



# Theorembeweiserpraktikum

## Inductive Types

Jakob von Raumer, Sebastian Ullrich | SS 2022



THEOREM PROVER

# Outline

- Type families
- Dependent function types
- Recursive definitions on inductive types
- Structures

# Type Families

- We have seen last time: *Predicates* take the form of functions  $p : \alpha \rightarrow \mathbf{Prop}$  for some type  $\alpha$ .
- What about functions  $\beta : \alpha \rightarrow \mathbf{Type}$ ? Remember that we call these *type families* on  $\alpha$ .
- Mathematically, they can be used to model set-valued functions.
- Some common examples for type families:
  - The family  $\mathbf{Fin} : \mathbf{Nat} \rightarrow \mathbf{Type}$  of *finite ordinal types*.
  - The family  $\mathbf{Vec} : \mathbf{Type} \rightarrow \mathbf{Nat} \rightarrow \mathbf{Type}$  of *vectors* (lists of a given length).
  - The family  $\mathbf{Bij} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \mathbf{Type}$  of *bijections* between types.

# Dependent Functions

- What happens if we take the universal quantifier  $\forall x, \beta x$  of such type families?
- We can see them as functions that assign to each  $x : \alpha$  an element of the type  $\beta x$ .
- Example: The function which returns the vector repeating an element  $x : \alpha$  a given number of times:  $\text{repeat} : \alpha \rightarrow \forall n : \text{Nat}, \text{Vec } \alpha n$ .
- Lean offers an alternative notation to the above:  $\text{repeat} : \alpha \rightarrow (n : \text{Nat}) \rightarrow \text{Vec } \alpha n$ .
- In definitions:

```
def repeat ( $\alpha : \text{Type}$ ) ( $x : \alpha$ ) ( $n : \text{Nat}$ ) :  $\text{Vec } \alpha n := \dots$ 
```

# Implicit Arguments

- Suppose we have a function `append` which appends two vectors. Its type would be

```
def append (m n : Nat) (v : Vec α n) (w : Vec α m) : Vec α (m + n) := ...
```

- The value of the first and second argument is *uniquely determined* by the third and fourth!
- We can use `{...}` to mark such arguments as *implicit*:

```
def append {m n : Nat} (v : Vec α n) (w : Vec α m) : Vec α (m + n) := ...
```

to use the function as `append v w` instead of `append m n v w`.

- Undeclared variables are automatically turned into implicit arguments, so we could just leave out `{m n : Nat}` above.
- We can use `@append` to make all arguments explicit, and `append (n := ...)` to do so for `n` only.

# Inductive Types: Natural Numbers

- So far, all inductive types we have defined were *non-recursive*.
- More interesting and especially infinite types can be constructed using *recursive* constructors.
- Example: The type of natural numbers `Nat` :

```
inductive Nat : Type where  
  | zero : Nat  
  | succ : Nat → Nat
```

- Intuitively, the type is supposed to contain all terms which can be composed using the two constructors:

```
zero  
succ zero  
succ (succ zero)  
succ (succ (succ zero))  
...
```

# Inductive Types: Constructors and Recursors

- The presence of the constructors ensures that `Nat` contains *at least* those terms.
- How to state that `Nat` does not contain more?  $\rightsquigarrow$  *Recursors*!
- The definition of `Nat` postulates, besides `Nat.zero` and `Nat.succ`, the following function:

```
#check @Nat.rec
/- output:
  Nat.rec : {motive : Nat → Sort u_1} →
    motive Nat.zero → ((n : Nat) → motive n → motive (Nat.succ n)) → (t : Nat) → motive t
-/
```

- Special cases:
  - `motive := fun _ => α` : Recursively defined functions on `Nat`.
  - `motive := fun n => p n` for a predicate `p` : Proof by induction.

# Recursive `match` blocks

Recursors are best not used directly, Lean's `match` syntax is a convenience feature to do this for us:

```
def add (m n : Nat) : Nat :=  
  match n with  
  | Nat.zero   => m  
  | Nat.succ n => Nat.succ (add m n)
```

produces the same result as

```
def add' (m : Nat) : Nat → Nat := @Nat.rec (fun _ => Nat) m (fun _ r => Nat.succ r)
```

Note: This means that  $m + (n + 1)$  is definitionally equal to  $(m + n) + 1$ , but not  $(m + 1) + n$  to  $m + (1 + n)$  !



# Indexed Inductive Types

- Like with predicates, we can define inductive type families with *indices*.
- Example: A definition of the type of vectors could look like this:

```
inductive Vec (α : Type) : Nat → Type where
| nil   : Vec α 0
| cons  : {n : Nat} → α → Vec α n → Vec α (n + 1)
```

- We now can define the `append` function as follows:

```
def append (v : Vec α m) (w : Vec α n) : Vec α (n + m) :=
  match v with
  | Vec.nil      => w
  | Vec.cons a v => Vec.cons a (append v w)
```

Here the last line works because  $(n + m) + 1$  is definitionally equal to the expected index  $n + (m + 1)$ .

# Structures

- You have already seen examples of inductive types with exactly one constructor ( And , Iff ).
- Those have some special properties and syntax in Lean. The code

```
structure Point (α : Type) where  
  x : α  
  y : α
```

means roughly the same as

```
inductive Point (α : Type) : Type where  
  | intro : (x : α) → (y : α) → Point α
```

# Structures

- Structures offer notation access to the constructor arguments (*fields*), as you have seen in `And.left` :

```
def fivethree : Point Nat := { x := 5, y := 3 }  
  
#eval fivethree.x -- prints 5
```

- Structures can be extended by structures with more fields:

```
structure Triplet (α : Type) extends Point α where  
  z : α
```

contains the fields `x` , `y` , and `z` .

# Namespaces

- *Namespaces* are a way to organize names of definitions hierarchically.
- Constructors and recursors of inductive types are automatically put into a namespace of the same name, see `Vec.nil`.
- We can **open** namespaces, to remove the prefix:

```
namespace Topic -- starts a namespace

def exampleDef := ...

end Topic -- ends a namespace
#check Topic.exampleDef

open Topic -- removes the necessity for the prefix "Topic."
#check exampleDef
```

- Definitions marked with **protected** keep their prefix when the namespace is opened.