A taste of type theory

Bartosz Milewski

Outline

- Motivation: dealing with equalities in type theory
- Recursive types
- Dependent types
- Induction
- Propositions as types
- Identity types
- Homotopy Type Theory

Equalities

```
((), a) ~ a
```

- 1 * a = a
- Isomorphism vs. equality
- Equational reasoning

```
lu :: forall a. ((), a) -> a
lu ((), x) = x

lu_inv :: forall a. a -> ((), a)
lu_inv x = ((), x)

lu . lu_inv = id
lu_inv . lu = id
```

Natural Numbers

- Introduction
- Elimination (primitive recursion)
- Computation
- (Uniqueness)

```
data Nat where
  Z :: Nat
  S :: Nat -> Nat
```

```
base :: a
step :: Nat -> a -> a

rec :: a
    -> (Nat -> a -> a)
    -> Nat -> a
```

```
-- factorial
base     = 1
step n x = (S n) * x
```

```
-- computation
f = rec base step
f Z     = base
f (S n) = step n (f n)
```

Dependent Types

- Example of type family indexed by Nat
- Different type for each number

```
data Vec a (n :: Nat) where
  Nil  :: Vec a Z
  Cons :: (n :: Nat) -> a -> Vec a n -> Vec a (S n)
```

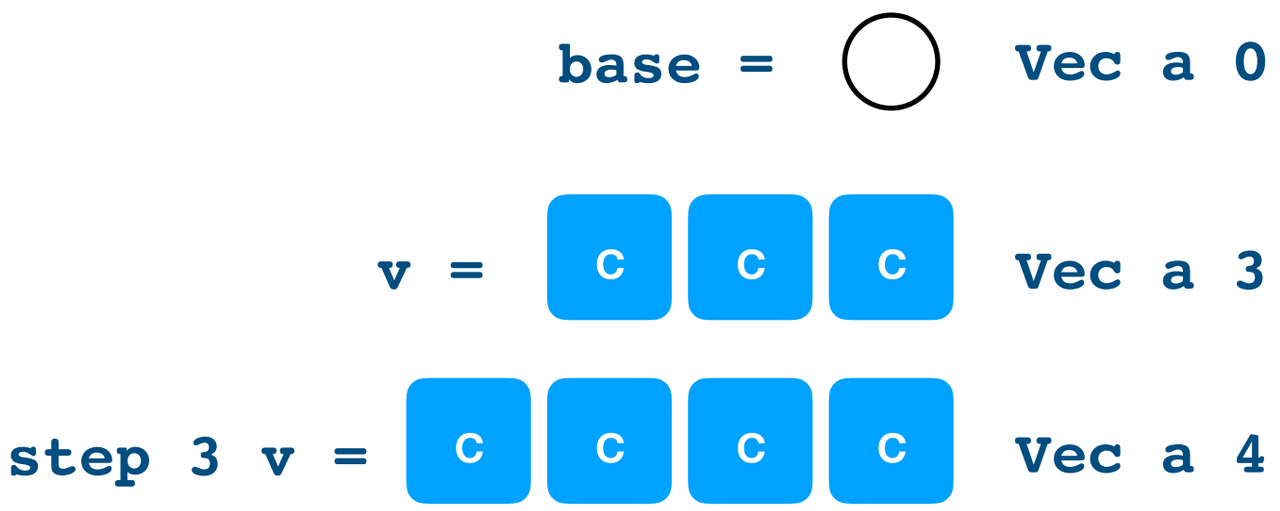
Induction

- Dependent elimination
- Type family (e.g., Vec)
- dependent base and step

```
base :: Vec a Z
step :: (n :: Nat)-> Vec a n -> Vec a (S n)
```

- Example: replicate

```
base = Nil
step n v = Cons n c v
```



Induction on Nats

- Given type family C, base, and dependent step

```
base :: C Z
step :: (n :: Nat)-> C n -> C (S n)
```

```
ind :: C Z
    -> (forall m. (m :: Nat) -> C m -> C (S m))
    -> (n :: Nat)  -> C n
```

- Computation rule

```
f = ind base step
f Z = base
f (S n) = step n (f n)
```

Curry Howard

- Propositions as types
    - Inhabited type = true proposition
    - Proof of proposition = construction of a value of a given type
        - Proof of (A and B) is a pair of a :: A and b :: B
- There may be many proofs of the same proposition

Induction

- Type family: For each n, proposition C n

- base: proof of C for n=0

```
base :: C Z
```

- step: proof of C(n+1) given proof of C(n)

```
step :: (n :: Nat)
     -> C n
     -> C (S n)
```

- => proof of C for any n

```
ind :: C Z
    -> (forall m. (m :: Nat) -> C m -> C (S m))
    -> (n :: Nat) -> C n
```

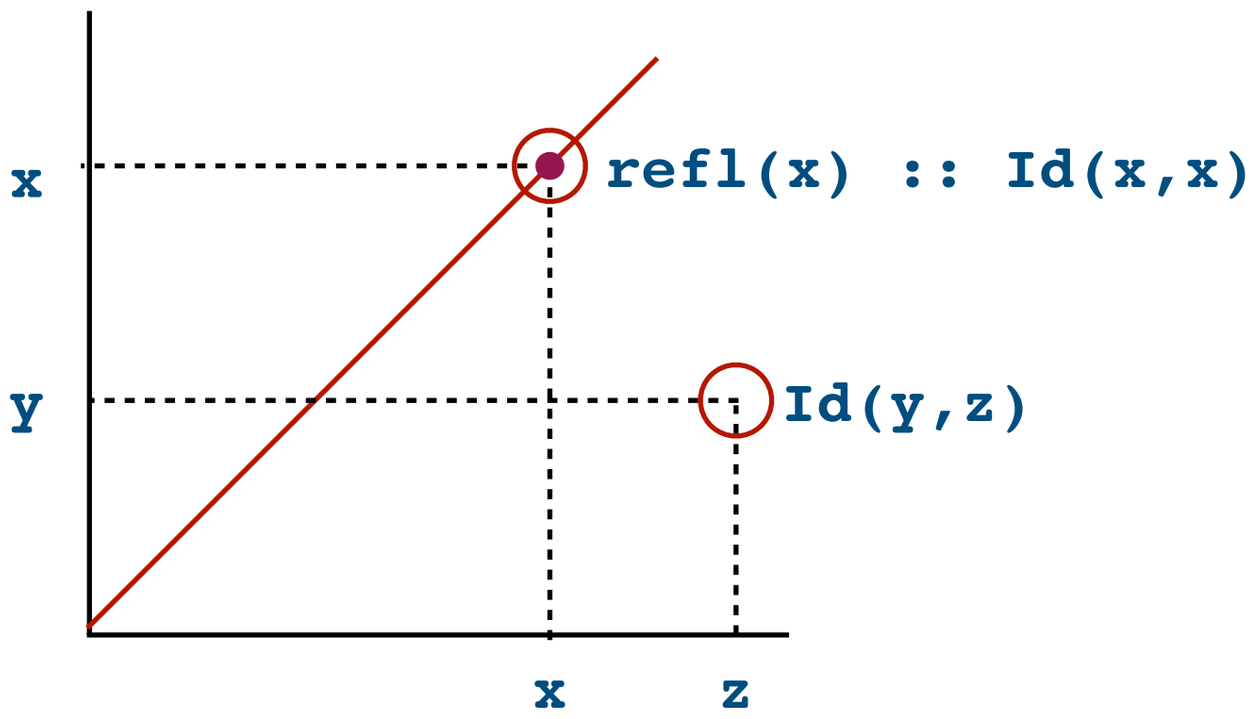
- Induction = dependent elimination of Nat

Identity Type

- Given $x :: a$ and $y :: a$, is x equal to y?
- Translation: is type $Id(x, y)$ inhabited?
- $Id(x, y)$ is a type family indexed by values of x and y
- Infix notation: $x =_a y$
- Notation $p :: x =_a y$,
  - p is an element (proof) of x being equal to y

refl

- Introduction: it's always true that $x =_a x$
- Constructor: $refl_a(x) :: x =_a x$
- Types on the diagonal always inhabited



Elimination

- Identity is a (doubly) dependent type
- By analogy with induction on Nats, a type family parameterized by identity type value (equality proof) *p*
- *p* is also called a *path* between *x* and *y*
- Type family (set of propositions) `(n :: Nat) -> C n`

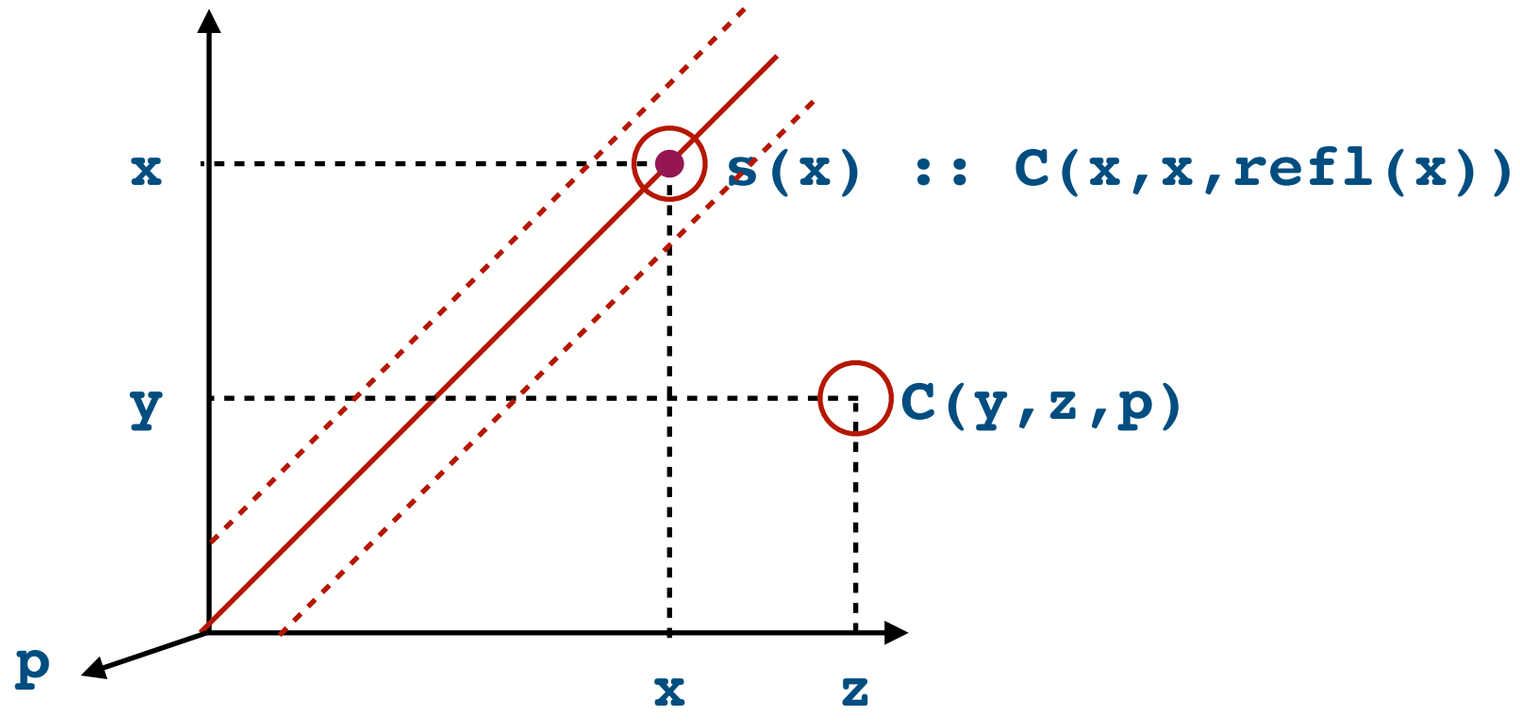
`(x :: a) -> (y :: a) -> (p :: x =`$_a$ `y) -> C(x, y, p)`

Elimination

- Two intros: Z and S
- The only intro is *refl*
- Type family *C(x, y, p)*
- Type *C(x, x, refl(x))*
- Tiny recursive step: *s*

```
base :: C Z
step :: (n :: Nat)
     -> C n
     -> C (S n)


s :: (x :: a)
  -> C (x, x, refl(x))
```



Elimination

- There exists a function defined for all n

```
ind :: C Z
    -> (forall m. (m :: Nat) -> C m -> C (S m))
    -> (n :: Nat) -> C n
```

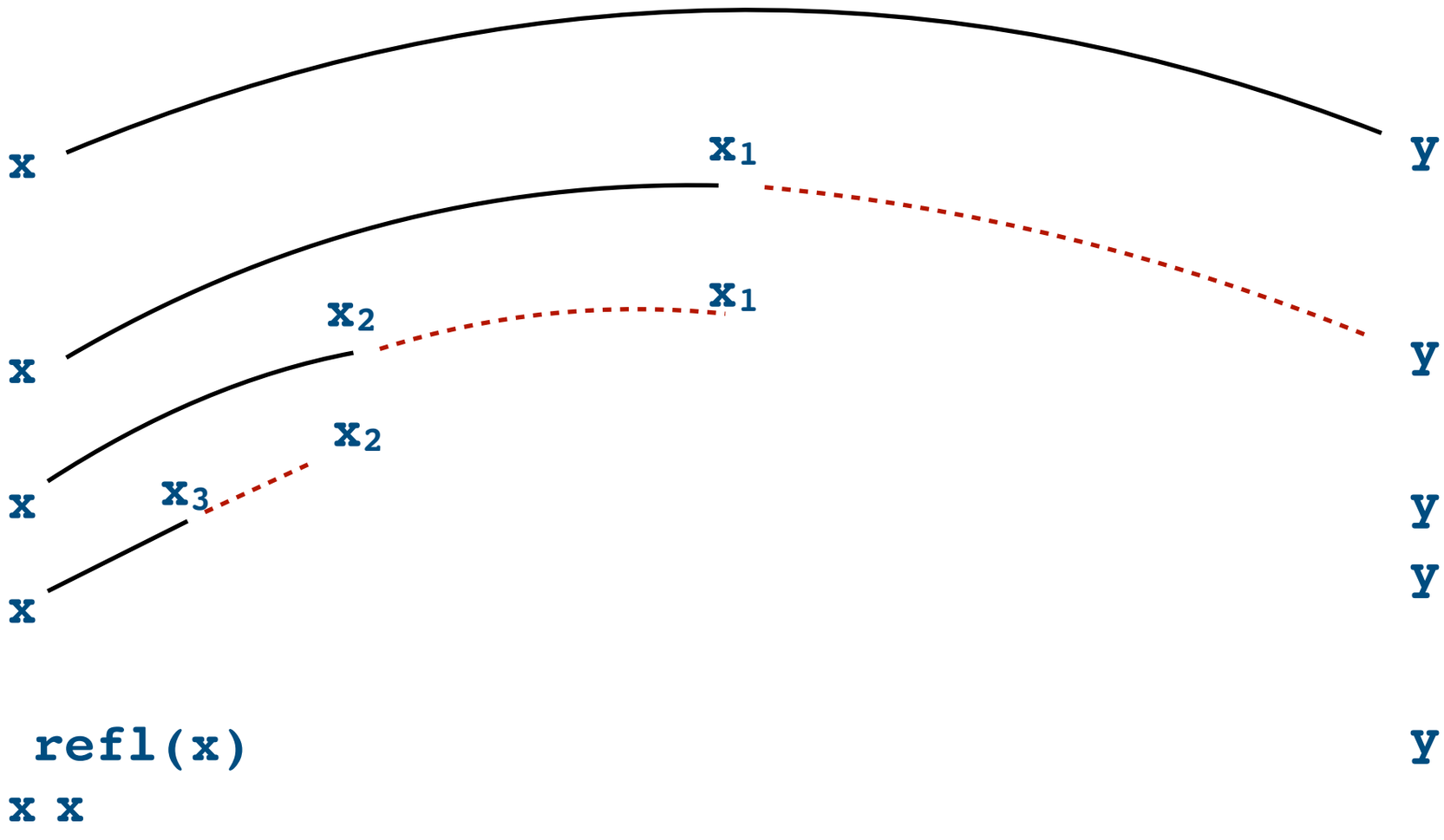
- There exists a function defined for all x, y, p

```
ind :: (forall z. (z :: a) -> C (z, z, refl(z)))
    -> (x :: a) -> (y :: a) -> (p :: x = y) -> C(x, y, p)
```

- Corresponding computation rule

```
ind s x x (refl(x)) = s x
```

Zeno's Paradox



- Every journey begins with reflection