Generic Programming in Agda

Johan Jeuring

Utrecht University

11 June, 2015

Universes

A universe construction is:

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

Simple Universe

Types:

data $\bot$: Set **where**

data $\top$: Set **where**
  tt : $\top$

Codes:

data Bool : Set **where**
  true : Bool
  false : Bool

Decoding or interpretation function:

isTrue : Bool $\to$ Set
isTrue false $=$ $\bot$
isTrue true $=$ $\top$

Functions Over Decidable Propositions (1)

We can define functions on the codes.

```
¬_ : Bool → Bool
¬ true  = false
¬ false = true
```

```
_∧_ : Bool → Bool → Bool
true ∧ x  = x
false ∧ _ = false
```

Functions Over Decidable Propositions (2)

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

```
¬¬-id : (a : Bool) → isTrue (¬ ¬ a) → isTrue a
¬¬-id true p = p
¬¬-id false ()
```

```
∧-intro : (a b : Bool) → isTrue a → isTrue b → isTrue (a ∧ b)
∧-intro true _ _ p = p
∧-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, lets define the remaining types of the universe:

- Disjoint union ("sum"):

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

- Cartesian product:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × 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:

$$
\begin{aligned}
&[\![\_]\!]_S : \mathsf{Sop} \to \mathsf{Set}\\
&[\![\ \mathsf{unit}\ ]\!]_S = \top\\
&[\![\ \mathsf{nat}\ ]\!]_S = \mathbb{N}\\
&[\![\ \mathsf{sum}\ r\ s\ ]\!]_S = [\![\ r\ ]\!]_S \uplus [\![\ s\ ]\!]_S\\
&[\![\ \mathsf{prod}\ r\ s\ ]\!]_S = [\![\ r\ ]\!]_S \times [\![\ s\ ]\!]_S
\end{aligned}
$$

Generic Equality with Sop

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

$$
\begin{array}{llllll}
\_{==}\_ & : \{A : \mathsf{Sop}\} \to [\![A]\!]_S \to [\![A]\!]_S \to \mathsf{Bool} \\
\_{==}\_ & \{\mathsf{unit}\} & \mathsf{tt} & \mathsf{tt} & = & \mathsf{true} \\
\_{==}\_ & \{\mathsf{nat}\} & \mathsf{zero} & \mathsf{zero} & = & \mathsf{true} \\
\_{==}\_ & \{\mathsf{nat}\} & (\mathsf{succ}\ m) & (\mathsf{succ}\ n) & = & m == n \\
\_{==}\_ & \{\mathsf{sum}\ r\ s\} & (\mathsf{inj}_1\ x) & (\mathsf{inj}_1\ y) & = & x == y \\
\_{==}\_ & \{\mathsf{sum}\ r\ s\} & (\mathsf{inj}_2\ x) & (\mathsf{inj}_2\ y) & = & x == y \\
\_{==}\_ & \{\mathsf{prod}\ r\ s\} & (x_1, y_1) & (x_2, y_2) & = & x_1 == x_2 \wedge y_1 == y_2 \\
\_{==}\_ & \{\_\} & \_ & \_ & = & \mathsf{false}
\end{array}
$$

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:

$$
\begin{array}{l}
[\![\_]\!]_R : \text{Regular} \to \text{Set} \to \text{Set} \\
[\![\ U\ ]\!]_R\ r = \top \\
[\![\ K\ a\ ]\!]_R\ r = a \\
[\![\ F \oplus G\ ]\!]_R\ r = [\![\ F\ ]\!]_R\ r \uplus [\![\ G\ ]\!]_R\ r \\
[\![\ F \otimes G\ ]\!]_R\ r = [\![\ F\ ]\!]_R\ r \times [\![\ G\ ]\!]_R\ r \\
[\![\ I\ ]\!]_R\ r = r
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{data}\ \mu\ (F : \text{Regular}) : \text{Set}\ \textbf{where} \\
\quad \langle\_\rangle : [\![\ F\ ]\!]_R\ (\mu\ F) \to \mu\ F
\end{array}
$$

map for Regular

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

$$
\begin{array}{llll}
\mathsf{map} : \forall \{A\ B\}\ F \to (A \to B) \to [\![F]\!]_R\ A \to [\![F]\!]_R\ B \\
\mathsf{map}\ U & \_\ tt & = & tt \\
\mathsf{map}\ (K\ \_) & \_\ x & = & x \\
\mathsf{map}\ (F \oplus G)\ f & (inj_1\ x) & = & inj_1\ (\mathsf{map}\ F\ f\ x) \\
\mathsf{map}\ (F \oplus G)\ f & (inj_2\ x) & = & inj_2\ (\mathsf{map}\ G\ f\ x) \\
\mathsf{map}\ (F \otimes G)\ f & (x, y) & = & (\mathsf{map}\ F\ f\ x, \mathsf{map}\ G\ f\ y) \\
\mathsf{map}\ I & f\ x & = & f\ x
\end{array}
$$

fold for Regular (1)

This is a first attempt to define fold.

$$\text{fold!} : \forall \{F\ A\} \rightarrow (\llbracket F \rrbracket_R\ A \rightarrow A) \rightarrow \mu\ F \rightarrow A$$
$$\text{fold!}\ \{F\}\ \text{alg}\ \langle\, x \,\rangle = \text{alg}\ (\text{map}\ F\ (\text{fold!}\ \{F\}\ \text{alg})\ x)$$

- 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:

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

$$\begin{array}{llll}
\mathsf{mapFold} & : \forall\,\{A\}\ F\ G\ \rightarrow\ (\llbracket\ G\ \rrbracket_R\ A\ \rightarrow\ A)\ \rightarrow\ \llbracket\ F\ \rrbracket_R\ (\mu\ G)\ \rightarrow\ \llbracket\ F\ \rrbracket_R\ A & & \\
\mathsf{mapFold}\ U & _\ _\ \ tt & = & tt \\
\mathsf{mapFold}\ (K\ _) & _\ _\ \ c & = & c \\
\mathsf{mapFold}\ (F_1 \oplus F_2) & G\ alg\ (inj_1\ x) & = & inj_1\ (\mathsf{mapFold}\ F_1\ G\ alg\ x) \\
\mathsf{mapFold}\ (F_1 \oplus F_2) & G\ alg\ (inj_2\ y) & = & inj_2\ (\mathsf{mapFold}\ F_2\ G\ alg\ y) \\
\mathsf{mapFold}\ (F_1 \otimes F_2) & G\ alg\ (x, y) & = & \mathsf{mapFold}\ F_1\ G\ alg\ x, \mathsf{mapFold}\ F_2\ G\ alg\ y \\
\mathsf{mapFold}\ I & G\ alg\ \langle\ x\ \rangle & = & alg\ (\mathsf{mapFold}\ G\ G\ alg\ x)
\end{array}$$

A fold that passes the termination checker is now straightforward:

$$\begin{array}{l}
\mathsf{fold}\ :\ \forall\,\{F\ A\}\ \rightarrow\ (\llbracket\ F\ \rrbracket_R\ A\ \rightarrow\ A)\ \rightarrow\ \mu\ F\ \rightarrow\ A \\
\mathsf{fold}\ \{F\}\ alg\ \langle\ x\ \rangle\ =\ alg\ (\mathsf{mapFold}\ F\ F\ alg\ x)
\end{array}$$

Conversion Between Codes

We can define a conversion from Sop to Regular.

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

- 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...

$$
\begin{aligned}
&\mathsf{cong}_2\ :\ \forall\,\{A\ B\ C\ x\ x'\ y\ y'\} \\
&\qquad \to\ (f\ :\ A\ \to\ B\ \to\ C)\ \to\ x \equiv x'\ \to\ y \equiv y'\ \to\ f\,x\,y \equiv f\,x'\,y' \\
&\mathsf{cong}_2\ \_\ \mathsf{refl}\ \mathsf{refl}\ =\ \mathsf{refl}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathsf{intr}\ :\ \forall\,\{r\}\ (S\ :\ \mathsf{Sop})\ \to\ [\![\ S\ ]\!]_S \equiv [\![\ S{\uparrow}R\ S\ ]\!]_R\ r \\
&\mathsf{intr\ unit} \qquad\quad =\ \mathsf{refl} \\
&\mathsf{intr\ nat} \qquad\quad\ =\ \mathsf{refl} \\
&\mathsf{intr}\ (\mathsf{sum}\ r\ s)\ \ =\ \mathsf{cong}_2\ \_\uplus\_\ (\mathsf{intr}\ r)\ (\mathsf{intr}\ s) \\
&\mathsf{intr}\ (\mathsf{prod}\ r\ s)\ =\ \mathsf{cong}_2\ \_\times\_\ (\mathsf{intr}\ r)\ (\mathsf{intr}\ s)
\end{aligned}
$$