Dependent Types in GHC

John Leo

Halfaya Research

April 8, 2017

References

- Pointer to this talk on github
- Pointer to references

What are Dependent Types?

$$
\begin{aligned}
&\mathsf{data}\ \mathbb{N} : \mathsf{Set}\ \mathsf{where}\\
&\quad \mathsf{zero} : \mathbb{N}\\
&\quad \mathsf{suc} : \mathbb{N} \to \mathbb{N}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Vec}\ (A : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}\ \mathsf{where}\\
&\quad [] : \mathsf{Vec}\ A\ \mathsf{zero}\\
&\quad \_{::}\_ : \{n : \mathbb{N}\} \to A \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n)
\end{aligned}
$$

Vector Append

$$\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathrm{zero} + n = n$$
$$\mathrm{suc}\ m + n = \mathrm{suc}\ (m + n)$$

$$\_{++}\_ : \{A : \mathrm{Set}\} \to \{m\ n : \mathbb{N}\} \to$$
$$\quad \mathrm{Vec}\ A\ m \to \mathrm{Vec}\ A\ n \to \mathrm{Vec}\ A\ (m + n)$$
$$[]\ {+}{+}\ y = y$$
$$(x :: xs)\ {+}{+}\ y = x :: (xs\ {+}{+}\ y)$$

Vector Lookup

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} → Fin n → Fin (suc n)

lookup : {A : Set} → {n : ℕ} → Fin n → Vec A n → A
lookup ()    [] -- can be omitted
lookup zero (x :: _ ) = x
lookup (suc n) (_ :: xs) = lookup n xs
```

Some Basic Types

```
data Bool : Set where
  true : Bool
  false : Bool

data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a
```

Vector Lookup 2

```
_¡_ : ℕ → ℕ → Bool
_      ¡ zero = false
zero ¡ suc n = true
suc m ¡ suc n = m ¡ n
```

```
lookup' : {A : Set} → {n : ℕ} →
  (m : ℕ) → m ¡ n ≡ true → Vec A n → A
lookup' _     ()     []
lookup' zero refl (x :: _ ) = x
lookup' (suc m) p (_ :: xs) = lookup' m p xs
```

Dependent Types

Advantages over other types

Applications

- Better correctness guarantees for software.
- Mechanical verification of mathematics.

Dependent Types are Not New

Timeline

The Golden Age is Now

- DeepSpec, etc.
- BigProof, etc.

Killer Apps for Typed Functional Programming

- Big Data
- Program Correctness
- Security

Robert Harper

> *Eventually all the arbitrary programming languages are going to be just swept away with the oceans, and we will have the permanance of constructive, intuistionistic type theory as the master theory of computation—without doubt, in my mind, no question. So, from my point of view—this is a personal statement—working in anything else is a waste of time.*

CMU Homotopy Type Theory lecture 1, 52:56–53:20.

Dependent Types in Haskell

Richard Eisenberg's PhD Thesis

1. Introduction
2. Preliminaries
3. Motivation
4. Dependent Haskell
5. PICO: The Intermediate Language
6. Type Inference and Elaboration, or how to BAKE a PICO
7. Implementation
8. Related and Future Work

Haskell Nat

```haskell
{-# LANGUAGE ExplicitForAll, GADTs, TypeFamilies,
             TypeOperators, TypeInType #-}
import Data.Kind (Type)

data Nat :: Type where
  Zero :: Nat
  Succ :: Nat -> Nat
```

```agda
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

Haskell Vector

```
data Vec :: Type -> Nat -> Type where
  VNil  :: ∀ (a :: Type). Vec a 'Zero
  VCons :: ∀ (a :: Type)(n :: Nat).
    a -> Vec a n -> Vec a ('Succ n)
```

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

Haskell Plus

```
type family (m :: Nat) + (n :: Nat) where
  Zero    + n = n
  'Succ m + n = 'Succ (m + n)
```

$$\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\text{zero} + n = n$$
$$\text{suc } m + n = \text{suc } (m + n)$$

Haskell Append

```
(++) :: ∀ (a :: Type)(m :: Nat)(n :: Nat).
        Vec a m -> Vec a n -> Vec a (m + n)
VNil      ++ v = v
VCons x u ++ v = VCons x (u ++ v)
```

$$
\begin{aligned}
&\_{+}{+}\_ : \{A : \mathsf{Set}\} \to \{m\ n : \mathbb{N}\} \to \\
&\quad \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (m + n) \\
&[]\ {+}{+}\ y = y \\
&(x :: xs)\ {+}{+}\ y = x :: (xs\ {+}{+}\ y)
\end{aligned}
$$

Haskell Fin

```
data Fin :: Nat -> Type where
  FZero :: ∀ (n :: Nat).          Fin (Succ n)
  FSucc :: ∀ (n :: Nat). Fin n -> Fin (Succ n)
```

data Fin : $\mathbb{N} \rightarrow$ Set where
  zero : $\{n : \mathbb{N}\} \rightarrow$ Fin (suc n)
  suc : $\{n : \mathbb{N}\} \rightarrow$ Fin $n \rightarrow$ Fin (suc n)

Haskell Lookup

```
lookup :: ∀ (a :: Type)(n :: Nat). Fin n -> Vec a n -> a
lookup FZero     (VCons x _)  = x
lookup (FSucc n) (VCons _ xs) = lookup n xs
```

```
lookup : {A : Set} → {n : ℕ} → Fin n → Vec A n → A
lookup ()   [] -- can be omitted
lookup zero (x :: _ ) = x
lookup (suc n) (_ :: xs) = lookup n xs
```

Haskell ≡ and <

```
data (a :: k) :~: (b :: k) where
  Refl :: ∀ (k :: Type)(a :: k). a :~: a

type family (m :: Nat) < (n :: Nat) where
  _         < Zero      = False
  'Zero     < ('Succ _) = True
  ('Succ m) < ('Succ n) = m < n
```

```
data _≡_ {A : Set} : A → A → Set where
  refl : {a : A} → a ≡ a

_¡_ : ℕ → ℕ → Bool
_     ¡ zero = false
zero ¡ suc n = true
suc m ¡ suc n = m ¡ n
```

Haskell Attempt at Lookup'

```
lookupBad :: ∀ (a :: Type)(m :: Nat)(n :: Nat).
             m -> m < n :~: True -> Vec a n -> a
```

```
  • Expected a type, but ‘m’ has kind ‘Nat’
  • In the type signature:
      lookupBad :: ∀ (a :: Type) (m :: Nat) (n :: Nat).
                   m -> (m < n) :~: True -> Vec a n -> a
```

```
> :k (->)
(->) :: Type -> Type -> Type
```

```
lookup' : {A : Set} → {n : ℕ} →
  (m : ℕ) → m ¡ n ≡ true → Vec A n → A
lookup' _      ()      []
lookup' zero refl (x :: _ ) = x
lookup' (suc m) p (_ :: xs) = lookup' m p xs
```

Singleton Nat

```
data SNat :: Nat -> Type where
  SZero :: SNat 'Zero
  SSucc :: ∀ (n :: Nat). SNat n -> SNat ('Succ n)
```

Haskell Lookup'

```
lookup' :: ∀ (a :: Type)(m :: Nat)(n :: Nat).
           SNat m -> m < n :~: True -> Vec a n -> a
lookup' SZero      Refl (VCons x _)  = x
lookup' (SSucc m) Refl (VCons _ xs) = lookup' m Refl xs
```

```
lookup' : {A : Set} → {n : ℕ} →
  (m : ℕ) → m ¡ n ≡ true → Vec A n → A
lookup' _     ()     []
lookup' zero refl (x :: _ ) = x
lookup' (suc m) p (_ :: xs) = lookup' m p xs
```

Another Variation

```
nth :: ∀ (a :: Type)(m :: Nat)(n :: Nat).
       (m < n) ~ 'True => SNat m -> Vec a n -> a
nth SZero     (VCons a _)  = a
nth (SSucc m) (VCons _ as) = nth m as
```

Lookup' vs Nth

```
lookup' :: ∀ (a :: Type)(m :: Nat)(n :: Nat).
           SNat m -> m < n :~: True -> Vec a n -> a
lookup' _         _    VNil         = undefined
lookup' SZero     Refl (VCons x _)  = x
lookup' (SSucc m) Refl (VCons _ xs) = lookup' m Refl xs

nth :: ∀ (a :: Type)(m :: Nat)(n :: Nat).
       (m < n) ~ 'True => SNat m -> Vec a n -> a
nth _         VNil          =  undefined
nth SZero     (VCons a _)   = a
nth (SSucc m) (VCons _ as) = nth m as
```

- ```
  Couldn't match type ''True' with ''False'
  Inaccessible code in
    a pattern with constructor: VNil :: ∀ a. Vec a 'Zero,
    in an equation for 'nth'
  ```
  ```