

Generic Programming in Agda

Johan Jeuring

Utrecht University

24 February, 2014

Universes

A **universe construction** is:

- A set of types
- A datatype of codes
- A function that maps codes to types

Simple Universe

Types:

```
data  $\perp$  : Set where
```

```
data  $\top$  : Set where
```

```
  tt :  $\top$ 
```

Codes:

```
data Bool : Set where
```

```
  true  : Bool
```

```
  false : Bool
```

Decoding or interpretation function:

```
isTrue : Bool  $\rightarrow$  Set
```

```
isTrue false =  $\perp$ 
```

```
isTrue true  =  $\top$ 
```

Functions Over Decidable Propositions (1)

We can define functions on the codes.

$\neg_ : \text{Bool} \rightarrow \text{Bool}$

$\neg \text{true} = \text{false}$

$\neg \text{false} = \text{true}$

$_ \wedge _ : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$\text{true} \wedge x = x$

$\text{false} \wedge _ = \text{false}$

Functions Over Decidable Propositions (2)

We can use the functions on codes to prove properties of propositions.

```
 $\neg\neg\text{-id} : (a : \text{Bool}) \rightarrow \text{isTrue } (\neg \neg a) \rightarrow \text{isTrue } a$ 
```

```
 $\neg\neg\text{-id true } p = p$ 
```

```
 $\neg\neg\text{-id false } ()$ 
```

```
 $\wedge\text{-intro} : (a b : \text{Bool}) \rightarrow \text{isTrue } a \rightarrow \text{isTrue } b \rightarrow \text{isTrue } (a \wedge b)$ 
```

```
 $\wedge\text{-intro true } \_ \_ p = p$ 
```

```
 $\wedge\text{-intro false } \_ () \_$ 
```

Universes in Generic Programming

- We have seen universes in Haskell for each of the GP libraries covered in this course.
- Now, we can define libraries in Agda and clearly distinguish the codes from the types and the interpretation of the codes.
- We can formally compare libraries using the universes in Agda.

Types

First, let's define the remaining types of the universe:

- Disjoint union (“sum”):

```
data _ $\uplus$ _ (A B : Set) : Set where  
  inj1 : A → A  $\uplus$  B  
  inj2 : B → A  $\uplus$  B
```

- Cartesian product:

```
data _ $\times$ _ (A B : Set) : Set where  
  _,_ : A → B → A  $\times$  B
```

LIGD-like Universe

The codes for a sums-of-products representation (like LIGD):

```
data Sop : Set where  
  unit   : Sop  
  nat    : Sop  
  sum    : (R S : Sop) → Sop  
  prod   : (R S : Sop) → Sop
```

The interpretation:

```
 $\llbracket \_ \rrbracket_S : \text{Sop} \rightarrow \text{Set}$   
 $\llbracket \text{unit} \rrbracket_S = \top$   
 $\llbracket \text{nat} \rrbracket_S = \mathbb{N}$   
 $\llbracket \text{sum } r \ s \rrbracket_S = \llbracket r \rrbracket_S \uplus \llbracket s \rrbracket_S$   
 $\llbracket \text{prod } r \ s \rrbracket_S = \llbracket r \rrbracket_S \times \llbracket s \rrbracket_S$ 
```


Generic Equality with Sop

Using the Sop universe construction, we can define generic functions:

```

$$\begin{aligned} \_ == \_ & : \{A : \text{Sop}\} \rightarrow \llbracket A \rrbracket_S \rightarrow \llbracket A \rrbracket_S \rightarrow \text{Bool} \\ \_ == \_ \{ \text{unit} \} & \quad \text{tt} \quad \text{tt} \quad = \text{true} \\ \_ == \_ \{ \text{nat} \} & \quad \text{zero} \quad \text{zero} \quad = \text{true} \\ \_ == \_ \{ \text{nat} \} & \quad (\text{succ } m) \quad (\text{succ } n) \quad = m == n \\ \_ == \_ \{ \text{sum } r \ s \} & \quad (\text{inj}_1 \ x) \quad (\text{inj}_1 \ y) \quad = x == y \\ \_ == \_ \{ \text{sum } r \ s \} & \quad (\text{inj}_2 \ x) \quad (\text{inj}_2 \ y) \quad = x == y \\ \_ == \_ \{ \text{prod } r \ s \} & \quad (x_1, y_1) \quad (x_2, y_2) \quad = x_1 == x_2 \wedge y_1 == y_2 \\ \_ == \_ \{ \_ \} & \quad \_ \quad \_ \quad = \text{false} \end{aligned}$$

```

Note that, unlike LIGD in Haskell, the codes here are implicit.

Regular Universe (1)

We can define the Regular library in a similar way. The codes:

```
data Regular : Set where  
  U      : Regular  
  K      : (A : Set) → Regular  
  _⊕_    : (F G : Regular) → Regular  
  _⊗_    : (F G : Regular) → Regular  
  I      : Regular
```

Note that `Regular` is similar to `Sop`, but:

- `K` is for all constant types (not just `ℕ`).
- `I` is for recursive positions.

Regular Universe (2)

The interpretation:

$$\begin{aligned} \llbracket _ \rrbracket_R &: \text{Regular} \rightarrow \text{Set} \rightarrow \text{Set} \\ \llbracket \mathbf{U} \rrbracket_R r &= \top \\ \llbracket \mathbf{K} a \rrbracket_R r &= a \\ \llbracket \mathbf{F} \oplus \mathbf{G} \rrbracket_R r &= \llbracket \mathbf{F} \rrbracket_R r \uplus \llbracket \mathbf{G} \rrbracket_R r \\ \llbracket \mathbf{F} \otimes \mathbf{G} \rrbracket_R r &= \llbracket \mathbf{F} \rrbracket_R r \times \llbracket \mathbf{G} \rrbracket_R r \\ \llbracket \mathbf{I} \rrbracket_R r &= r \end{aligned}$$

- The interpretation is parameterized by the recursive type.
- The fixed point (i.e. `Fix`):

```
data  $\mu$  (F : Regular) : Set where  
  <_> :  $\llbracket \mathbf{F} \rrbracket_R (\mu \mathbf{F}) \rightarrow \mu \mathbf{F}$ 
```

map for Regular

Since we have a functor representation, we can naturally define `map`.

$$\begin{aligned} \text{map} &: \forall \{A B\} F \rightarrow (A \rightarrow B) \rightarrow \llbracket F \rrbracket_R A \rightarrow \llbracket F \rrbracket_R B \\ \text{map } U & \quad _ \text{tt} \quad = \text{tt} \\ \text{map } (K _) & \quad _ x \quad = x \\ \text{map } (F \oplus G) f & \text{ (inj}_1 x) = \text{inj}_1 (\text{map } F f x) \\ \text{map } (F \oplus G) f & \text{ (inj}_2 x) = \text{inj}_2 (\text{map } G f x) \\ \text{map } (F \otimes G) f & (x, y) = (\text{map } F f x, \text{map } G f y) \\ \text{map } I & \quad f x \quad = f x \end{aligned}$$

fold for Regular (1)

This is a first attempt to define `fold`.

```
fold! : ∀ {F A} → ([[ F ]]_R A → A) → μ F → A
fold! {F} alg ⟨ x ⟩ = alg (map F (fold! {F} alg) x)
```

- The termination checker cannot prove that this function terminates.
- Agda expects a structurally smaller value for recursion.
- `x` is passed to a higher-order function.
- The termination checker cannot see through such functions.

fold for Regular (2)

Solution: Fuse `map` and `fold` into a single function such that:

$$\text{mapFold } F \ G \ \text{alg } x = \text{map } F \ (\text{fold } G \ \text{alg}) \ x$$

$$\begin{aligned} \text{mapFold} & : \forall \{A\} \ F \ G \rightarrow (\llbracket G \rrbracket_R A \rightarrow A) \rightarrow \llbracket F \rrbracket_R (\mu G) \rightarrow \llbracket F \rrbracket_R A \\ \text{mapFold } U & \quad _ _ \ \text{tt} \quad = \text{tt} \\ \text{mapFold } (K _) & \quad _ _ \ c \quad = c \\ \text{mapFold } (F_1 \oplus F_2) \ G \ \text{alg} \ (\text{inj}_1 \ x) & = \text{inj}_1 \ (\text{mapFold } F_1 \ G \ \text{alg} \ x) \\ \text{mapFold } (F_1 \oplus F_2) \ G \ \text{alg} \ (\text{inj}_2 \ y) & = \text{inj}_2 \ (\text{mapFold } F_2 \ G \ \text{alg} \ y) \\ \text{mapFold } (F_1 \otimes F_2) \ G \ \text{alg} \ (x, y) & = \text{mapFold } F_1 \ G \ \text{alg} \ x, \text{mapFold } F_2 \ G \ \text{alg} \ y \\ \text{mapFold } I & \quad G \ \text{alg} \ \langle x \rangle \quad = \text{alg} \ (\text{mapFold } G \ G \ \text{alg} \ x) \end{aligned}$$

A `fold` that passes the termination checker is now straightforward:

$$\begin{aligned} \text{fold} & : \forall \{F \ A\} \rightarrow (\llbracket F \rrbracket_R A \rightarrow A) \rightarrow \mu F \rightarrow A \\ \text{fold } \{F\} \ \text{alg} \ \langle x \rangle & = \text{alg} \ (\text{mapFold } F \ F \ \text{alg} \ x) \end{aligned}$$

Conversion Between Codes

We can define a conversion from `Sop` to `Regular`.

$$\begin{aligned} S\uparrow R &: \text{Sop} \rightarrow \text{Regular} \\ S\uparrow R \text{ unit} &= U \\ S\uparrow R \text{ nat} &= K \mathbb{N} \\ S\uparrow R (\text{sum } r \text{ } s) &= S\uparrow R r \oplus S\uparrow R s \\ S\uparrow R (\text{prod } r \text{ } s) &= S\uparrow R r \otimes S\uparrow R s \end{aligned}$$

- `Sop` does not support recursion.
- `Sop` has a strictly smaller universe than `Regular`.

Conversion Between Interpretations

With some help from a congruence rule that allows us to test equality on a binary function...

```
cong2 : ∀ {A B C x x' y y'}  
         → (f : A → B → C) → x ≡ x' → y ≡ y' → f x y ≡ f x' y'  
cong2 _ refl refl = refl
```

We can prove that the two interpretations of a `Sop` code are equivalent.

```
intr : ∀ {r} (S : Sop) → [[ S ]]S ≡ [[ S↑R S ]]R r  
intr unit      = refl  
intr nat       = refl  
intr (sum r s) = cong2 _ ⊕ _ (intr r) (intr s)  
intr (prod r s) = cong2 _ × _ (intr r) (intr s)
```