OPLSS 22 Introduction to Type Theory (4)

Cosmo Viola

June 2022

Today, we're going to talk about dependent types. A dependent type is a function whose codomain is Set. Polymorphism is a special case of dependent types, where the function takes a type as input; consider the constructor $List : Set \to Set$. However, dependent types can depend on any value. Consider the type of vectors, which are lists of a given length:

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A 0
  _ :: _ A → Vec A n → Vec A (suc n)
```

We can define a function that produces vectors of a given length containing all zeros.

```
zero : (n : ℕ) → Vec A n
zeros zero = []
zeros (suc n) = 0 :: zeros n
```

We can define a function that appends vectors together. The type of the output will depend on the length of the input vectors.

```
_++v_ : Vec A m → Vec A n → Vec A (m + n)
[] ++v bs = bs
(a :: as) ++v bs = a :: (as ++v bs)
```

Because the type of vectors comes bundled with the vector's length, we can have static array bounds checking, in contrast to List where we don't know in advance if we'll find an element or not.

```
_!!_ : List A → ℕ → Maybe A
[] !! n = nothing
(a :: as) !! zero = just a
(a :: as) !! suc n = as !! n
```

We use the Maybe type for the output to represent the case where the array index is out of bounds.

Next, we define a family of finite types parameterized by their number of elements. We can use this type to reimplement our statically checked list access, but without the need for the Maybe type output.

```
data Fin : ℕ → Set where
  zero : Fin (suc n)
  suc : Fin n → Fin (suc n)

_!!v_ : Vec A n → Fin n → A
(a :: as) !!v zero = a
(a :: as) !!v suc i = as !!v i
```

Next, we will talk about Π-types and Σ-types. Π-types are also called dependent product types, though they work like functions; an element of a Π-type is a function where the output type depends on the input to the function. A Σ-type, on the other hand, is a pair where the type of the second element depends on the type of the first.

```
Π : (A : Set) (B : A→ Set) → Set
Π A B = (x : A) → B x

record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁

FlexVec : Set → Set
FlexVec A = Σ ℕ λ n → Vec A n
```

This type, FlexVec, is equivalent to List.

It turns out that we can define the coproduct using Σ-types.

```
_⊎_ : Set → Set → Set
A ⊎ B = Σ Bool AorB
  where AorB : Bool → Set
        AorB : true = A
        AorB : false = B
```

We can also define products using Π-types.

```
_ ×' _ = Set → Set → Set
A ×' B = Π Bool AorB
  where AorB : Bool → Set
        AorB true = A
        AorB false = B
```

With Π-types and Σ-types, we can give the propositions as types interpretation for predicate logic. A predicate will be of the form $A \to Set$ and a relation of the form $A \to B \to Set$. Then, we need to define quantifiers.

```
Forall : (A : Set) (P : A → Set) → Set
Forall A P = Π A P

Exists : (A : Set) (P : A → Set) → Set
Exists A P = Σ A P
```

Then, we can write propositions using these quantifiers. For instance, we can write the following:

```
f : (x : ℕ) → Even x ∨ Odd x

isPrime : ℕ → Set
Σ ℕ isPrime
```

The first of these asserts that all naturals are either even or odd, while the second asserts that there exists a prime natural.

Here are some tautologies of predicate logic:

$$\forall x \in A, P\ x \vee Q\ x \iff (\forall x \in A, P\ x) \vee (\forall x \in A, Q\ x)$$

$$\forall x \in A, (P\ x \to R) \iff (\exists x \in A, P\ x) \to R$$

We can prove the analogues for Σ and Π types.

Next, we will discuss equality. Here is the type of equality in Agda:

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

The only constructor of this equality is reflexivity. Equality is an equivalence relation, so it should also be symmetric and transitive. We can prove these:

```
sym : (a b : A) → a ≡ b → b ≡ a
sym refl = refl

trans : {a b c : A} → a ≡ b → b ≡ c → a ≡ c
trans refl q = q

cong : (f : A → B) → (a b : A) → a ≡ b → f a ≡ f b
cong f refl = refl
```

Notice that the following is not true:

```
uncong : (f : A → B) → (a b : A) → f a ≡ f b → a ≡ b
```

Given addition on the natural numbers, we can prove associativity:

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)

assoc : (l + m) + n ≡ l + (m + n)
assoc {zero} {m} {n} = refl
assoc {suc l} {m} {n} = cong suc (assoc {l} {m} {n})
```

Proof by induction is equivalent to dependent recursion. The eliminator generalizes the iterator to dependent types.

```
Eℕ : (M : ℕ → Set) → M zero
          → ((m : ℕ) → M m → M (suc m))
          → (m : ℕ) → M m
Eℕ M m−0 m−s zero = m−0
Eℕ M m−0 m−s (suc n) = m−s n (Eℕ M m−0 m−s n)
```

For fac, you need a recursor.

```
It−ℕ : M → (M → M) → ℕ → M
R−ℕ : M → (ℕ → M → M) → ℕ → M
```

Exercise: R-ℕ is derivable from It-ℕ.

```
R−ℕ = Eℕ (λ _ → M)
```

Eℕ is not derivable from It-ℕ. Eℕ comes from the initiality of ℕ.

Do coinductive types have a coeliminator?

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A

CoIt−Stream : (M → A) → (M → M) → M → Stream A

postulate
  CoInd : (R : Stream A → Stream A → Set)
        → ({as bs : Stream A} → R as bs
          → (head as ≡ head bs)
            × (R (tail as) (tail bs)))
        → {}as bs : Stream A} → R as bs → as ≡ bs
```

Execsise: prove the following using CoInd.

```
mapS suc (from 0) ≡ from (suc 0)
```

With refl, we can only prove things equal that are identical. This is a consequence of intensional type theory.

Because we postulated CoInd, the computational behavior (canonicity) of our system has been destroyed. Now, there are closed natural numbers that are not numerals.