Two presentations of equality in dependently typed languages

Li Nuo

March 17, 2011

1 Background

In intensional type theory such as Agda, propositional equality (establish equality based on identity) can be defined in two ways: either defined as an inductive relation or as a parameterized inductive predicate:

As a binary relation

```
data Id (A : Set) : A → A → Set where
  refl : (a : A) → Id A a a
```

This one was first proposed by Per Martin-Löf as propositional equality [2]. refl here is a function which creates a single equality term for each element of A.

As a predicate

```
data Id' (A : Set) (a : A) : A → Set where
  refl : Id' A a a
```

This version was proposed by Christine Paulin-Mohring [3] and is adopted in the Agda standard library. The formalisation of this identity type requires the equated value. Id' A a can be seen as a predicate of whether some x : A is the same as a in the type declaration. Since the equated value has been determined within the type, we only need an unique proof refl.

In intensional type theory, we have a corresponding elimination rule for each of them. It was called idpeel in [2] but we use the common name J here. We can easily define it using pattern matching in Agda as below.

As a binary relation

```
J : (A : Set) (P : (a b : A) → Id A a b → Set)
  → (m : (a : A) → P a a (refl a))
  → (a b : A) (p : Id A a b) → P a b p
J A P m a .a (refl .a) = m a
```

P is the proposition dependent on the equality. m is function to create inhabitant for this proposition when we we know the two sides of the identity term are definitionally equal. Using pattern matching we can easily derive that the equality term must only be constructed by the function refl and if it is inhabitant, the two sides must be identical. Then P a b p can be turned into P a a (refl a) so that we can trivially formalise it using m a.

The main ingredient of J is pattern matching $(a, b, p : Id\ A\ a\ b)$ with $(b, b, refl\ b)$.

As a predicate

```
J' : (A : Set) (a : A)
   → (P : (b : A) → Id' A a b → Set)
   → (m : P a refl)
   → (b : A) (p : Id' A a b) → P b p
J' A .a P m a refl = m
```

J' is the Paul-Mohring's rule. For Id' A a, since the type formalisation requires one side of the equality, we treat it as a predicate on the other side of the equality. Therefore, within the elimination rule, Id' A a is fixed, which means P, m and p now share the same free variable a as in Id' A a. Then m becomes the proof that P is only valid when the other side is the same as a. Also with pattern matching, $(b, p : Id'\ A\ a\ b)$ is identified with $(a, refl : Id'\ A\ a\ a)$. Hence, P b p can be turned into P a refl which is just m.

These two formalisation of equality can be used alternatively in many places. They should be ispomorphic in intensional type theory. We can easily prove it as below.

Firstly we establish the isomorphic functions between the two formulasation.

```
φ : ∀ {A} {a b : A} → Id A a b → Id' A a b
φ {A} {a} {b} eq = J A (λ a' b' _ → Id' A a' b') (λ a' → refl) a b eq
ψ : ∀ {A} {a b : A} → Id' A a b → Id A a b
ψ {A} {a} {b} eq = J' A a (λ b' _ → Id A a b') (refl a) b eq
```

Then we need to prove the two ways of composing them are simply identity functions. There are one small question that to use which formalisation of equality of the equality of the equality. However, if we have either of them, we have proved that the two equality are definitionally equal. Therefore we can choose either of them to make the proof as simple as possible.

```
idId : ∀ A (a b : A) → (p : Id A a b) → Id (Id A a b) (ψ (φ p)) p
idId A a b p = J A (λ a' b' eq → Id (Id A a' b') (ψ (φ eq)) eq)
  (λ a' → refl (refl a')) a b p
idId' : ∀ A (a b : A) → (p : Id' A a b) → Id' (Id' A a b) (φ (ψ p)) p
idId' A a b p = J' A a (λ b' eq → Id' (Id' A a b') (φ (ψ eq)) eq) refl b p
```

What is more interesting is the question below.

2 The Question

Now the question is: how to implement J using only J' and vice versa? We will still use corresponding equality to be used by each elimination rule.

3 The Solution

From J' to J is quite simple. If we assume a is the left hand side, and we rename P a with P', then J' A a P' (m a) can turn P' b p for some b into P' a refl which is just m a.

```
JId' : (A : Set) (P : (a b : A) → Id' A a b → Set)
  → ((a : A) → P a a refl)
  → (a b : A) (p : Id' A a b) → P a b p
JId' A P m a = J' A a (P a) (m a)
```

We can easily verfiy it by definitionally expanding J'.

The other direction is more tricky, because for J' we have proposition P, proof term m and equality term p dependent on the same value a which is not trivially suited for J. We must try to devise other ways to solve it.

We first define subst from J

```
id : {A : Set} → A → A
id = λ x → x
subst : (A : Set) (a b : A) (p : Id A a b)
  (B : A → Set) → B a → B b
subst A a b p B = J A (λ a' b' _ → B a' → B b') (λ _ → id) a b p
```

And then define Q A a as the product of some b with the equality between a and b

```
Q : (A : Set) (a : A) → Set
Q A a = Σ A (λ b → Id A a b)
```

Finally we prove J' from J (note that subst is also defined by J).

```
J'Id : (A : Set) (a : A) → (P : (b : A) → Id A a b → Set)
  → P a (refl a)
  → (b : A) (p : Id A a b) → P b p
J'Id A a P m b p = subst (Q A a) (a, refl a) (b, p)
  (J A (λ a' b' p' → Id (Q A a') (a', refl a') (b', p'))
  (λ a' → refl (a', refl a')) a b p) (uncurry P) m
```

The idea behind it is that we need to identify P b p with P a refl. We cannot substitute b with a or p with refl a independently because the second argument is dependent on the first argument. we must substitute them simultaneously. Therefore it is natural to use a dependent product. It happens simultaneously when we use pattern match to prove it.

We use a function Q to form the dependent product. To substitute (b, p) with (a, refl a) we must have the equality term Id (Q a) (a, refl a) (b, p). We can easily get this term using J. So we have done the proof. Also we can expand it definitionally to verfy it.

The question shows that no matter which formulation of equality and which elimination rule we have, we can prove the other elimination rule for the other formulation of equality.

4 Proof of equivalence relation

Based on the elimination rule, we can then implement the proof in [2] that Id is an equivalence relation.

Reflexivity is trivially in the definition so we only need to prove symmetric and transitivity.

```
sym : (A : Set) → (a b : A) → Id A a b → Id A b a
sym A = J A (λ a b _ → Id A b a) refl
```

The definition is simplified based on the η reduction.

```
trans : (A : Set) → (a b c : A) → Id A a b → Id A b c → Id A a c
trans A a b c = J A (λ a b _ → Id A b c → Id A a c) (λ _ → id) a b
```

The one for Id' is similar, but due to a is the present in more than one place in the elimination rule, we can not simplfy the definition that much.

```
sym' : (A : Set) → (a b : A) → Id' A a b → Id' A b a
sym' A a = J' A a (λ b _ → Id' A b a) refl
```

```
trans' : (A : Set) → (a b c : A) → Id' A a b → Id' A b c → Id' A a c
trans' A a b c = J' A a (λ b _ → Id' A b c → Id' A a c) id b
```

By these properties, we show that both Id and Id' are equivalence relations.

5 Identity of equality

The identity of equality is obvious from the oberservation of the definition of the equality. However we cannot prove it using only J, because J only reveals a and b is convertible if we have the equality. What is missing from the pattern matching is that the only inhabitant of the equality type refl.

However Hedberg [1] has proved that if we know Id is decidable, then we can prove the UIP (uniqueness of identity proof). I have implemented the theorem in Agda. We have proved Identity elimination J, Identity coercion subst, Identity composition trans, Identity inverse sym. Then we need to prove following lemmas,

Identity mapping

```
resp : (A B : Set) → (f : A → B) → (a b : A) → Id A a b → Id B (f a) (f b)
resp A B f = J A (λ a' b' _ → Id B (f a') (f b')) (λ a' → refl (f a'))
```

A groupoid law - inverse law

```
invrl : (A : Set) (a b : A) (u : Id A a b) →
  Id (Id A b b) (trans A b a b (sym A a b u) u) (refl b)
invrl A a b u = J A (λ a b u → Id (Id A b b)
  (trans A b a b (sym A a b u) u) (refl b)) (λ b → refl (refl b)) a b u
```

Constancy lemma

```
con : (A : Set) → Dec A → A → A
con A (yes p) _ = p
con A (no _) a = a

iscon : (A : Set) → (d : Dec A) → (a a' : A) → Id A (con A d a) (con A d a')
iscon A (yes p) a a' = refl p
iscon A (no ¬p) a a' with ¬p a
iscon A (no ¬p) a a' | ()
```

If A is true, the function will always return a constant proof of A, otherwise it will work as identity function. However when A is false, there is no inhabitant of A, therefore it must be a constant function.

Collapse lemma

```
collaps : (A : Set) (f : A → A)
  (is_c : ∀ x x' → Id A (f x) (f x'))
  (g : A → A)
  (is_li : ∀ x → Id A (g (f x)) x) (a b : A) → Id A a b
collaps A f is_c g is_li a b = trans A a (g (f a)) b (sym A (g (f a)) a (is_li a))
(trans A (g (f a)) (g (f b)) b (resp A A g (f a) (f b) (is_c a b)) (is_li b))
```

Left inverse lemma

```
leftinv : (A : Set) (nt : (x y : A) → Id A x y → Id A x y) →
  (a b : A) → Id A a b → Id A a b
leftinv A nt a b v = trans A a a b (sym A a a (nt a a (refl a))) v

isleftinv : (A : Set) (nt : (a b : A) → Id A a b
  → Id A a b) (a b : A) (u : Id A a b)
  → Id (Id A a b) (leftinv A nt a b (nt a b u)) u
isleftinv A nt a b u = J A
  (λ a b u → Id (Id A a b) (leftinv A nt a b (nt a b u)) u)
  (λ x → invrl A x x (nt x x (refl x))) a b u
```

DI⊆CI-theorem

```
condi : (A : Set) (di : Decidable (Id A)) →
  (x y : A) (u : Id A x y) → Id A x y
condi A di x y u = con (Id A x y) (di x y) u
```

Theorem : UIP of decidable identity type

```
dici : (A : Set) → Decidable (Id A) →
  ∀ (x y : A) (u v : Id A x y) → Id (Id A x y) u v
```

```
dici A di x y u v = collaps (Id A x y)
  (condi A di x y) (iscon (Id A x y) (di x y))
  (leftinv A (condi A di) x y) (isleftinv A (condi A di) x y) u v
```

Finally we proved that if Id A is decidable e.g. Id Bool, then we have have UIP for that type.

References

[1] Hedberg Michael. A coherence theorem for martin-löf's type theory. 1998.

[2] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory: an introduction*. Clarendon Press, New York, NY, USA, 1990.

[3] Streicher Thomas. *Investigations into Intenstional Type Theory*. Habilitation Thesis, Ludwig-Maximilians Universität, Munich, 1993.