Karlsruhe Institute of Technology

Theorembeweiserpraktikum

Predicate Logic

Jakob von Raumer, Sebastian Ullrich | SS 2021

THEOREM PROVER

Predicate Logic

Predicate logic extends propositional logic by allowing us to reason about properties of *objects*

```
variable (p q : α → Prop)  -- p and q are predicates over the type α
```

The *universal quantifier* allows us to state that a property holds for *every* object (of a given type/"set")

```
example : (∀ x : α, p x) → (∀ x : α, q x) → (∀ x : α, p x ∧ q x) := ...
```

(compare with: $\forall x \in A. P(x) \wedge Q(x)$ etc.)
How do we prove such a statement? What type does $\forall$ correspond to and what λ-terms inhabit it?

Predicate Logic

Recall that the implication `p → q` can be seen as a function mapping a proof/element of `p` to one of `q`

Then `∀ x : α, p x` seems to be related: it should map any element `x` of `α` to one of `p x`

Thus `∀ x : α, p x` is the type of *dependent functions*: the output type depends on the input *value*

Functions whose output type is constant are now a special case:

$$p \to q \quad \equiv \quad \forall\ \_ : p,\ q$$

Now we know how to reason about ∀: using the same function abstraction and application as before!

```
example : (∀ x : α, p x) → (∀ x : α, q x) → (∀ x : α, p x ∧ q x) :=
  fun hp hq =>  -- goal is now `... ⊢ ∀ x : α, p x ∧ q x`
    fun x =>    -- goal is now `..., x : α ⊢ p x ∧ q x`
      And.intro (hp x) (hq x)  -- `hp x : p x`
```

Predicate Logic

Recall that the implication `p → q` can be seen as a function mapping a proof/element of `p` to one of `q`

Then `∀ x : α, p x` seems to be related: it should map any element `x` of `α` to one of `p x`

Thus `∀ x : α, p x` is the type of *dependent functions*: the output type depends on the input *value*

Functions whose output type is constant are now a special case:

$$\texttt{p → q} \quad \equiv \quad \texttt{∀ \_ : p, q}$$

Now we know how to reason about $\forall$: using the same function abstraction and application as before!

```
example : (∀ x : α, p x) → (∀ x : α, q x) → (∀ x : α, p x ∧ q x) :=
  fun hp hq =>  -- goal is now `... ⊢ ∀ x : α, p x ∧ q x`
    fun x =>    -- goal is now `..., x : α ⊢ p x ∧ q x`
      And.intro (hp x) (hq x)  -- `hp x : p x`
```

We'll see how to deal with the existential quantifier $\exists$ in the exercises

Predicates

Predicates are just functions into `Prop` and so can be defined in terms of other predicates:

```
def BadDay : Weekday → Prop := fun w => ¬ GoodDay w
```

Predicates

Predicates are just functions into `Prop` and so can be defined in terms of other predicates:

```
def BadDay : Weekday → Prop := fun w => ¬ GoodDay w
```

Fundamentally, predicates are introduced as inductive types!

```
inductive GoodDay : Weekday → Prop where
  | sat : GoodDay Weekday.saturday
  | sun : GoodDay Weekday.sunday
```

Thus we can reason about them using, again, constructor application and `match`

Predicates

Working with `GoodDay` is very similar to `Or`, but note a crucial difference in their definition:

```
-- parameters to the left of `:`, fixed in constructors
inductive Or (p q : Prop) : Prop where
  | inl (hp : p) : Or p q
  | inr (hq : q) : Or p q

-- parameter to the right of `:`, can vary in constructors
inductive GoodDay : Weekday → Prop where
  | sat : GoodDay Weekday.saturday
  | sun : GoodDay Weekday.sunday
```

"Basic" algebraic data types do not allow such varying parameters, which are called *indices*, and the resulting type a *type family*. The names come from the fact that we can think of the family as generating a specific type with a custom set of constructors given a specific index `w : Weekday`:

$$\mathtt{GoodDay}_{\mathtt{w}} : \mathtt{Prop}$$

Predicates

We might want to define `GoodDay` instead as

```
def GoodDay : Weekday → Prop := fun w => w = Weekday.saturday ∨ w = Weekday.sunday
```

We already know `∨` ... but how do we define `=` ?

Equality

```
inductive Eq : α → α → Prop where
  | refl (a : α) : Eq a a

infix:50 " = " => Eq
```

Fundamentally the single constructor tells us:

Two things are equal when they are the same!

This definition can also be thought of as stating that `Eq` is "the least reflexive relation" (credits: Andrej Bauer)

Things that are the same in Lean

Karlsruhe Institute of Technology

```
variable (x : α) (f : α → β)

example : f x = f x := Eq.refl (f x)

example : (f x) = f x := rfl  -- same as `Eq.refl _`
```

Structurally equal terms are the same

Things that are also the same in Lean

```
variable (x : α) (f : α → β)

theorem α_conv : (fun x => f x) = (fun y => f y) := rfl

theorem β_conv : (fun y => f y) x = f x := rfl

def id : α → α := fun x => x
theorem δ_conv : id x = x := rfl
-- includes reducing `match`

theorem η_conv : (fun x => f x) = f := rfl
```

We say two terms e and f are *definitionally equal* ($e \equiv f$) when they are convertible under the above rules

Definitional equality is an internal, automatically proved predicate and cannot be written down as a proposition; we use our *propositional* equality $e = f$ for that

Things that also equal in Lean

Karlsruhe Institute of Technology

```
axiom propext : (p ↔ q) → p = q
```

We can also (carefully) introduce new propositional equalities as axioms

Reasoning about `Eq`

```
theorem Eq.comm : ∀ x y : α, x = y → y = x :=
  fun x y h =>  -- goal: `..., h : x = y ⊢ y = x`
    match h with
    | Eq.refl x =>  -- goal: `... ⊢ x = x`
      rfl
```

Matching against `Eq.refl` *forces* y to be the same as x!

Reasoning about `Eq`

```
theorem Eq.comm : ∀ x y : α, x = y → y = x :=
  fun x y h =>  -- goal: `..., h : x = y ⊢ y = x`
    match h with
    | Eq.refl x =>  -- goal: `... ⊢ x = x`
      rfl
```

Matching against `Eq.refl` *forces* y to be the same as x!

Doing basically the same with a special syntax:

```
theorem Eq.comm : ∀ x y : α, x = y → y = x :=
  fun x y h =>
    -- "substitute in" operator, input as `\t`
    h ▸ rfl  -- goal at `rfl`: `... ⊢ x = x`
```

We will later introduce special *tactics* for even easier equational reasoning

Summary

type	*introduction*	*elimination*
`p`	`... ⊢ p`	`h : p, ... ⊢ ...`
implication/function type	*abstraction*	*application*
`q → r`	`fun hq => (_ : r)`	`h hq`
univ. quant./dep. function type		
`∀ x : α, r x`	`fun x => (_ : r x)`	`h x`
inductive type	*constructor app.*	*matching*
`x = y`	`rfl`	`h ▸ _`
	(given `x` ≡ `y`)	(implicit match against `Eq.refl`)

Next Steps

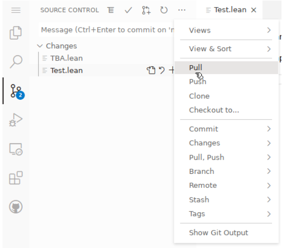
updated `tba-2021` repo with

- these slides at `slides/lecture2.pdf`
- exercise sheet #2 at `TBA/Exercises/Exercise2.lean`
- sample solutions for exercise sheet #1 at `TBA/Solutions/Exercise1.lean`
  - Take a look to learn about secret techniques & syntax sugars!
- new Lean version (set in `leanpkg.toml`)
  - removed some confusing error messages
  - easier Windows installation (no more missing DLLs)

Next Steps

To get the new exercise sheet,

- either open `https://gitpod.io/#/https://github.com/IPDSnelting/tba-2021/` again, creating a fresh workspace
  - Note: unused workspaces are removed after 14 days by default
- or run `git pull` in your existing workspace, e.g. via the “Source Control” tab (Ctrl+Shift+G), to keep your changes



  - Then run `F1 > Lean 4: Restart Server` just to be safe