1 Background

We can define equality in two ways which are similar but somewhat different in the meaning of the definition.

1.1 As a binary relation

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

We can treat equality relation as the predicate of whether certain ordered pairs is belonging to Id A. We can decribe it in another way: it is a partition of the set A x A.

1.2 As a predicate

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

This one is just what we use in standard library. We can treat (Id A a) it as a predicate of whther certain element of A is the same as a. It also represents the singleton set including only one element refl.

For each of them, we have a corresponding elimination rule, defined as

```
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 .b b (refl .b) = m b
```

The P and m are both indexed by different 'a'. P is actually a trinary relation.

m can be seen as a property of P. For all a, <a , a, refl a> is inhabited in P. And the result is a more general property, For all a b, <a , b, x : Id A a b> is inhabited in P.

J actually changes

```
∀ (a : A) → P a a (refl a)
```

to

```
∀ (a b : A) (p : Id A a b) → P a b p
```

```
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 .b P m b refl = m
```

The P and m are now restricted by the same 'a' as the the predicate identity. P and m here are actually special cases of the P (P [a]) and m (m [a]) in J. 'a' can be regarded as a constant in the discourse.

J' actually changes

```
P a refl
```

to

```
(b : A) (p : Id' A a b) → P b p
```

. m can be seen as the only object in P and the result is used to unify elements equal to a (a constant) to get the unique object.

2 The Problem

Now the problem is: how to implement J using only J' (also we use the equality Id') and vice versa? We will still use corresponding equality to each elimination rule, otherwise it cannot eliminate the identity.

3 Solution

The first direction is quite simple. When we eliminate the predicate identity, we only need to create the special cases of P and m for J'.

```
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)
```

The other direction is more tricky. We first define 'subst' from J

```
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 (lambda a' b' _ → B a' → B b') (lambda _ → id) a b p
```

Then to prove J' from J and Id,

```
Q : (A : Set) (a : A) → Set
Q A a = sigma A (lambda b → Id A a b)
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 (lambda a' b' x → Id (Q A a') (a', refl a') (b', x))
  (lambda a' → refl (a', (refl a'))) a b p) (uncurry P) m
```

We can not just use J to eliminate the identity because it requires more general P and m. We need to formalise the result P b p from P a (refl a). We cannot substitute a or (refl a) singlely because the second argument is dependent on the first argument. So when we substitute we should reveal the dependent relation.

We could use dependent product to do this work. In this way, we can substitute them simultaneously. The problem now becomes substitute in

```
P ((lambda a : A p : Id A a b → (a, p)) a (refl a))
```

to

```
P ((lambda a : A p : Id A a b → (a, p)) b p)
```

From J, we have Id (Q a) (a , refl a) (b , x : Id a b) so that we can prove P' <b , p> from P' <a , (refl a)> using subst. Because P' < b , p> is namely P b p, we have proved.