Type Theorists HATE Him!

Learn this ONE WEIRD TRICK to fake dependent types in a language that doesn't support them

LEARN THE TRUTH NOW

Ryan Scott

rgscott@indiana.edu

github.com/RyanGlScott

September 1, 2017

Dependent types

Dependent types... in Idris

```
mkSingle : (x : Bool) ->
           if x then Nat else List Nat
mkSingle True  = 0
mkSingle False = []
```

Dependent types… in Haskell?

```
mkSingle :: (x :: Bool) ->
            if x then Nat else [Nat]
mkSingle True  = 0
mkSingle False = []
```

Dependent types… in Haskell?

```
mkSingle :: (x :: Bool) ->
            if x then Nat else [Nat]
mkSingle True  = 0
mkSingle False = []
```

```
<interactive>:1:28: error: parse error on input ‘if’
```

:(

There's hope yet

- Haskell *can* support dependent types, if you're willing to squint
- We'll need to enable a modest number of GHC extensions

There's hope yet

- Haskell *can* support dependent types, if you're willing to squint
- We'll need to enable a modest number of GHC extensions

```
{-# LANGUAGE DefaultSignatures          #-}
{-# LANGUAGE EmptyCase                  #-}
{-# LANGUAGE ExistentialQuantification  #-}
{-# LANGUAGE FlexibleContexts           #-}
{-# LANGUAGE FlexibleInstances          #-}
{-# LANGUAGE GADTs                      #-}
{-# LANGUAGE InstanceSigs               #-}
{-# LANGUAGE KindSignatures             #-}
{-# LANGUAGE RankNTypes                 #-}
{-# LANGUAGE ScopedTypeVariables        #-}
{-# LANGUAGE TemplateHaskell            #-}
{-# LANGUAGE TypeFamilies               #-}
{-# LANGUAGE TypeInType                 #-}
{-# LANGUAGE TypeOperators              #-}
{-# LANGUAGE UndecidableInstances       #-}
```

Singleton types

```
data family Sing :: k -> Type
```

Singleton types

```
data family Sing :: k -> Type
```

```
data instance Sing :: Bool -> Type where
  SFalse :: Sing False
  STrue  :: Sing True
```

Singleton types

```
data family Sing :: k -> Type
```

```
data instance Sing :: Bool -> Type where
  SFalse :: Sing False
  STrue  :: Sing True
```

Singleton types

```
data family Sing :: k -> Type
```

```
data instance Sing :: Bool -> Type where
  SFalse :: Sing False
  STrue  :: Sing True
```

Singleton types

```
data family Sing :: k -> Type
```

```
data instance Sing :: Bool -> Type where
  SFalse :: Sing False
  STrue  :: Sing True

data instance Sing (z :: Bool)
  = (z ~ False) => SFalse
  | (z ~ True)  => STrue
```

Singleton types

```
data family Sing :: k -> Type
```

```
data instance Sing :: Bool -> Type where
  SFalse :: Sing False
  STrue  :: Sing True

data instance Sing (z :: Bool)
  = (z ~ False) => SFalse
  | (z ~ True)  => STrue
```

Singleton types

```
data family Sing :: k -> Type
```

```
data Nat = Z | S Nat

data instance Sing :: Nat -> Type where
  SZ :: Sing Z
  SS :: Sing (n :: Nat) -> Sing (S n)
```

Singleton types

```
data family Sing :: k -> Type
```

```
data Nat = Z | S Nat

data instance Sing :: Nat -> Type where
  SZ :: Sing Z
  SS :: Sing (n :: Nat) -> Sing (S n)


data instance Sing (z :: Nat)
  = (z ~ Z) => SZ
  | forall (n :: Nat). (z ~ S n) => SS n
```

Dependent types... in Idris (redux)

```
mkSingle : (x : Bool) ->
           if x then Nat else List Nat
mkSingle True  = 0
mkSingle False = []
```

Dependent types… in Haskell? (Heck yeah!)

```
type family
  If (c :: Bool) (t :: k) (f :: k) :: k where
  If True  t f = t
  If False t f = f
```

Dependent types... in Haskell? (Heck yeah!)

```haskell
type family
  If (c :: Bool) (t :: k) (f :: k) :: k where
  If True  t f = t
  If False t f = f

mkSingle :: Sing (x :: Bool) ->
            If x Nat [Nat]
mkSingle STrue  = 0
mkSingle SFalse = []
```

Dependent types... in Haskell? (Heck yeah!)

```
type Π = Sing
```

Dependent types... in Haskell? (Heck yeah!)

```
type Π = Sing
```

```
mkSingle :: Π (x :: Bool) ->
            If x Nat [Nat]
mkSingle STrue  = 0
mkSingle SFalse = []
```

What else can we ~~fake~~ emulate?

- Dependent pattern matching

What else can we ~~fake~~ emulate?

- Dependent pattern matching... in Idris

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect Z a
  Cons : a -> Vect n a -> Vect (S n) a
```

What else can we ~~fake~~ emulate?

- λ Dependent pattern matching... in Idris

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect Z a
  Cons : a -> Vect n a -> Vect (S n) a


len : Vect n a -> Nat
len Nil           = 0
len (Cons x xs)   = 1 + len xs
```

What else can we ~~fake~~ emulate?

- λ Dependent pattern matching... in Idris

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect Z a
  Cons : a -> Vect n a -> Vect (S n) a


len : {n : Nat} -> Vect n a -> Nat
len {n=Z}   Nil          = 0
len {n=S k} (Cons x xs) = 1 + len xs
```

What else can we ~~fake~~ emulate?

- Dependent pattern matching… in Haskell!

What else can we ~~fake~~ emulate?

- Dependent pattern matching… in Haskell!

```
data Vect :: Nat -> Type -> Type where
  Nil  :: Vect Z a
  Cons :: a -> Vect n a -> Vect (S n) a
```

What else can we ~~fake~~ emulate?

- Dependent pattern matching… in Haskell!

```
data Vect :: Nat -> Type -> Type where
  Nil  :: Vect Z a
  Cons :: a -> Vect n a -> Vect (S n) a
```

```
len :: Sing (n :: Nat) -> Vect n a -> Nat
len SZ     Nil          = 0
len (SS k) (Cons x xs) = 1 + len xs
```

What else can we ~~fake~~ emulate?

- λ Dependent pattern matching... in Haskell!

```
class SingI (a :: k) where
  sing :: Sing (a :: k)

instance SingI Z where
  sing = SZ

instance SingI n => SingI (S n) where
  sing = SS sing
```

What else can we ~~fake~~ emulate?

- Dependent pattern matching... in Haskell!

```
len :: Sing (n :: Nat) -> Vect n a -> Nat
len SZ     Nil          = 0
len (SS k) (Cons x xs)  = 1 + len xs
```

What else can we ~~fake~~ emulate?

- Dependent pattern matching… in Haskell!

```
len :: Sing (n :: Nat) -> Vect n a -> Nat
len SZ     Nil          = 0
len (SS k) (Cons x xs) = 1 + len xs

len’ :: forall (n :: Nat). SingI n =>
        Vect n a -> Nat
len’ = len (sing :: n)
```

- With enough elbow grease, one can simulate a great deal of dependently typed code
- Impress your friends at the bar! Be the envy of your family!
- `http://hackage.haskell.org/package/singletons`

Any questions?