Univers∀l Qu∀ntification

Mapping (an aside)

What do we mean by "mapping"

- What do we mean when we say an implication maps from a *proof of P* to a *proof of Q*?
  - $P \to Q$
  - Imagine the implication as directions telling you how to get from *P* to *Q* (e.g., from the bookstore to Rice Hall)
  - In this case, the directions are only useful if you are at the bookstore, or know how to get to the bookstore
- What do we mean when we say a function maps from *A* to *B*?
  - This is more general — a function mapping from the natural numbers to the booleans means the function takes a natural number as an input (the *domain*) and returns a boolean (the *range*)

Caution about use of the word “mapping”

- Lean (and other languages) have *maps* that take specific sets of inputs and return specific sets of outputs
  - For example, a phone book could be a (rather large) map — it maps names to phone numbers
  - A dictionary is another example
- We will not be discussing these types of maps in more detail yet

Univers∀l Qu∀ntification

Notation

- In general mathematics:
  - $\forall\, n \in \mathbb{N}: n + 1 \neq 0$
- In Lean:
  - ∀(n : nat), n + 1 ≠ 0
    - Parentheses around (n : nat) are optional
- Read this as:
  - "For all n that are members of $\mathbb{N}$, n + 1 is not equal to zero"
- This is just another proposition
  - #check ∀ n : nat, n + 1 ≠ 0

Fly You Fools!

- "You can fool all of the people some of the time…"

```
variables People Time: Type
variable fool: People → Time → Prop

#check ∀(p: People), ∃(t: Time), (fool p t)
#check ∃(t: Time), ∀(p: People), (fool p t)
```

- "…and you can fool some of the people all of the time."

```
#check ∀(t: Time), ∃(p: People), (fool p t)
#check ∃(p: People), ∀(t: Time), (fool p t)
```

Universal Quantification Introduction Rule

- Unlike with equality (`eq.refl`) we don't have a simple introduction rule for universal quantification
- Instead we have an algorithm: for a type T and predicate Q that takes an instance of type T, if we want to prove $\forall$(t: T), (Q t), we
  - Assume we have an arbitrary t of type T
  - Prove that from that assumption the predicate Q holds for that arbitrary t

Proof (with tactic notation)

```
example: ∀(P : Prop), ¬(P ∧ ¬P) :=
begin
  assume P,
  assume pfPAndNotP,
  have pfP := pfPAndNotP.1,
  have pfNotP := and.elim_right pfPAndNotP,
  -- pfNotP "maps" a proof of P to false
  exact pfNotP pfP,
end
```

Proof (with lambda notation)

```
example: ∀(P : Prop), ¬(P ∧ ¬P) :=
  -- assume an arbitrary proposition
  λ P: Prop,
    -- assume it's both true and false
    λ h: (P ∧ ¬P),
      -- derive a contradiction
      (h.right h.left : false)
      -- thereby proving ¬ h
```

Proof 2 (with tactic notation)

```
example: ∀(n : nat), n + 1 ≠ 0 :=
begin
  assume n, -- assumes an arbitrary nat, n
  -- assume proof of n + 1 = 0
  assume pfNPlus1IsZero,
  -- show it “maps” to false
  exact (nat.no_confusion pfNPlus1IsZero),
end
```

Proof 2 (with lambda notation)

```
example: ∀(n : nat), n + 1 ≠ 0 :=
  -- assumes an arbitrary nat, n
  λ n : nat,
    -- assume proof of n + 1 = 0
    λ h : (n + 1 = 0),
      -- show it “maps” to false
      (nat.no_confusion h : false)
      -- therefore n + 1 ≠ 0
```

Yet another example

```
 -- Axiom of the Excluded Middle
axiom AoEM: ∀(P: Prop), P ∨ ¬P

example: ∀(n : nat), ∀(m : nat), m = n ∨ m ≠ n :=
begin
  assume n : nat,
  -- the context now includes n
  assume m : nat,
  -- the context now also has m
  cases (AoEM (m = n)) with pfMEqN pfMNotEqN,
    -- case m = n
    exact or.inl pfMEqN,

    -- case m ≠ n
    exact or.inr pfMNotEqN,
end
```

Forall as implication

- What does `∀(p : P), Q` mean?

```
variables P Q : Prop
#check (∀ (p : P), Q)

[Lean] P → Q : Prop
```

- How is this the same as an implication?

```
lemma same : (∀ (p : P), Q) = (P → Q) := rfl
```

Forall as implication (2)

```
variable ap2q : (∀(p : P), Q)
```

- Assume a proof of P.

```
variable p : P
```

- What is the type of (ap2q p)?

```
    #check ap2q p
```

  - Q
- `∀(p: P), Q` is thus a mapping of total functions from P to Q

So, why not use implication?

- Forall lets us name our assumed value

```
#check ∀(n : nat), n = 0 ∨ n ≠ 0.
```

- For some cases, implications make more sense, in other cases we will use forall.
  - If both work, there's not necessarily a "better" way

Nested bindings

- `#check ∀(p: P), (∀(q: Q), R)`

- `#check ∀(n: ℕ), (∀(m: ℕ), m + n >= 0)`

- Nesting bindings will become more useful when we introduce existential quantifiers!
  - Fun question: what does the phrase “you can fool all of the people some of the time and some of the people all of the time” mean?

Universal Quantification Elimination Rule

- As with introduction, there is not a named elimination rule, but rather a simple algorithm
- To use an existing proof (i.e., a fact in the antecedent or “above the turnstile” for a goal), the approach is to *instantiate* the universal quantifier with a specific value
- Remember that the universal quantifier is a *mapping* from an arbitrary instance of the type being quantified over to the body of the universal quantifier

Example Elimination

```
variable Q: ℕ → Prop

example: (∀(n: ℕ), (Q n)) → (Q 3) :=
begin
  assume pfQForallN,
  exact pfQForallN 3
end
```

Fin