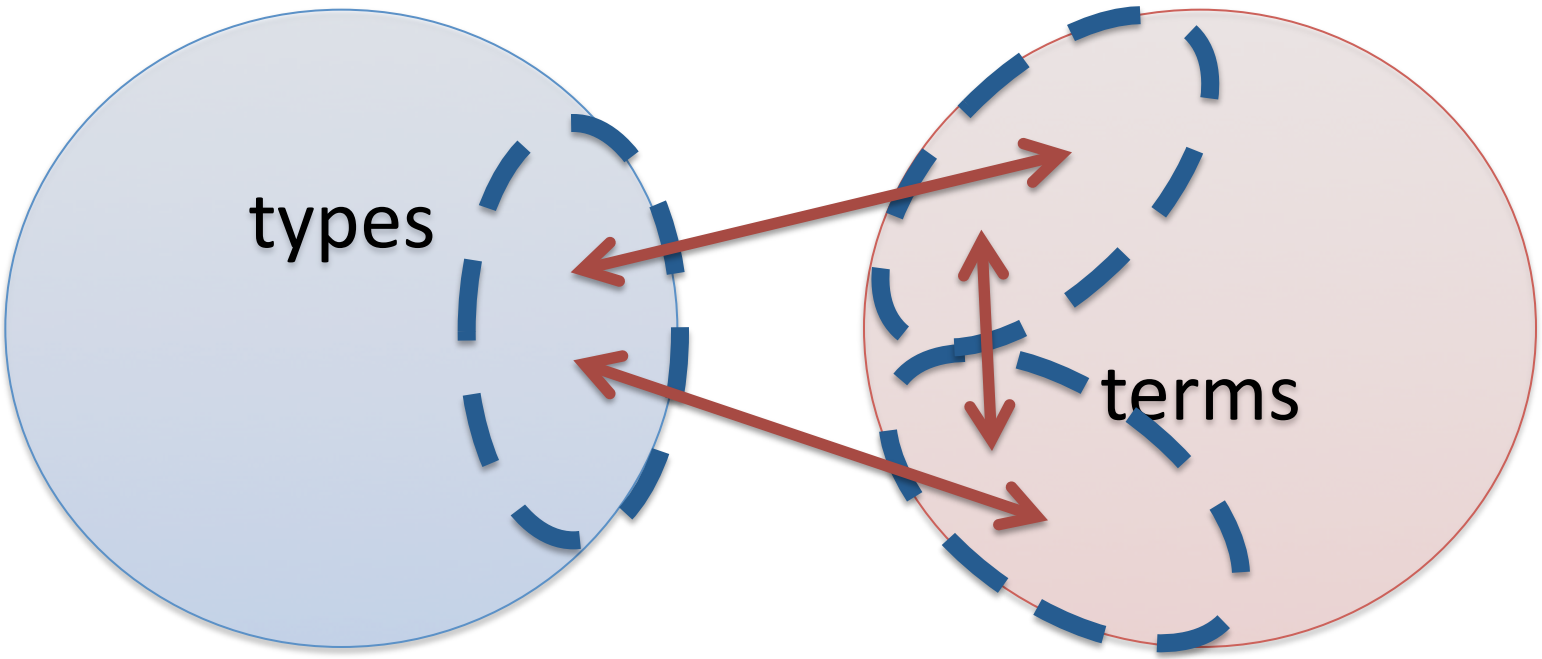
- Clearly separates specificational arguments from runtime arguments (important for efficient compilation)

```
f :: forall (c :: Color). Bool
f :: forall (c :: Color). Sing c -> Bool
```

- Parametricity (?)
  - Have parametricity results for GADTs, type-level computation [Vytiniotis & Weirich 2007, 2010]
  - Just figuring out parametricity for dependent types [Bernardy et. al 2010]

Problems with singletons

- Must define same datatype and functions multiple times
- Must convert between “vanilla” and “refined” versions of the data



Problems with singletons

```
data Bool = True | False   -- promotion makes Bool also a kind

data SBool (b :: Bool) where
  STrue :: SBool True
  SFalse :: SBool False

(&&) :: Bool -> Bool -> Bool
True && x = x
False && _ = False

type family (:&&:) (b1 :: Bool) (b2 :: Bool) :: Bool
type instance True :&&: x = x
type instance False :&&: x = False

(%&&) :: SBool b1 -> SBool b2 -> SBool (b1 :&&: b2)
True %&&  x = x
False %&& _ = False
```

Bool and SBool are isomorphic

```
fromSBool :: SBool a -> Bool
fromSBool STrue = True
fromSBool SFalse = False

data ExSBool = forall a. Ex (SBool a)
toSBool :: Bool -> ExSBool
toSBool True  = Ex STrue
toSBool False = Ex SFalse
```

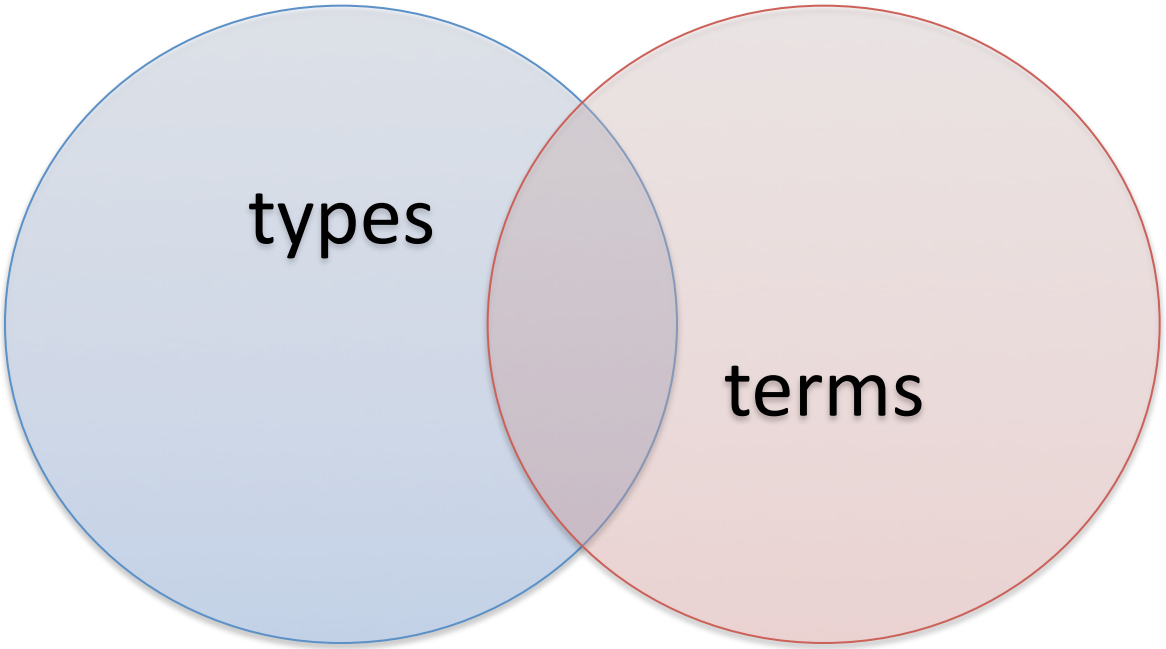
Blurring the phase distinction

- Strathclyde Haskell Extension (SHE Preprocessor) [McBride]
- Datatype Promotion [TLDI 2012, Yorgey et al.]
- Automatic Singletons [work in progress, Eisenberg]
  - Template Haskell code to generate duplicate code, isomorphism
  - Uniform interface to singleton code

```
newtype family Sing (a :: k)   -- kind-indexed function


type instance Sing (a :: Bool) where
     STrue  :: Sing True
     SFalse :: Sing False
type instance Sing (a :: Maybe k) where
     SNothing :: Sing Nothing
     SJust :: Sing (b :: k) -> Sing (Just b)
```

Future Work

- Is this enough for practical programming?
  - Overheads nudge towards *limited* use of dependency, maybe that's a good thing?



"Partial" Correctness

- In Coq/Agda *all* programs must terminate
  - Under Curry-Howard Isomorphism all programs are proofs
  - Logical consistency: "False" is an uninhabited type
- There is no termination checking in GHC
  - Good news: Much easier for programmers who don't do proofs
  - Bad news: *No logical consistency*. Every type and every kind is inhabited in GHC
- Have we really proved anything?

“Partial” Correctness

```
data Eq (a :: k) (b :: k) where
    Refl ::  Eq a a

symmetry ::  Eq a b  ->  Eq b a
symmetry Refl = Refl
```

Can define an equality proposition
in Haskell, asserting the equality between
two types. Note: kind polymorphic!

“Partial” Correctness

```
data Eq (a :: k) (b :: k) where
    Refl ::  Eq a a

symmetry ::  Eq a b  ->  Eq b a
~~symmetry Refl = Refl~~
symmetry _ = error “Just Kidding”
```

Can define an equality proposition in Haskell, asserting the equality between two types. Note: kind polymorphic!

“Partial” Correctness

```
insert :: Ord a => a -> RBT a -> RBT a
```

- In Agda/Coq, know that insertion function is totally correct
- In GHC, type soundness asserts that if *insert terminates without error and we get a tree*, then the tree satisfies RBT invariants
  - But insert could trigger a pattern match failure or diverge
  - And the tree could have _|_ elements!

Future directions?

- Layer totality checking on top of GHC?
  - Exhaustiveness checking for pattern matching difficult
- Combine partial and total dependently-typed language together
  - Type system indicates termination behavior
  - Proofs are a sublanguage of programs
  - Proofs can reason about (potentially) nonterminating programs
  - Programs can manipulate first-class proofs

- Related work
  - TRELLYS [Kimmel et al. PLPV 12, Casinghino et al. MSFP 12]
  - F-Star [Swamy et al. ICFP 11, POPL 12]

Type Inference

What *is* type inference?

- *Inferring the types of values*, including the interface to functions.

- *Implicit argument inference*
  Supplying specificational arguments to functions that can be inferred from type checking

```
f :: forall a. a -> a
f (\x -> x + 1)
```

- *Constraint resolution*
  Satisfying constraints at the applications of functions

```
insert :: Ord a => a -> RBT a -> RBT a
insert “a”  t
```

GHC's Secret Weapon

Constraint-based type inference

Comprensive type inference story for type classes, GADTs, type functions, etc. based on constraints

OUTSIDEIN(X) [Schrijvers et al. 2009, Vytiniotis et al. JFP 2010]

Type classes and dependent types

- Specificational constraints

```
class Valid (c1::Color) (c2::Color) (c3::Color)
instance Valid B B R
instance Valid c1 c2 B
```

- "Fake" implicit arguments

```
class SingI (a :: k) where
  sing :: Sing (a :: k)
instance SingI (True :: Bool) where sing = STrue
instance SingI (False :: Bool) where sing = SFalse
```

- Type classes being adopted by Coq [Gonthier et al. 2011] and Agda [Devriese & Piessens, 2011]

Future research w/ constraints

Constrained polymorphism provides the framework for new type-system extensions:

- First-class constraints [Orchard and Schrijvers 2010, Bolingbroke 2011]

- TypeNats [Diatchki, current work]
  - Natural numbers + equality and inequality axioms
  - Solved (now) by adding to GHC's constraint solver
  - Future: external/pluggable constraint solver?

Program verification is "just **functional** and **logic** programming"?

MPTCs vs. Type functions

- Open question: how to express type-level computation?

- Type families (aka type functions):

```
type family Inc (c::Color) (n::Nat) :: Nat
type instance (Inc B n) = S n
type instance (Inc R n) = n
```

- MPTC

```
type class Inc (c::Color) (n::Nat) (m ::Nat) | c n -> m
type instance Inc B n (Sn)
type instance Inc R n n
```

- Type functions more natural to functional programmers and more similar to Agda/Coq.
- GHC restricts how they can be used.

Future research w/ type functions

- In Haskell, can only generalize over datatype constants, not type functions.
  - Gives strong inference guarantee:
    `m1 a1 = m2 a2` implies `m1 = m2` and `a1 = a2`
  - Necessary to typecheck common idioms: `[return 2,return 3]`

- Type functions *must* be saturated
  - Can't instantiate Monad type class with type function (such as Id)
  - `F a1 ~ F a2` does not imply `a1 ~ a2`

- Can we do better?
  - Mark some type functions as injective?
  - Explicit instantiation for type functions?

Conclusion

- GHC continues to be a productive testlab for type system extensions
  - GADTs
  - Type Families (type level functions)
  - Existential datatypes (and constraints)
  - First-class polymorphism
  - Datatype Promotion
  - Kind polymorphism
  - Constraint Kinds
  - *TypeNats*
  - *Automatic singletons*

- On the way to the world's finest dependently-typed programming language

Thanks!