Agda: Equality

Andreas Abel

2 July 2012

1 Definitional Equality

Agda has an internal notion of program equality. In essence, two programs are equal if they compute the same value. For instance, (λ x → x + y) 5 and 5 + y are equal since the second arises if you compute the value of the first expression. When you define a function

```
open import Data.Bool hiding (_∨_)
_∨_ : Bool → Bool → Bool
true  ∨ y = true
false ∨ y = y
```

then you add the defining equations to Agda's internal equality. This internal equality is usually called *definitional equality*.

But careful, not every equation you write holds literally in Agda! Consider the following, alternative definition of disjunction:

```
_∨_ : Bool → Bool → Bool
false ∨ false = false
x     ∨ y     = true
```

Only the first equation holds as such, and this is actually good, otherwise we would also have false ∨ false = true as instance of the second equation. In this case, the second equation does not hold because it overlaps with the first equation. Internally, it is expanded into three equations, which hold definitionally.

```
false ∨ true  = true
true  ∨ false = true
true  ∨ true  = true
```

Consider yet another version of disjunction:

```
_∨_ : Bool → Bool → Bool
false ∨ false = false
```

```
true ∨ false = true
x    ∨ true  = true
```

These clauses are not overlapping, yet still the third equation does not hold definitionally. We can test it by normalizing the following expression:

```
test∨ : Bool → Bool
test∨ = λ x → x ∨ true
```

If type `C-c C-n test`` we do not get λ x → true as we might expect, but λ x → x ∨ true. Internally, Agda has split the third clause x ∨ true = true into the two clauses:

```
false ∨ true = true
true  ∨ true = true
```

Because the first clause did not have a variable as first argument but the constructor false, Agda splits the first argument of all clauses into the two cases false and true. To get the correct behavior, we need to change the order of the clauses.

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

Now test∨ evaluates to λ x → true.

2 Propositional Equality

If we want to *prove* that two programs are equal, we cannot directly use the definitional equality. A proof is itself a program with a type, and in this the type should express that two things are equal. This type can be defined in Agda; to use it, import Relation.Binary.PropositionalEquality. Propositional equality as defined in the standard library is universe polymorphic; we start with a simpler version.

```
module Level0Equality (A : Set) where
  data _≡_ : A → A → Set where
    refl : (a : A) → a ≡ a
```

We are defining _≡_, the least binary relation on A such that a ≡ a for all a : A.

If we have two *definitionally* equal terms a and b, then refl a and refl b are both proofs of *(propositional)* equality of a and b. This is because internally, Agda does not distinguish between definitionally equal terms.

Propositional equality is an *equivalence relation*, meaning that it is reflexive (by definition), symmetric (sym), and transitive (trans). Symmetry and transitivity can be proven by pattern matching.

```
sym : ∀ x y → x ≡ y → y ≡ x
sym .a .a (refl a) = refl a
```

We match on x ≡ y, and since refl is the only constructor of this data type, refl a for some variable a is only matching pattern. By this in turn forces x ≡ y to be definitionally equal to a ≡ a meaning that both x and y must be definitionally equal to a. Such a forced coincidence is expressed in Agda via a *dot pattern* (aka *inaccessible* pattern). If instead of a pattern, we write .expression, we mean that during the match, the value at this position is forced to be expression, and an actual match is not necessary.

Transitivity is likewise easy, using dependent pattern matching:

```
trans : ∀ x y z → x ≡ y → y ≡ z → x ≡ z
trans .a .a .a (refl .a) (refl a) = refl a
```

A central concept of equality is *substitutivity*: in any proposition (type), we can replace a term with a propositionally equal one, without changing the meaning of the proposition (type).

```
subst : ∀ x y → x ≡ y → ∀ (P : A → Set) → P x → P y
subst .a .a (refl a) P Pa = Pa
```

If P x holds and x ≡ y then P y also holds. Leibniz took substitutivity as the *definition* of equality: If two objects x and y are indistinguishable by any observation P, then they are equal. Thus, x ≡ y = ∀ (P : A → Set) → P x → P y would be an alternative definition of propositional equality. It is equivalent to the inductive definition, since we can prove the opposite of subst:

```
leibniz : ∀ x y → (∀ (P : A → Set) → P x → P y) → x ≡ y
leibniz x y H = H (_≡_ x) (refl x)
```

The proof H that x and y are Leibniz-equal is instantiated to the predicate P a = (x ≡ a). Then we supply a proof refl x of P x and obtain the desired proof of P y = (x ≡ y).

3 The Standard Definition of Propositional Equality

If we import Relation.Binary.PropositionalEquality, we get the definition and standard tools to work with equality.

```
open import Relation.Binary.PropositionalEquality
```

The definition in the standard library is universe polymorphic. It is hidden in a .Core module—such modules do not need to be imported explicitly.

```
module Relation.Binary.Core where
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

Further, refl does not require an argument.

The properties of equality are specified with more hidden arguments than we did in the last section.

```
module Relation.Binary.PropositionalEquality.Core where
sym : ∀ {a} {A : Set a} → Symmetric (_≡_ {A = A})
sym refl = refl
trans : ∀ {a} {A : Set a} → Transitive (_≡_ {A = A})
trans refl eq = eq
subst : ∀ {a p} {A : Set a} → Substitutive (_≡_ {A = A}) p
subst P refl p = p
```

Note the different argument order in subst.

An important consequence of substitutivity is *congruence*.

```
module Relation.Binary.PropositionalEquality where
cong : ∀ {a b} {A : Set a} {B : Set b}
       (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

It says that any function f respects propositional equality, i.e., yields propositionally equal results if applied to propositionally equal arguments. We use cong if we want to apply equality in a subterm.

4 An Example

Let us prove the associativity of list concatenation!

```
open import Data.List


++-assoc : ∀ {a} {A : Set a} (xs ys zs : List A) →
           (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
```

We start with the following universal template that we load into Agda (`C-c C-l`):

```
++-assoc xs ys zs = ?
```

What is the best way to proceed? Append _++_ is defined by cases on the first argument:

```
_++_ : ∀ {a} {A : Set a} → List A → List A → List A
[]        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

A good proof strategy is to case on the same argument as the involved recursive function does, because then definitional equality triggers simplifications. So, let us case (`C-c C-c`) on variable xs.

```
++-assoc []       ys zs = { } 0
++-assoc (x :: xs) ys zs = { } 1
```

The first goal ?0 : ([] ++ ys) ++ zs ≡ [] ++ ys ++ zs is definitionally equal to ys ++ zs ≡ ys ++ zs; this can be seen by pressing `C-C C-,` in hole 0. The proof is simply refl. The second goal simplifies to x :: ((xs ++ ys) ++ zs) ≡ x :: (xs ++ (ys ++ zs)) which we can prove by applying the induction hypothesis in subterm l of x :: l.

```
++-assoc []       ys zs = refl
++-assoc (x :: xs) ys zs = cong (λ l → x :: l) (++-assoc xs ys zs)
```

The induction hypothesis is just a recursive call. Since the first argument is decreasing, the termination checker accepts this call. In essence, we have proven the associativity of append by induction on the first argument.

Admittedly, this proof is not very readable. The proposition we are currently manipulating is not visible. One has to have trained Agda-eyes to recognize the argument. We can make the proof more verbose by using the equation chains of the standard library.

5 Equation Chains

Relation.Binary.PropositionalEquality provides a module ≡-Reasoning that provides nice mixfix syntax for writing down a chain of equalities:

```
open ≡-Reasoning
++-assoc : ∀ {a} {A : Set a} (xs ys zs : List A) →
           (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc [] ys zs = begin
    ([] ++ ys) ++ zs
  ≡⟨ refl ⟩
    ys ++ zs
  ≡⟨ refl ⟩
    [] ++ (ys ++ zs)
  ∎
```

We start our chain with begin and end it with ∎ (enter `\qed`). Each equality sign is decorated with a justification (proof term) of that equation; in this case all proofs are just refl (enter `\equiv\< refl \>`, because all these equations hold by definition.

In the step case of our induction, we make also each single transformation step obvious.

```
++-assoc (x :: xs) ys zs = begin
    ((x :: xs) ++ ys) ++ zs
```

```
≡⟨ refl ⟩
  (x :: (xs ++ ys)) ++ zs
≡⟨ refl ⟩
  x :: ((xs ++ ys) ++ zs)
≡⟨ cong (_::_ x) (++-assoc xs ys zs) ⟩
  x :: (xs ++ (ys ++ zs))
≡⟨ refl ⟩
  (x :: xs) ++ (ys ++ zs)
■
```

Now our proof is similar to a detailed pen and paper proof! While easier to read, it may be a bit harder to maintain due to its verbosity.

Acknowledgements This document has been generated with lhs2TeX by Ralf Hinze and Andres Löh. A great tool!