Cubical Agda features

Guilherme Silva

11th September 2021

New Features

The most important features in Cubical Agda are:

- New Equality Type
- Quotient types

Equalities

There are multiple definitions of equalities and the most important are:

- Martin-Löf
- Leibniz
- Cubical

Imports

```
{-# OPTIONS --cubical #-}

module slides where

open import Agda.Primitive.Cubical

open import Cubical.Data.Unit
open import Cubical.Data.Bool
open import Cubical.Data.Int
open import Cubical.Data.Nat hiding (_·_)
open import Cubical.Data.Prod
open import Cubical.Foundations.Isomorphism
```

Martin-Löf equality

At the begin of Agda and in most theorems proves, equality is given by Martin-Löf's definition:

```
module Martin-Löf {ℓ} {A : Set ℓ} where

  data _≡_ (x : A) : A → Set ℓ where
    refl : x ≡ x
```

Martin-Löf pattern match

This equality is very convenient in proof assistances like Agda because it is possible to pattern match using them:

```
private variable
  x y z : A

sym : x ≡ y → y ≡ x
sym refl = refl

trans : x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

Martin-Löf problem

But the problem of this equality is that it does not handle extensionality and other axioms very well:

```
module FunExt {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} where
  open Martin-Löf

  funExt-Type = {f g : A → B}
    → ((x : A) → f x ≡ g x) → f ≡ g
```

Cubical Equality

To solve this problem, Agda adopted cubical type theory that equality is a function from the path to type:

```
module CubicalEquality {ℓ} {A : Set ℓ} where
  postulate
    PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ

  _≡_ : A → A → Set ℓ
  _≡_ = PathP λ _ → A
```

Cubical Equality Operators

From this equality, I will define reflection, symmetry and extensionality:

```
module CubicalResults {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} where
  open import Cubical.Core.Primitives

  private variable
    x y z : A

  refl : x ≡ x
  refl {x = x} = λ _ → x

  sym : x ≡ y → y ≡ x
  sym p i = p (~ i)

  funExt : {f g : A → B}
    → ((x : A) → f x ≡ g x) → f ≡ g
  funExt p i x = p x i
```

The operator $\sim$ invert the interval. If the interval i goes from $i0$ to $i1$, the interval $\sim i$ goes from $i1$ to $i0$.

Cubical equality computational behavior

Another advantage of cubical equality is that it has a computational behavior. If there is a term that has a type that two types are equal, in this term, there is information of how they are equal. So this term represents the isomorphism of these two types.

Bool example

In this example, I will show the equality of two Bools:

```
module CubicalIsomorphism where
  open import Cubical.Foundations.Prelude

  IsoBool : Iso Bool Bool
  IsoBool = iso not not ¬¬b ¬¬b
    where
      ¬¬b : ∀ b → not (not b) ≡ b
      ¬¬b false = refl
      ¬¬b true = refl

  Bool≡Bool : Bool ≡ Bool
  Bool≡Bool = isoToPath IsoBool

  _ : (transport Bool≡Bool false ≡ true)
    × (transport Bool≡Bool true ≡ false)
  _ = refl , refl
```

Quotient Types' Motivation

In simple type theories with quotient types, it is possible to create two distinct elements and after make them equal.
One of the best examples is making $\frac{1}{2}$ and $\frac{2}{4}$ be equal elements.

Imports

```
open import Cubical.Foundations.Prelude hiding (isProp; isSet)
open import Cubical.Relation.Nullary using (¬_)

private variable
  ℓ ℓ₁ : Level
  A : Set ℓ
  B : Set ℓ₁
  x y z : A
```

The simplest example of quotient types

This is an example when two elements of the same data types are equal:

```
data Bool≡ : Set where
  true false : Bool≡
  t≡f : true ≡ false

_ : true ≡ false
_ = t≡f
```

Functions of quotient types

Let a function f between elements of Bool≡ and another set, if $x \equiv y$, then $f(x) \equiv f(y)$. In this case, x is true, y is false and $f(x)$ is tt.

```
f : Bool≡ → Unit
f true  = tt
f false = tt
f (t≡f i) = refl i -- proving that f true ≡ f false
```

Truncation type

In truncation type, every element of it is equal. Therefore we can define Bool≡ as ||| Bool |||:

```
data |||_||| {ℓ} (A : Set ℓ) : Set ℓ where
  ||_|| : A → ||| A |||
  squash : ∀ (x y : ||| A |||) → x ≡ y

Bool≡' = ||| Bool |||
```

Not all equalities are the same

In cubical type theory, the equalities are not always the same. In this example, the circle is isomorphic (the same) to the integers:

```
data Circle : Type where
  base : Circle
  loop : base ≡ base
```

Making equalities equal

To make the equality equal, it is necessary when declaring a data type to state that it is a set:

$$\begin{aligned}&\mathsf{isProp} : \mathsf{Type}\ \ell \to \mathsf{Type}\ \ell\\&\mathsf{isProp}\ A = (x\ y : A) \to x \equiv y\end{aligned}$$

$$\begin{aligned}&\mathsf{isSet} : \mathsf{Type}\ \ell \to \mathsf{Type}\ \ell\\&\mathsf{isSet}\ A = (x\ y : A) \to \mathsf{isProp}\ (x \equiv y)\end{aligned}$$

Rational Numbers

Rational numbers are defined by numerator and a denominator different from zero. Two rational numbers $\frac{p_1}{q_1}$ and $\frac{p_2}{q_2}$ are equal when $p_1 \times q_2 \equiv p_2 \times q_1$:

```
data ℚ : Type where
  _/_[_] : (p : ℤ) (q : ℤ) → ¬ (q ≡ pos 0) → ℚ
  path : ∀ p 1 q 1 p 2 q 2 {pr 1 pr 2} → (p 1 · q 2) ≡ (p 2 · q 1)
    → p 1 / q 1 [ pr 1 ] ≡ p 2 / q 2 [ pr 2 ]
  trunc : isSet ℚ

_ : ∀ {pr 1 : ¬ (2 ≡ pos 0)} {pr 2 : ¬ (4 ≡ pos 0)}
  → 1 / 2 [ pr 1 ] ≡ 2 / 4 [ pr 2 ]
_ = path 1 2 2 4 refl
```

Integers

Another way to define integers is making the positive and negative zero equals:

```
data ℤ' : Type where
  pos' : ℕ → ℤ'
  neg' : ℕ → ℤ'
  0+-≡ : pos' 0 ≡ neg' 0
```

Sets

One way of defining sets is to create a list where you can swap elements and remove them when they are equal:

```
infixr 20 _::_
data LFSet (A : Type ℓ) : Type ℓ where
  []      : LFSet A
  _::_    : (x : A) → (xs : LFSet A) → LFSet A
  dup     : ∀ x xs → x :: x :: xs ≡ x :: xs
  comm    : ∀ x y xs → x :: y :: xs ≡ y :: x :: xs
  trunc   : isSet (LFSet A)

_ : tt :: tt :: [] ≡ tt :: []
_ = dup tt []

_ : let true::false : LFSet Bool
        true::false = true :: false :: []
    in true::false ≡ false :: true :: []
_ = comm true false []
```

Conclusion

- Each equality has some advantage. So it is possible to choose which one is more convenient for the proof.
- Proofs using quotient types can be harder to solve. But sometimes, the definition of the data type using them can be easier to understand. So a code that is easier to understand is less likely of having a bug.