

# Parametricity for Dependent Types

Patrik Jansson

Chalmers University of Technology and University of Gothenburg



DTP, Shonan Village, 2011-09

$$\Gamma \vdash t : A \implies \llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket : \llbracket A \rrbracket \bar{t}$$

From “Theorems for Free — Parametricity for dependent types”.  
 In submission for J. of Functional Programming. Parts appeared as  
 [Bernardy, Jansson & Paterson, ParaDep, ICFP 2010].

## Parametricity: intuition, classic

Theorems for Free [Wadler, FPCA 1989] & [Reynolds, 1983]

- ▶ Type expression  $t$  using a type parameter  $a$ .
- ▶ Universal quantification gives a type (scheme)  $\sigma = \forall a. t$
- ▶ A function  $f : \sigma$  may not depend on  $a$ .
- ▶ The type imposes a contract on the function that can be expressed in logic.
- ▶ The contract (the formula  $\llbracket \sigma \rrbracket$ ) only depends on the type

### Example (filter, informal)

$filter : \forall a. (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$

$\forall a_1 : *, a_2 : *, r : a_2 \rightarrow a_1, f : a_1 \rightarrow Bool, xs : [a_2].$

$filter\ f\ (map\ r\ xs) = map\ r\ (filter\ (f \circ r)\ xs)$

# Parametricity in System F (classic)

## Definition (type to relation)

$$\begin{array}{ll}
 \llbracket (S \times T) \rrbracket & \overline{(s_i, t_i)} = \llbracket S \rrbracket \overline{s_i} \wedge \llbracket T \rrbracket \overline{t_i} \\
 \llbracket \text{List } S \rrbracket & \overline{[s_i^1, \dots, s_i^n]} = \llbracket S \rrbracket \overline{s_i^1} \wedge \dots \wedge \llbracket S \rrbracket \overline{s_i^n} \\
 \llbracket K \rrbracket & \overline{x_i} = (\equiv) \overline{x_i} \\
 \llbracket S \rightarrow T \rrbracket & \overline{f_i} = \forall \overline{s_i} : \overline{S}. \llbracket S \rrbracket \overline{s_i} \rightarrow \llbracket T \rrbracket (\overline{f_i} \overline{s_i}) \\
 \llbracket \forall s : \star. T \rrbracket & \overline{f_i} = \forall \overline{s_i} : \overline{\star}. \forall s_R : \text{Rel } \overline{s_i}. \llbracket T \rrbracket (\overline{f_i} \overline{\{s_i\}}) \\
 \llbracket s \rrbracket & \overline{x_i} = s_R \overline{x_i}
 \end{array}$$

## Theorem (Reynolds: terms, types, formulae and values)

$$\vdash t : T \Rightarrow \llbracket T \rrbracket \overline{t} \quad (1)$$

# Motivation

Why extend parametricity to dependent types?

- ▶ Originally: Testing polymorphic properties [Bernardy, Jansson, Claessen, ESOP 2010]. (Normal form involves dep. types.)
- ▶ Understand parametricity by “working with it”
- ▶ Target and source system can be the same. Neat opportunity.
- ▶ Many applications / extensions of parametricity appear. Can they be generalized?
  - ▶ Functors [Fegaras & Sheard, 1996]
  - ▶ Constructors classes [Voigtländer, 2008]
  - ▶  $R_\omega$  [Vytiniotis & Weirich, manuscript]
  - ▶ Lambda-cube [Takeuti, manuscript]

What rules should be used for  $F_\omega$  and beyond?

## Generalizing the $\Pi$ rule (towards our contribution)

( $\Pi$  stands for “dependent function space”)

$\Pi$  generalizes  $\rightarrow$  (functions) and  $\forall$  (quantification).

$$\llbracket S \rightarrow T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{S}. \llbracket S \rrbracket \bar{s}_i \rightarrow \llbracket T \rrbracket (\overline{f_i s_i})$$

$$\llbracket \forall s : \star. T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{\star}. \forall s_R : Rel \bar{s}_i. \llbracket T \rrbracket (\overline{f_i \{s_i\}})$$

$$Rel \quad \bar{t}_i = \bar{t}_i \rightarrow \star$$

$$\llbracket s \rrbracket \quad \bar{x}_i = s_R \bar{x}_i$$

# Generalizing the $\Pi$ rule (towards our contribution)

( $\Pi$  stands for “dependent function space”)

$\Pi$  generalizes  $\rightarrow$  (functions) and  $\forall$  (quantification).

$$\llbracket S \rightarrow T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{S}. \llbracket S \rrbracket \bar{s}_i \rightarrow \llbracket T \rrbracket (\overline{f_i s_i})$$

$$\llbracket \forall_- : S. T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{S}. \forall_- : \llbracket S \rrbracket \bar{s}_i. \llbracket T \rrbracket (\overline{f_i s_i})$$

$$\llbracket \forall s : \star. T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{\star}. \forall s_R : Rel \bar{s}_i. \llbracket T \rrbracket (\overline{f_i \{s_i\}})$$

$$Rel \quad \bar{t}_i = \bar{t}_i \rightarrow \star$$

$$\llbracket s \rrbracket \quad \bar{x}_i = s_R \bar{x}_i$$

# Generalizing the $\Pi$ rule (towards our contribution)

( $\Pi$  stands for “dependent function space”)

$\Pi$  generalizes  $\rightarrow$  (functions) and  $\forall$  (quantification).

$$\llbracket S \rightarrow T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{S}. \llbracket S \rrbracket \bar{s}_i \rightarrow \llbracket T \rrbracket (\bar{f}_i \bar{s}_i)$$

$$\llbracket \forall_- : S. T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{S}. \forall_- : \llbracket S \rrbracket \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i)$$

$$\llbracket \forall s : \star. T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{\star}. \forall s_R : Rel \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i)$$

$$\llbracket \forall s : \star. T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{\star}. \forall s_R : Rel \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \{s_i\})$$

$$Rel \quad \bar{t}_i = \bar{t}_i \rightarrow \star$$

$$\llbracket s \rrbracket \quad \bar{x}_i = s_R \bar{x}_i$$

# Generalizing the $\Pi$ rule (towards our contribution)

( $\Pi$  stands for “dependent function space”)

$\Pi$  generalizes  $\rightarrow$  (functions) and  $\forall$  (quantification).

$$\llbracket S \rightarrow T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{S}. \llbracket S \rrbracket \bar{s}_i \rightarrow \llbracket T \rrbracket (\bar{f}_i \bar{s}_i)$$

$$\llbracket \forall_- : S. T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{S}. \forall_- : \llbracket S \rrbracket \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i)$$

$$\llbracket \forall s : \star. T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{\star}. \forall s_R : Rel \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i)$$

$$\llbracket \forall s : \star. T \rrbracket \quad \bar{f}_i = \forall \bar{s}_i : \bar{\star}. \forall s_R : Rel \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \{s_i\})$$

$$\llbracket \star \rrbracket \quad \bar{t}_i = Rel \bar{t}_i$$

$$Rel \quad \bar{t}_i = \bar{t}_i \rightarrow \star$$

$$\llbracket s \rrbracket \quad \bar{x}_i = s_R \bar{x}_i$$

# Generalizing the $\Pi$ rule (towards our contribution)

( $\Pi$  stands for “dependent function space”)

$\Pi$  generalizes  $\rightarrow$  (functions) and  $\forall$  (quantification).

$$\begin{array}{ll} \llbracket S \rightarrow T \rrbracket & \bar{f}_i = \forall \bar{s}_i : \bar{S}. \llbracket S \rrbracket \bar{s}_i \rightarrow \llbracket T \rrbracket (\bar{f}_i \bar{s}_i) \\ \llbracket \forall_- : S. T \rrbracket & \bar{f}_i = \forall \bar{s}_i : \bar{S}. \forall_- : \llbracket S \rrbracket \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i) \\ \llbracket \forall s : S. T \rrbracket & \bar{f}_i = \forall \bar{s}_i : \bar{S}. \forall s_R : \llbracket S \rrbracket \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i) \\ \llbracket \forall s : \star. T \rrbracket & \bar{f}_i = \forall \bar{s}_i : \bar{\star}. \forall s_R : Rel \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i) \\ \llbracket \forall s : \star. T \rrbracket & \bar{f}_i = \forall \bar{s}_i : \bar{\star}. \forall s_R : Rel \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \{\bar{s}_i\}) \end{array}$$

$$\begin{array}{ll} \llbracket \star \rrbracket & \bar{t}_i = Rel \bar{t}_i \\ Rel & \bar{t}_i = \bar{t}_i \rightarrow \star \\ \llbracket s \rrbracket & \bar{x}_i = s_R \bar{x}_i \end{array}$$

# Parametricity in a pure type system

## Definition (type to relation (as type))

$$\begin{aligned} \llbracket \forall s : S. T \rrbracket &= \lambda \bar{f}_i : (\overline{\forall s : S. T}). \forall \bar{s}_i : \bar{S}. \forall s_R : \llbracket S \rrbracket \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i) \\ \llbracket \star \rrbracket &= \lambda \bar{t}_i : \bar{x}. \bar{t}_i \rightarrow \star \\ \llbracket X \rrbracket &= X_R \end{aligned}$$

We extend the translation to all (well-typed) PTS terms: adding application and abstraction.

# Parametricity in a pure type system

## Definition (type to relation (as type))

$$\llbracket \forall s : S. T \rrbracket = \lambda \bar{f}_i : (\overline{\forall s : S. T}). \forall \bar{s}_i : \bar{S}. \forall s_R : \llbracket S \rrbracket \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i)$$

$$\llbracket \star \rrbracket = \lambda \bar{t}_i : \bar{x}. \bar{t}_i \rightarrow \star$$

$$\llbracket x \rrbracket = x_R$$

$$\llbracket F a \rrbracket = \llbracket F \rrbracket \bar{a} \llbracket a \rrbracket$$

$$\llbracket \lambda x : A. b \rrbracket = \lambda \bar{x}_i : \bar{A}. \lambda x_R : \llbracket A \rrbracket \bar{x}_i. \llbracket b \rrbracket$$

We extend the translation to all (well-typed) PTS terms: adding application and abstraction.

# Parametricity in a pure type system

## Definition (type to relation (as type))

$$\begin{aligned}
 \llbracket \forall s : S. T \rrbracket &= \lambda \bar{f}_i : (\overline{\forall s : S. T}). \forall \bar{s}_i : \bar{S}. \forall s_R : \llbracket S \rrbracket \bar{s}_i. \llbracket T \rrbracket (\bar{f}_i \bar{s}_i) \\
 \llbracket \star \rrbracket &= \lambda \bar{t}_i : \bar{x}. \bar{t}_i \rightarrow \star \\
 \llbracket x \rrbracket &= x_R \\
 \llbracket F a \rrbracket &= \llbracket F \rrbracket \bar{a} \llbracket a \rrbracket \\
 \llbracket \lambda x : A. b \rrbracket &= \lambda \bar{x}_i : \bar{A}. \lambda x_R : \llbracket A \rrbracket \bar{x}_i. \llbracket b \rrbracket
 \end{aligned}$$

We extend the translation to all (well-typed) PTS terms: adding application and abstraction.

## Theorem (Revised)

$$\boxed{\Gamma \vdash t : A \implies \llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket : \llbracket A \rrbracket \bar{t}} \quad (2)$$

# Datatypes?

One can add constants  $K$ , as long as they respect:

$$\vdash K : T \Rightarrow \vdash \llbracket K \rrbracket : \llbracket T \rrbracket \bar{K}$$

We have written conforming constants for:

- ▶ Bool, Nat, List, ...
- ▶  $\Sigma$
- ▶  $W$
- ▶ Equality type

## Parametricity for the identity function

-- The type of the polymorphic identity function  
 $Id : *1$  -- Could also be called *Unit*  
 $Id = \prod * (\lambda A \rightarrow (A \rightarrow A))$

-- The type of the relational version  
 $[Id] = [\prod] [*] (\lambda [A] \rightarrow ([A] [\rightarrow] [A]))$   
 $FreeTheoremId = (id : Id) \rightarrow [Id] id id$

-- just expanding the definition  
 $= (id : Id) \rightarrow$   
 $\{A1 A2 : *\} (AR : A1 \rightarrow A2 \rightarrow *) \rightarrow$   
 $\{a1 : A1\} \{a2 : A2\} \rightarrow AR a1 a2 \rightarrow$   
 $AR (id A1 a1) (id A2 a2)$

## Example: There is just one *id*

*postulate* -- Let's assume the theorem holds

*param* : *FreeTheoremId*

*lemma* :  $\forall\{A\ B : *\}$  (*id* : *Id*) (*h* :  $A \rightarrow B$ ) *a*  $\rightarrow$

$h\ (id\ A\ a) \equiv id\ B\ (h\ a)$

*lemma id h a = param id* ( $\lambda\ a\ b \rightarrow h\ a \equiv b$ ) *refl*

-- There is just one polymorphic id.fun.

*finally* : (*id* : *Id*) (*A* :  $*$ ) (*a* : *A*)  $\rightarrow$

$a \equiv id\ A\ a$

*finally id A a = lemma id* (*const a*) *a*

# Church-encoded natural numbers

$Nat : *1$

$Nat = \Pi * \quad \lambda N \rightarrow (N \rightarrow (N \rightarrow N) \rightarrow N)$

$[Nat] = [\Pi] [*] \lambda [N] \rightarrow ([N] [\rightarrow] ([N] [\rightarrow] [N]) [\rightarrow] [N])$

$FreeTheoremNat = (n : Nat) \rightarrow [Nat] n n$

$czer : Nat$

$czer = \lambda N z s \rightarrow z$

$csuc : Nat \rightarrow Nat$

$csuc cn = \lambda N z s \rightarrow s (cn N z s)$

$cfold : (A : *) \rightarrow (z : A) \rightarrow (s : A \rightarrow A) \rightarrow Nat \rightarrow A$

$cfold A z s n = n A z s$

# Church naturals are naturals

*postulate*

*param* : FreeTheoremNat

*toN* : Nat → N

*toN* = cfold N zero succ

*toNat* : N → Nat

*toNat* zero = czer

*toNat* (succ m) = csuc (*toNat* m)

-- Poor man's extensional equality

$n1 \equiv_3 n2 = \forall A z s \rightarrow n1 A z s \equiv n2 A z s$

*theorem* :  $\forall n \rightarrow toNat (toN n) \equiv_3 n$

*theorem* n A z s =

*param* n ( $\lambda m x \rightarrow toNat m A z s \equiv x$ ) refl (cong s)

## Example: unary parametricity translation

Import the unary version of the translation (instead of the binary),  
but use the same expression:

```
[Nat] = [Π] [*] λ [N] → [N] [→] ([N] [→] [N]) [→] [N]
FreeTheoremNat1 = (n : Nat) → [Nat] n
  -- Expanded we see that it is basically the type of natrec
= (n : Nat)
  { A : Set } (P : A → Set)
  { z : A } → P z → -- Base case
  { s : A → A } → ({ m : A } → P m → P (s m)) → -- Ind step
  P (n A z s)
```

## Example: Simple datatypes (without Church)

```
data Bool : * where
```

```
  false : Bool
```

```
  true : Bool
```

```
data [Bool] : [*] Bool Bool where
```

```
  [false] : [Bool] false false
```

```
  [true] : [Bool] true true
```

## Example: Simple datatypes (without Church)

**data** *Bool* : \***where**

*false* : *Bool*

*true* : *Bool*

**data** [*Bool*] : [\*] *Bool Bool* **where**

[*false*] : [*Bool*] *false false*

[*true*] : [*Bool*] *true true*

**data** *Nat* : \***where**

*zer* : *Nat*

*suc* : *Nat* → *Nat*

**data** [*Nat*] : [\*] *Nat Nat* **where**

[*zer*] : [*Nat*] *zer zer*

[*suc*] : ([*Nat*] [→] [*Nat*]) *suc suc*

-- or, expanded

[*suc*] :  $\forall \{n\ m\} \rightarrow$  [*Nat*] *n m* → [*Nat*] (*suc n*) (*suc m*)



# Outlook

- ▶ Better understanding of parametricity
  - ▶ unifications
  - ▶ extensions
  - ▶ still simple (a few lines)
- ▶ Agda: free proofs for free theorems
- ▶ Build your own parametric rules
  - ▶ encode your system in agda
  - ▶ use the above rules
  - ▶ (eg: universe polymorphism, type-safe casts (generic programming), PolyTest, ...)

## Current status / TODOs

Done:

- ▶ Published in ICFP 2010, submitted to JFP 2011, part of JPB's PhD thesis.

TODO:

- ▶ apply results to “Testing polymorphic properties”
- ▶ Adga (or other) implementation to get a prototype tool
- ▶ explore implications to generic (polytypic) function (types)