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?

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

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

Vector Append

$$
\begin{aligned}
&\_{+}\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathsf{zero} + n = n \\
&\mathsf{suc}\ m + n = \mathsf{suc}\ (m + n)
\end{aligned}
$$

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

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
```

Why Use Dependent Types?

- More expressive and precise
- Propositions as Types (PAT, Curry-Howard Correspondence):
  - Universal Quantification ($\forall$) corresponds to Π types.
    $\Pi_{x:A}B(x)$ or $(x : A) \rightarrow B_x$.
  - Existential Quantification ($\exists$) corresponds to Σ types.
    $\Sigma_{x:A}B(x)$ or $(x : A) \times B_x$.

Dependent Types are Not New

1971 System F
1971 Martin-Löf Type Theory
1972 LCF/ML
1978 Hindley-Milner (H 1969)
1979 Constructive math and computer programming
1982 Damas-Milner
1983 Standard ML
1984 Calculus of Constructions
1984 NuPrl
1985 Miranda
1987 Caml

1988 CiC
1989 Coq
1990 Haskell
1990 Nordström, et al, ALF
1998 Cayenne
1999 Agda 1
1991 Caml Light
1996 OCaml
2007 Agda 2
2011 Idris
2013 Homotopy Type Theory
2013 Lean
2015 Cubical Type Theory

The Golden Age is Now

- Better correctness guarantees for software.
- Mechanical verification of mathematics.
- DeepSpec, etc.
- BigProof, etc.

Robert Harper

> *Eventually all the arbitrary programming languages are going to be just swept away with the oceans, and we will have the permanence of constructive, intuitionistic 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.

Killer Apps for Typed Functional Programming

- Big Data
- Program Correctness
- Security

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

Time Line

From Richard Eisenberg's Blog:

When can we expect dependent types in GHC?

The short answer:
GHC 8.4 (2018) at the very earliest.
More likely 8.6 or 8.8 (2019-20).

Nat

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

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

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

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)
```

Plus

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

$$
\begin{aligned}
&\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathsf{zero} + n = n \\
&\mathsf{suc}\ m + n = \mathsf{suc}\ (m + n)
\end{aligned}
$$

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)
```

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

Fin

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

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

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
```

$\equiv$ 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
```

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)
```

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
```

$$
\begin{array}{l}
\mathsf{lookup'} : \{A : \mathsf{Set}\} \to \{n : \mathbb{N}\} \to \\
\quad (m : \mathbb{N}) \to m < n \equiv \mathsf{true} \to \mathsf{Vec}\ A\ n \to A \\
\mathsf{lookup'}\ \_ \quad ()\quad []\\
\mathsf{lookup'}\ \mathsf{zero}\ \mathsf{refl}\ (x :: \_) = x \\
\mathsf{lookup'}\ (\mathsf{suc}\ m)\ p\ (\_ :: xs) = \mathsf{lookup'}\ m\ p\ xs
\end{array}
$$

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’