

# Dependent Polymorphism

Makoto Hamana

Department of Computer Science,  
Gunma University, Japan

<http://www.cs.gunma-u.ac.jp/~hamana/>

# This Talk

- [I] A **semantics** for dependently-typed programming
  
- [II] A kind of polymorphism from semantics
  - **dependent polymorphism**

# This Talk

- [I] A semantics for dependently-typed programming
- [II] A kind of polymorphism from semantics
  - dependent polymorphism

Dependent typeを  
プログラミングで使える時代が  
来ました

# Dependently-typed Programming, Now!

- (i) Agda [Chalmers'07-,AIST]
- (ii) Coq with program/equations tactic [Sozeau ICFP'07,ITP'10]
- (iii) **Epigram** [McBride,McKinna '04-]
- (iv) Haskell with type classes/GADTs [McBride JFP'02, Hinze'03]

## Origin

- ▷ *Dependently Typed Functional Programs and Their Proofs*  
Conor McBride, Ph.D thesis, University of Edinburgh, 1999.

# How to Use Dependent Types in Programming?

```
data Nat : Set where
```

```
  zero : Nat
```

```
  suc   : Nat -> Nat
```

```
data Vec : Nat -> Set where      -- an inductive family
```

```
  []    : Vec zero
```

```
  _::__ : {n : Nat} -> (a : A) -> Vec n -> Vec (suc n)
```

Vec ... type of length-indexed lists

```
      []    : Vec zero
```

```
  a1 :: []   : Vec (suc zero)
```

```
a2 :: a1 :: []   : Vec (suc (suc zero))
```

# How to Use Dependent Types in Programming?

## Safe head

`head : {n : Nat} -> Vec (suc n) -> A`

`head (x :: xs) = x`

▷ Never fails

## Typical Example: append

$\_++\_ : \{m\ n : \text{Nat}\} \rightarrow \text{Vec}\ m \rightarrow \text{Vec}\ n \rightarrow \text{Vec}\ (m + n)$

$[] \quad ++\ ys = ys$

$(x :: xs) ++ ys = x :: (xs ++ ys)$

- ▷ The index of result type precisely specifies the resulting list
- ▷ Is it always possible?



## More Example: filter

### Agda

```
filter : {n : Nat} ->  
        (Nat -> Bool) -> Vec n -> Vec (?)
```

```
filter p [] = []
```

```
filter p (x :: xs) with p x
```

```
... | False = filter p xs
```

```
... | True   = x :: filter p xs
```

## More Example: filter

Agda: first attempt

```
filter : {n : Nat} ->  
        (p : Nat -> Bool) -> (xs : Vec n) -> Vec (length (filter p xs))
```

```
filter p [] = []
```

```
filter p (x :: xs) with p x
```

```
... | False = filter p xs
```

```
... | True   = x :: filter p xs
```

## More Example: filter

### Agda: correct code

```
len-filter : {n : Nat} -> (Nat -> Bool) -> Vec n -> Nat
```

```
len-filter p [] = 0
```

```
len-filter p (x :: xs) with p x
```

```
... | False = len-filter p xs
```

```
... | True   = suc (len-filter p xs)
```

```
filter : {n : Nat} ->
```

```
  (p : Nat -> Bool) -> (xs : Vec n) -> Vec (len-filter p xs)
```

```
filter p [] = []
```

```
filter p (x :: xs) with p x
```

```
... | False = filter p xs
```

```
... | True   = x :: filter p xs
```

## More Example: filter

### Agda: correct code

```
len-filter : {n : Nat} -> (Nat -> Bool) -> Vec n -> Nat
```

```
len-filter p [] = 0
```

```
len-filter p (x :: xs) with p x
```

```
... | False = len-filter p xs
```

```
... | True   = suc (len-filter p xs)
```

```
filter : {n : Nat} ->
```

```
  (p : Nat -> Bool) -> (xs : Vec n) -> Vec (len-filter p xs)
```

```
filter p [] = []
```

► Dependent polymorphism helps

```
filter p (x :: xs) with p x
```

```
... | False = filter p xs
```

```
... | True   = x :: filter p xs
```

# Classification of Polymorphism

Straychey [1967], Reynolds [1983]

Ad-hoc

$$Int \times Int \xrightarrow{+_{Int}} Int$$

Parametric

$$\begin{array}{ccc}
 Int & & Int \times Int \xrightarrow{fst_{Int}} Int \\
 \updownarrow \mathcal{R} & & \updownarrow \mathcal{R} \\
 Real & & Real \times Real \xrightarrow{fst_{Real}} Real
 \end{array}$$

Dependent

$$\begin{array}{ccc}
 Vec & & Vec \times Vec \xrightarrow{++} Vec \\
 \downarrow \text{"dependency"} & & \downarrow \\
 Nat & & Nat \times Nat \xrightarrow{+} Nat
 \end{array}$$

# This Talk

- [I] A **semantics** for dependently-typed programming
- [II] A kind of **polymorphism** from semantics

# New Semantics of Inductive Families

- 1) Simplified version of semantics of dependently-sorted abstract syntax [Fiore LICS'08]
- 2) Dependency category  $\mathcal{S}$  of sorts
- 3) The category of discourse is

**Set** <sup>$\mathcal{S}$</sup>

# Dependency Category $\mathcal{S}$

**data**  $Nat : Set$  where

$zero : Nat$

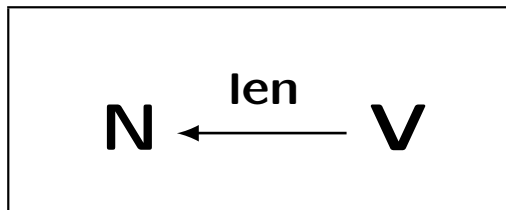
$suc : Nat \rightarrow Nat$

**data**  $Vec : Nat \rightarrow Set$  where

$nil : Vec\ Zero$

$cons : (n : Nat) \times (a : A) \times Vec\ n \rightarrow Vec\ (suc\ n)$

▷ Dependency category  $\mathcal{S}$  of sorts — skeletal, DAG



Objects: sorts

Arrows: “sort dependencies”



# Semantic Construction of Models

**data**  $Nat : Set$  where  
   $zero : Nat$   
   $suc : Nat \rightarrow Nat$

**data**  $Vec : Nat \rightarrow Set$  where  
   $nil : Vec\ Zero$   
   $cons : (n : Nat) \times (a : A) \times Vec\ n \rightarrow Vec\ (suc\ n)$

- ▷ Functor  $F : \mathbf{Set}^{\mathcal{S}} \rightarrow \mathbf{Set}^{\mathcal{S}}$  modelling an inductive family
- ▷ Initial  $F$ -algebra

# Why Interesting? – Programming Viewpoint

- ▷ The category of discourse  $\mathbf{Set}^{\mathcal{S}}$
- ▷ A natural transformation

$$f : A \rightarrow B \quad \text{in } \mathbf{Set}^{\mathcal{S}}$$

is a family of functions

$$\{f_s : A_s \rightarrow B_s \mid s \in \mathcal{S}\}$$

satisfying “naturality” ... polymorphism?

$$\begin{array}{ccc}
 \begin{array}{c} s \\ \downarrow d \\ s' \\ \text{in } \mathcal{S} \end{array} & & \begin{array}{ccc} A_s & \xrightarrow{f_s} & B_s \\ \downarrow A(d) & & \downarrow B(d) \\ A_{s'} & \xrightarrow{f_{s'}} & B_{s'} \\ \text{in } \mathbf{Set} & & \end{array}
 \end{array}$$

# Sort Dependency in Models

▷ Term model  $T \in \mathbf{Set}^{\mathcal{S}}$

$$T_{\mathbf{N}} = \{zero\} \cup \{suc(n) \mid n \in T_{\mathbf{N}}\}$$

$$T_{\mathbf{V}} = \{nil\} \cup \{cons(n, b, y) \mid n \in T_{\mathbf{N}}, b \in T_{\mathbf{B}}, y \in T_{\mathbf{V}}, \\ T(\mathit{len})(y) = n\}$$

▷ Functoriality of  $T : \mathcal{S} \rightarrow \mathbf{Set}$

$$\begin{array}{ccc}
 \mathbf{N} & \xleftarrow{\mathit{len}} & \mathbf{V} & \text{in } \mathcal{S} \\
 & & \downarrow T & \\
 T_{\mathbf{N}} & \xleftarrow{T(\mathit{len})} & T_{\mathbf{V}} & \text{in } \mathbf{Set}
 \end{array}$$

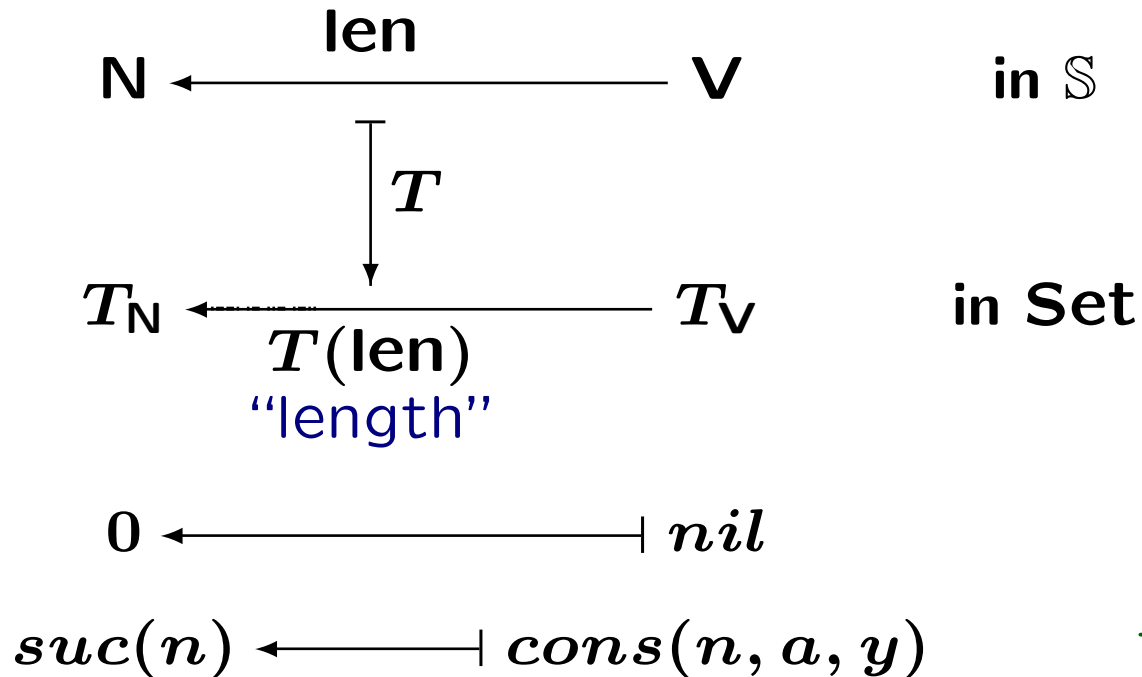
# Sort Dependency in Model

▷ Term model  $T \in \mathbf{Set}^{\mathcal{S}}$

$$T_{\mathbf{N}} = \{zero\} \cup \{suc(n) \mid n \in T_{\mathbf{N}}\}$$

$$T_{\mathbf{V}} = \{nil\} \cup \{cons(n, b, y) \mid n \in T_{\mathbf{N}}, b \in T_{\mathbf{B}}, y \in T_{\mathbf{V}}, \\ T(\mathbf{len})(y) = n\}$$

▷ Functoriality of  $T : \mathcal{S} \rightarrow \mathbf{Set}$



# Dependent Polymorphism

$$\begin{array}{ll}
 \_ \mathbin{++} \_ : \mathit{Vec}(m) \times \mathit{Vec}(n) \rightarrow \mathit{Vec}(m + n) & \_ + \_ : \mathit{Nat} \rightarrow \mathit{Nat} \\
 \mathit{nil} \mathbin{++} \mathit{ys} = \mathit{ys} & \mathit{zero} + \mathit{y} = \mathit{y} \\
 (\mathit{x} : \mathit{xs}) \mathbin{++} \mathit{ys} = \mathit{x} : (\mathit{xs} \mathbin{++} \mathit{ys}) & \mathit{suc}(n) + \mathit{y} = \mathit{suc}(n + \mathit{y})
 \end{array}$$

$$\mathbf{v} \quad T_{\mathbf{v}} \times T_{\mathbf{v}} \xrightarrow{\mathbin{++}} T_{\mathbf{v}}$$

# Dependent Polymorphism

$$_ \text{++} _ : \text{Vec}(m) \times \text{Vec}(n) \rightarrow \text{Vec}(m + n)$$

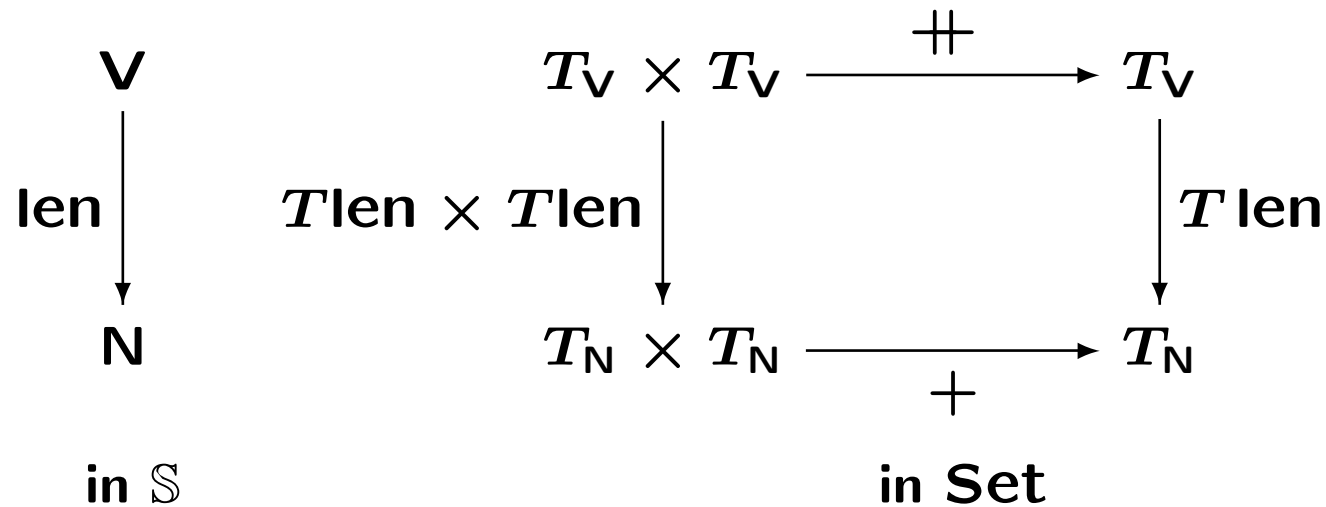
$$\text{nil} \text{++} \text{ys} = \text{ys}$$

$$(x : \text{xs}) \text{++} \text{ys} = x : (\text{xs} \text{++} \text{ys})$$

$$_ + _ : \text{Nat} \rightarrow \text{Nat}$$

$$\text{zero} + y = y$$

$$\text{suc}(n) + y = \text{suc}(n + y)$$



# Dependent Polymorphism

$$_ \ ++ \ _ : Vec(m) \times Vec(n) \rightarrow Vec(m + n)$$

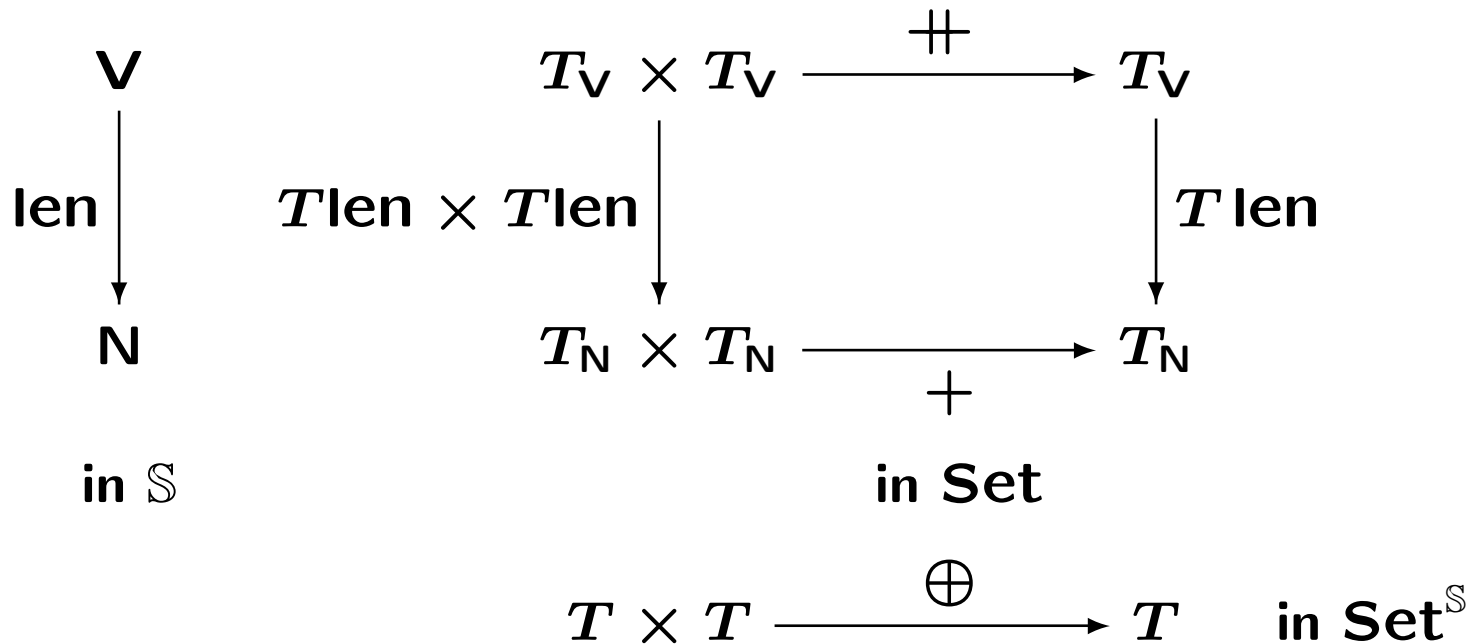
$$nil \ ++ \ ys = ys$$

$$(x :: xs) \ ++ \ ys = x :: (xs \ ++ \ ys)$$

$$_ \ + \ _ : Nat \rightarrow Nat$$

$$zero \ + \ y = y$$

$$suc(n) \ + \ y = suc(n + y)$$



Schematic definition

$$z \oplus y = y$$

$$c(n) \oplus y = c(n \oplus y)$$

... is dependently polymorphic

## Conclusion: What types admit this reading?

1. When indices are the “shapes” of data in a type
  - (i) Vectors  $a2 :: (a1 :: []) : \text{Vec} (\text{suc} (\text{suc} \text{zero}))$
  - (ii) “Shape indexed” type of trees [Hamana LMCS’10]  
e.g.  $\text{bin}(\text{lf}(3), \text{lf}(5)) : \text{Tree} (\text{B}(\text{L}, \text{L}))$
2. When indices are calculated by `fold`
  - ▶ Theoretical basis of `code reuse` in dependently-typed programming