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 : Nat} -> Vec m -> Vec n -> Vec (m + n)
[]          ++ 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 [] = []
filter p (x :: xs) with p x
... | False = filter p xs
... | True  = x :: filter p xs
```

▶ Dependent polymorphism helps

Classification of Polymorphism

Straychey [1967], Reynolds [1983]

Ad-hoc

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

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

Parametric

$$\begin{tikzcd}
Int \arrow[d, leftrightarrow, "\mathcal{R}"'] & Int \times Int \arrow[r, "\mathsf{fst}_{Int}"] \arrow[d, leftrightarrow, "\mathcal{R} \times \mathcal{R}"'] & Int \arrow[d, leftrightarrow, "\mathcal{R}"] \\
Real & Real \times Real \arrow[r, "\mathsf{fst}_{Real}"] & Real
\end{tikzcd}$$

Dependent

$$\begin{tikzcd}
Vec \arrow[d, "\text{“dependency”}"'] & Vec \times Vec \arrow[r, "+\!\!+"] \arrow[d] & Vec \arrow[d] \\
Nat & Nat \times Nat \arrow[r, "+"'] & Nat
\end{tikzcd}$$

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 $\mathbb{S}$ of sorts

3) The category of discourse is

$$\mathbf{Set}^{\mathbb{S}}$$

Dependency Category $\mathbb{S}$

$$\textbf{data}\ \ \boldsymbol{Nat} \ : \ \boldsymbol{Set}\ \ \textbf{where}$$
$$\qquad \boldsymbol{zero} \ : \ \boldsymbol{Nat}$$
$$\qquad \boldsymbol{suc} \ : \ \boldsymbol{Nat} \ \longrightarrow \ \boldsymbol{Nat}$$

$$\textbf{data}\ \ \boldsymbol{Vec} \ : \ \boldsymbol{Nat} \ \longrightarrow \ \boldsymbol{Set}\ \ \textbf{where}$$
$$\qquad \boldsymbol{nil} \ : \ \boldsymbol{Vec\ Zero}$$
$$\qquad \boldsymbol{cons} \ : \ (\boldsymbol{n} : \boldsymbol{Nat}) \ \times \ (\boldsymbol{a} : \boldsymbol{A}) \ \times \ \boldsymbol{Vec}\ \boldsymbol{n} \ \longrightarrow \ \boldsymbol{Vec}\ (\boldsymbol{suc}\ \boldsymbol{n})$$

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

$$\boxed{\mathbf{N} \xleftarrow{\ \mathbf{len}\ } \mathbf{V}}$$

Objects: sorts

Arrows : “sort dependencies”

Semantic Construction of Models

$$\mathbf{data}\ \ Nat\ :\ Set\ \ \mathbf{where}$$
$$zero\ :\ Nat$$
$$suc\ :\ Nat \longrightarrow Nat$$

$$\mathbf{data}\ \ Vec\ :\ Nat \longrightarrow Set\ \ \mathbf{where}$$
$$nil\ :\ Vec\ Zero$$
$$cons : (n : Nat) \times (a : A) \times Vec\ n \longrightarrow Vec\ (suc\ n)$$

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

Why Interesting? – Programming Viewpoint

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

$$f : A \longrightarrow B \qquad \text{in } \mathbf{Set}^{\mathbb{S}}$$

is a family of functions

$$\{f_s : A_s \longrightarrow B_s \mid s \in \mathbb{S}\}$$

satisfying "naturality" ⋯ polymorphism?

$$\begin{tikzcd} s \arrow[d, "d"'] \\ s' \end{tikzcd} \qquad \begin{tikzcd} A_s \arrow[r, "f_s"] \arrow[d, "A(d)"'] & B_s \arrow[d, "B(d)"] \\ A_{s'} \arrow[r, "f_{s'}"] & B_{s'} \end{tikzcd}$$

in $\mathbb{S}$ in $\mathbf{Set}$

Sort Dependency in Models

- Term model $T \in \mathbf{Set}^{\mathbb{S}}$

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

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

- Functoriality of $T : \mathbb{S} \longrightarrow \mathbf{Set}$

$$\begin{array}{ccccc} \mathsf{N} & \xleftarrow{\mathsf{len}} & \mathsf{V} & & \text{in } \mathbb{S} \\ & \big\downarrow T & & & \\ T_{\mathsf{N}} & \xleftarrow[T(\mathsf{len})]{} & T_{\mathsf{V}} & & \text{in } \mathbf{Set} \end{array}$$

Sort Dependency in Model

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

$$\boldsymbol{T}_{\mathsf{N}} = \{zero\} \cup \{suc(n) \mid n \in \boldsymbol{T}_{\mathsf{N}}\}$$

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

▷ Functoriality of $\boldsymbol{T} : \mathbb{S} \longrightarrow \mathbf{Set}$

$$\mathbf{N} \xleftarrow{\ \mathbf{len}\ } \mathbf{V} \qquad \text{in } \mathbb{S}$$

$$\Big\downarrow \boldsymbol{T}$$

$$\boldsymbol{T}_{\mathsf{N}} \xleftarrow[\boldsymbol{T}(\mathbf{len})]{} \boldsymbol{T}_{\mathsf{V}} \qquad \text{in } \mathbf{Set}$$

“length”

$$\mathbf{0} \longleftarrow\!\shortmid \ nil$$

$$suc(n) \longleftarrow\!\shortmid \ cons(n, a, y)$$

term level dependency

Dependent Polymorphism

$$\_ +\!\!+ \_ : Vec(m) \times Vec(n) \to Vec(m + n) \qquad \_ + \_ : Nat \to Nat$$

$$nil +\!\!+ ys = ys \qquad zero + y = y$$

$$(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) \qquad suc(n) + y = suc(n + y)$$

$$\mathsf{V} \qquad\qquad T_{\mathsf{V}} \times T_{\mathsf{V}} \xrightarrow{+\!\!+} T_{\mathsf{V}}$$

Dependent Polymorphism

$$\_ \mathbin{+\!\!+} \_ : Vec(m) \times Vec(n) \to Vec(m+n) \qquad \_ + \_ : Nat \to Nat$$

$$nil \mathbin{+\!\!+} ys = ys \qquad zero + y = y$$

$$(x : xs) \mathbin{+\!\!+} ys = x : (xs \mathbin{+\!\!+} ys) \qquad suc(n) + y = suc(n + y)$$

$$\begin{tikzcd} \mathbf{V} \arrow[d, "\mathbf{len}"'] \\ \mathbf{N} \end{tikzcd} \qquad \begin{tikzcd} T_{\mathsf{V}} \times T_{\mathsf{V}} \arrow[r, "\mathbin{+\!\!+}"] \arrow[d, "T\mathbf{len} \times T\mathbf{len}"'] & T_{\mathsf{V}} \arrow[d, "T\,\mathbf{len}"] \\ T_{\mathsf{N}} \times T_{\mathsf{N}} \arrow[r, "+"'] & T_{\mathsf{N}} \end{tikzcd}$$

in $\mathbb{S}$ **in Set**

Dependent Polymorphism

$$\_ +\!\!+ \_ : Vec(m) \times Vec(n) \to Vec(m+n) \qquad \_ + \_ : Nat \to Nat$$

$$nil +\!\!+ ys = ys \qquad zero + y = y$$

$$(x :: xs) +\!\!+ ys = x :: (xs +\!\!+ ys) \qquad suc(n) + y = suc(n+y)$$

$$\begin{array}{ccc} \mathbf{V} & & \\ \downarrow \mathsf{len} & & \\ \mathbf{N} & & \\ & & \\ \text{in } \mathbb{S} & & \end{array} \qquad \begin{array}{ccc} T_{\mathsf{V}} \times T_{\mathsf{V}} & \xrightarrow{+\!\!+} & T_{\mathsf{V}} \\ {\scriptstyle T\mathsf{len} \times T\mathsf{len}} \downarrow & & \downarrow {\scriptstyle T\,\mathsf{len}} \\ T_{\mathsf{N}} \times T_{\mathsf{N}} & \xrightarrow[+]{} & T_{\mathsf{N}} \\ & \text{in } \mathbf{Set} & \end{array}$$

$$T \times T \xrightarrow{\oplus} T \qquad \text{in } \mathbf{Set}^{\mathbb{S}}$$

Schematic definition

$$z \oplus y = y$$

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

$\cdots$ is dependently polymorphic

Conclusion: What types admit this reading?

1. When indices are the "shapes" of data in a type
   (i) Vectors a2 :: (a1 :: []) : Vec (suc (suc zero))
   (ii) "Shape indexed" type of trees [Hamana LMCS'10]
   e.g. bin(lf(3), lf(5)) : Tree (B(L,L))
2. When indices are calculated by `fold`

▶ Theoretical basis of code reuse
in dependently-typed programming