Chapter 6

Predicate logic

Predicate logic is the language of Mathematics. Unlike propositional logic which on its own isn't good for anything we can express any mathematical concept in predicate logic. This is why predicate logic has been used to by Peano to present the theory of natural numbers and later by Zermelo and Fraenkel to write down the axioms of set theory. However, from the type-theoretic perspective we don't have to present the formalism of predicate logic but as for propositional logic we just need to explain how we can represent the idioms of predicate logic in Type Theory using the propositions as types translation.

In chapter 3 we introduced propositional logic with the connectives $\_\wedge\_$ (and), $\_\vee\_$ (or), implication $\_\Rightarrow\_$, negation $\neg\_$, logical equivalence $\_\Leftrightarrow\_$ and the logical constants True and False. Now we are going to extend this to predicate logic and we are going to introduce the quantifiers $\forall$ (for all) and $\exists$ (exists).

6.1 Predicates and Quantifiers

What is a predicate? Actually predicate is just another word for property. An example is the predicate $\mathsf{Even} : \mathbb{N} \to \mathsf{prop}$, a predicate is a function from some domain to propositions. Since we assume $\mathsf{prop} = \mathsf{Set}$ a predicate is nothing but a dependent type. So for example $\mathsf{Even}\ 2 : \mathsf{prop}$ is the proposition that 2 is even, while $\mathsf{Even}\ 3 : \mathsf{prop}$ is the proposition that 3 is even. We expect that the former is non-empty, i.e. has evidence, while the should be no evidence for the latter.

Predicates with several inputs are called relations. Using currying we can represent relations without using products $(\_\times\_)$ so for example

$$\_\leqslant\_ : \mathbb{N} \to \mathbb{N} \to \mathsf{prop}$$

is the less or equal relation. This is not limited to binary relations, an example for a ternary once is congruence modulo $\_\equiv\_\mathsf{mod}\_$ e.g. $3 \equiv 7\ \mathsf{mod}\ 2 : \mathsf{prop}$

expresses the (true) proposition that 3 and 7 have the same remainder when divided by 2.

To avoid repetition we just assume as given $\mathsf{A} : \mathsf{Set}$ predicates $\mathsf{P}\ \mathsf{Q} : \mathsf{A} \to \mathsf{prop}$ and a proposition $\mathsf{R} : \mathsf{prop}$.

We need to define the following operations:

$\forall$[x $\in$ A] P x For all x:A , P x holds.

$\exists$[x $\in$ A] P x There is an x : A, such that P x holds.

So for example $\forall[\ \mathsf{x} \in \mathbb{N}\ ]\ \mathsf{Even\ x} \vee \mathsf{Odd\ x}$ expresses that all natural are even or odd, and $\exists[\ \mathsf{x} \in \mathbb{N}\ ]\ \mathsf{Even\ x}$ expresses that there is an even natural number.

What is evidence for $\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$? It is a function that assigns to every $\mathsf{x} : \mathsf{A}$ evidence for $\mathsf{P\ x}$. Hence we use the dependent function type and define

$$\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x} = (\mathsf{x} : \mathsf{A}) \to \mathsf{P\ x}$$

What is evidence for $\exists\mathsf{P}\ \mathsf{x} \in \mathsf{A}\ ,\ \mathsf{P\ x}$? It is a pair of an element $\mathsf{a} : \mathsf{A}$ and evidence for $\mathsf{P\ a}$. Hence we use a dependent pair type or Σ-type to interpret exists:

$$\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x} = \Sigma[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$$

We can use function application to instantiate a universally quantified assumption. That is given $\mathsf{h} : \forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$ and $\mathsf{a} : \mathsf{A}$ we obtain $\mathsf{f\ a} : \mathsf{P\ a}$. To prove a universally quantified proposition $\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$ we assume as given an arbitrary $\mathsf{x} : \mathsf{A}$ and prove $\mathsf{p} : \mathsf{P\ x}$ and now $\lambda\ \mathsf{x} \to \mathsf{p} : \forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$.

To prove an existential statement $\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$ we need to find a witness $\mathsf{a} : \mathsf{A}$ and show that the witness satisfies the property $\mathsf{p} : \mathsf{P\ a}$ then $(\mathsf{a}\ ,\ \mathsf{p}) : \exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$. On the other hand we can show the elimination principle for existential quantification: that is if we assume $\mathsf{h} : \exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x}$ and we can show that a given any $\mathsf{x} : \mathsf{A}$ which that satisfies $\mathsf{p\ x}$ we can prove a proposition R then we know R. This can be expressed as follows:

```
ex-e : (∃[ x ∈ A ] P x) → (∀[ x ∈ A ] (P x ⇒ R)) → R
```

This is easy to prove using pattern matching:

```
ex-e (a , p) f = f a p
```

I noticed that students often struggle with this principle because it is a bit hard to verbalize. Here is my attempt using a concrete example" I use A to be the students in my class, I use P x to mean that x is clever and R means that I m happy. Now we know it is true that if any student is clever then I am happy this corresponds to $\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P\ x} \to \mathsf{R}$.

We can now complete the table showing the propositions as types explanation for all logical connectives:

latin	*english*	*logic*	*types / definitions*
Conjunction	And	$\mathsf{P} \wedge \mathsf{Q}$	$\mathsf{P} \times \mathsf{Q}$
Disjunction	Or	$\mathsf{P} \vee \mathsf{Q}$	$\mathsf{P} \uplus \mathsf{Q}$
Implication	if-then	$\mathsf{P} \Rightarrow \mathsf{Q}$	$\mathsf{P} \rightarrow \mathsf{Q}$
Verum	true	True	$\top$
Falsum	false	False	$\bot$
Negation	not	$\neg\,\mathsf{P}$	$\mathsf{P} \Rightarrow \mathsf{False}$
Equivalence	if-and-only-if	$\mathsf{P} \Leftrightarrow \mathsf{Q}$	$(\mathsf{P} \Rightarrow \mathsf{Q}) \wedge (\mathsf{Q} \Rightarrow \mathsf{P})$
Universal quantification	for all	$\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}$	$(\mathsf{x} : \mathsf{A}) \rightarrow \mathsf{P}\,\mathsf{x}$
Existential quantification	exists	$\exists[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}$	$\Sigma[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}$

Abstract explanations like this are either obvious or incomprehensible. Hence the best is to look at some examples which is what we are going to do in the rest of this chapter.

6.2 Some Tautologies

To illustrate what we can do with our definition of quantifiers, we shall prove some tautologies, that is statements in predicate logic which are true in general.

The first tautology I want to prove is that $\forall$ commutes with $\wedge$, that is

$$\mathsf{taut_1} \ : \ (\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P}\,\mathsf{x} \wedge \mathsf{Q}\,\mathsf{x}) \Leftrightarrow (\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}) \wedge (\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{Q}\,\mathsf{x})$$

I think it is alway helpful to think of a concrete example: so let's say A is the set of students in the lecture, $\mathsf{P}\,\mathsf{x}$ means that student x has yellow laces, and $\mathsf{Q}\,\mathsf{x}$ means that x wears a black shirt. Now it is clearly the same to say *Everybody has yellow laces and wears a black shirt* and *Everybody has yellow laces and everybody wears a black shirt*, and this has nothing to do with students, laces or shirt colors.

Ok two prove the statement we need to prove two implications:

$$\begin{aligned} \mathsf{proj_1}\ \mathsf{taut_1} &= ? \\ \mathsf{proj_2}\ \mathsf{taut_1} &= ? \end{aligned}$$

Each of them is an implication, hence we may as well introduce parameters.

$$\begin{aligned} \mathsf{proj_1}\ \mathsf{taut_1}\ \mathsf{h} &= ? \\ \mathsf{proj_2}\ \mathsf{taut_1}\ \mathsf{h} &= ? \end{aligned}$$

In the first case we have an assumption $\mathsf{h} : \forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P}\,\mathsf{x} \wedge \mathsf{Q}\,\mathsf{x})$, i.e. a dependent function and we need to prove $(\forall\,[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P}\,\mathsf{x}) \wedge (\forall\,[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{Q}\,\mathsf{x})$, that is a pair of functions ? , ?. In the first component we need to prove $\forall[\,\mathsf{x} \in \mathsf{A}\,]\,\mathsf{P}\,\mathsf{x})$ that is a function $\lambda\,\mathsf{x} \rightarrow\ ?$ where $? : \mathsf{P}\,\mathsf{x}$. Now we can derive $\mathsf{h}\,\mathsf{x} : \mathsf{P}\,\mathsf{x} \wedge \mathsf{Q}\,\mathsf{x}$ and hence obtain $\mathsf{proj_1}\ (\mathsf{h}\,\mathsf{x}) : \mathsf{P}\,\mathsf{x}$. Similarly, we use $\lambda\,\mathsf{x} \rightarrow \mathsf{proj_2}\ (\mathsf{h}\,\mathsf{x})$ for the 2nd component.

$$\mathsf{proj}_1\ \mathsf{taut}_1\ \mathsf{h}\ =\ (\lambda\,\mathsf{x}\ \rightarrow\ \mathsf{proj}_1\ (\mathsf{h}\ \mathsf{x}))\ ,\ \lambda\,\mathsf{x}\ \rightarrow\ \mathsf{proj}_2\ (\mathsf{h}\ \mathsf{x})$$

For the other direction

$$\mathsf{proj}_2\ \mathsf{taut}_1\ \mathsf{h}\ =\ ?$$

We have that $\mathsf{h}\ :\ (\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x}) \wedge (\forall\ [\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{Q}\ \mathsf{x})$ and we need to show $?\ :\ \forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x} \wedge \mathsf{Q}\ \mathsf{x}$. We first split our assumption h into two

$$\mathsf{proj}_2\ \mathsf{taut}_1\ (\mathsf{hp}\ ,\ \mathsf{hq})\ =\ ?$$

With $\mathsf{hp}\ :\ \forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x})$ and $\mathsf{hq}\ :\ \forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{Q}\ \mathsf{x}$. Now to prove $?\ :\ \forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x} \wedge \mathsf{Q}\ \mathsf{x}$ we need to derive a function, hence we use $\lambda\,\mathsf{x}\ \rightarrow\ ?$ where $?\ :\ \mathsf{P}\ \mathsf{x} \wedge \mathsf{Q}\ \mathsf{x}$. Hence we need to produce a pair $?\ ,\ ?\ :\ \mathsf{P}\ \mathsf{x} \wedge \mathsf{Q}\ \mathsf{x}$ which we can do using $\mathsf{hp}\ \mathsf{x}\ :\ \mathsf{P}\ \mathsf{x}$ and $\mathsf{hq}\ \mathsf{x}\ :\ \mathsf{Q}\ \mathsf{x}$. Hence we have

$$\mathsf{proj}_2\ \mathsf{taut}_1\ (\mathsf{hp}\ ,\ \mathsf{hq})\ =\ \lambda\,\mathsf{x}\ \rightarrow\ \mathsf{hp}\ \mathsf{x}\ ,\ \mathsf{hq}\ \mathsf{x}$$

What happens if we replace $\_\wedge\_$ by $\_\vee\_$ in the previous statement. Can we prove

$$(\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x} \vee \mathsf{Q}\ \mathsf{x}) \Leftrightarrow (\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x}) \vee (\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{Q}\ \mathsf{x})$$

A quick analysis using common sense shows that this doesn't hold. It maybe that all students have yellow laces or black shirts but this doesn't mean that all students have yellow laces or all students have black shirts. I.e. the left to right implication doesn't hold. However, the right to left one is actually valid and can be proven:

$$\begin{aligned}
&\mathsf{taut}_2\ :\ (\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x}) \vee (\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{Q}\ \mathsf{x}) \Rightarrow (\forall[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x} \vee \mathsf{Q}\ \mathsf{x})\\
&\mathsf{taut}_2\ (\mathsf{inj}_1\ \mathsf{p})\ =\ \lambda\,\mathsf{x}\ \rightarrow\ \mathsf{inj}_1\ (\mathsf{p}\ \mathsf{x})\\
&\mathsf{taut}_2\ (\mathsf{inj}_2\ \mathsf{q})\ =\ \lambda\,\mathsf{x}\ \rightarrow\ \mathsf{inj}_2\ (\mathsf{q}\ \mathsf{x})
\end{aligned}$$

Ok, what happens if we start agan with the first tautology but this time we replace $\forall\mathsf{P}$ with $\exists\mathsf{P}$? Does exists commute with and?

$$(\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x} \wedge \mathsf{Q}\ \mathsf{x}) \Leftrightarrow (\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x}) \wedge (\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{Q}\ \mathsf{x})$$

Again common sense shows that this doesn't hold, it may well be that there is a student who has yellow laces and another one with a black shirt but this doesn't mean that there is one student with yellow laces and a black shirt. This time the implication from right to left fails but we can prove the one from left to right:

$$\begin{aligned}
&\mathsf{taut}_3\ :\ (\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x} \wedge \mathsf{Q}\ \mathsf{x}) \Rightarrow (\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{P}\ \mathsf{x}) \wedge (\exists[\ \mathsf{x} \in \mathsf{A}\ ]\ \mathsf{Q}\ \mathsf{x})\\
&\mathsf{taut}_3\ (\mathsf{a}\ ,\ (\mathsf{p}\ ,\ \mathsf{q}))\ =\ (\mathsf{a}\ ,\ \mathsf{p})\ ,\ (\mathsf{a}\ ,\ \mathsf{q})
\end{aligned}$$

The code shows clearly why we can't turn it around. We use the same a twice but this won't work in the other direction when both witnesses may be different.

What happens if we do both replacements at the same time. Can we prove:

```
(∃[ x ∈ A ] P x ∨ Q x) ⇔ (∃[ x ∈ A ] P x) ∨ (∃[ x ∈ A ] Q x)
```

And indeed we can:

```
taut₄ : (∃[ x ∈ A ] P x ∨ Q x) ⇔ (∃[ x ∈ A ] P x) ∨ (∃[ x ∈ A ] Q x)
proj₁ taut₄ (a , inj₁ p) = inj₁ (a , p)
proj₁ taut₄ (a , inj₂ q) = inj₂ (a , q)
proj₂ taut₄ (inj₁ (a , p)) = a , inj₁ p
proj₂ taut₄ (inj₂ (a , q)) = a , inj₂ q
```

I leave it to you to go through the code line by line.

To summarize we can show:

```
(∀[ x ∈ A ] P x ∧ Q x) ⇔ (∀[ x ∈ A ] P x) ∧ (∀[ x ∈ A ] Q x)
(∃[ x ∈ A ] P x ∨ Q x) ⇔ (∃[ x ∈ A ] P x) ∨ (∃[ x ∈ A ] Q x)
```

what is the pattern? ∀ is represented as a Π-type which is an infinte product and ∧ is represented as × which is binary product. Now it should be no surprise that products commute with products. In the same vain ∃ was represented as a Σ-type which is an infinite sum and ∨ was represented using ⊎ which is a binary sum and again sums commute with sums.

6.3 Equality

While we usually define relations that are specific to a set (e.g. `_⩽_ : ℕ → ℕ → prop`), there is one relation which is used for all sets: equality. We write

```
_≡_ : A → A → prop
```

That is given `a b : A` we write `a ≡ b : prop` for the proposition that a is equal to b. So for example we write `1 + 2 ≡ 1 + 2 : prop` which is actually inhabited, while `0 ≡ 1 : prop` is empty.

Equality is defined as an inductive type with one constructor: reflexivity.

```
data _≡_ : A → A → prop where
  refl : {a : A} → a ≡ a
```

Indeed, this is not fundamentally different to the dependent types we have already seen e.g. Vec and Fin. However, the special thing about equality is that the same index is repeated in the codomain of refl.

We can understand the equality type as saying that identical things are equal. This sounds trivial but before we weren't able to talk about identity. The equality type reflects the notion of identity as a type and hence we can talk about it, i.e. use it in a proposition.

Now that we have defined equality, we should verify that basic properties hold. What are the basic properties of equality? First of all it should be an equivalence relation. A relation _˜_ : A → A → prop is an equivalence relation, if it is reflexive, symmetric and transitive. We can express this using a record: [1]

```
record isEqRel (_˜_ : A → A → prop) : prop where
  field
    reflexive  : {a : A}     → a ∼ a
    symmetric  : {a b : A}   → a ∼ b → b ∼ a
    transitive : {a b c : A} → a ∼ b → b ∼ c → a ∼ c
```

Ok let's show that _≡_ {A} : A → A → prop is an equivalence relation. The first property, reflexivity, is obvious because it is exactly what the constructor refl says. What about symmetry?

```
sym : {a b : A} → a ≡ b → b ≡ a


sym h = ?
```

We have assumed a b : A and h : a ≡ b and our goal is to prove b ≡ a. We are going to use the fact that the only proof of a ≡ b is refl and pattern match on h. We end up with

```
sym refl = ?
```

And now we only have one assumption a : A and our goal is to prove a ≡ a because once we matched h with refl we also need to identify a and b. Now this is easy to complete:

```
sym refl = refl
```

To summarise: we can show that equality is symmetric because the only way we can prove a ≡ b is by refl and in this case a and b are identical and we can use refl to prove b ≡ a which is just a ≡ a.

Transitivity is also straightforward using basically the same idea:

```
trans : {a b c : A} → a ≡ b → b ≡ c → a ≡ c
```

Ok we start with:

```
trans p q = ?
```

where a b c : A, p : a ≡ b and q : b ≡ c and we need to prove a ≡ c. An obvious move is to pattern match one of the equality proofs. But which one? It turns out it doesn't matter. Let's just go with p.

[1] Ok, at this point I am giving up on using the special ∀-syntax which translates into a dependent function type because it is more convenient to use {..} → here because we don't always want to fill in these parameters when using sym or any of the other combinators for equality.

```
trans refl q = ?
```

Matching p with refl has identified a and b. Hence we now only have a c : A and q : a ≡ c and our goal is still a ≡ c which just follow by assumption:

```
trans refl q = q
```

To summarise: we have shown that _≡_ {A} is an equivalence relation:

```
≡-is-eqRel : isEqRel (_≡_ {A})
≡-is-eqRel = record {
  reflexive  = refl;
  symmetric  = sym;
  transitive = trans}
```

There are other properties of equality which are useful in proofs. Every function preserves equality, we say equality is a congruence.

```
cong : (f : A → B) → {a b : A} → a ≡ b → f a ≡ f b
```

Again this is easy to prove by pattern matching a ≡ b:

```
cong f refl = refl
```

At this point we may wonder why we can't prove the inverse of cong

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

However, we cannot pattern match f a ≡ f b. Even though we know that it has to be proven by refl we don't know that the argument would have to be f a, and indeed f may be a constant function and there is no reason why we should assume that both sides need to be f a. Actually uncong states that every function is injective which clearly isn't true.

We can only pattern match an equality assumption if at least one side is a variable which can be identified with the other side — which implies it cannot contain this variable as a subterm.

Another important property of equality is that we can replace equals by equals. That is if we have a goal P b and we know that a ≡ b then we can prove P a instead. This is expressed by the following function:

```
subst : (P : A → prop) → {a b : A} → a ≡ b → P a → P b
```

which again is easy to prove using pattern matching:

```
subst P refl p = p
```

Indeed, all the principles we have proved before can be easily derived from subst. I'll leave this as an exercise.

6.3.1 Propositional vs definitional equality

Sometimes it is mentioned as a criticism about Type Theory that we have two equalities. And indeed, maybe you have noticed that I sometimes use _=_ to talk about equalities which follow directly from the definitions of functions and now we have introduce _≡_ as *propositional equality.* Isn't this rather confusing.

I don't think so. There is only one equality we can talk about *in* Type Theory and this is exactly propositional equality _≡_, which captures precisely what we mean by equality in Mathematics. However, when we talk *about* Type Theory then Type Theory is indeed a bit more involved then conventional logic, in that we also have a static equality, i.e. definitional equality. And we cannot talk about this in Type Theory as we cannot talk about the inhabitance _ : _. Both are called *judgements* and not propositions. They are static properties of the language, so to say rules of the grammar but not the content we are talking about.

Once we accept that a : A is a judgement we also need to be able to talk about a judgemental equality. For example let's say I define

```
n : ℕ
n = 3
```

and later I construct:

```
v : Vec ℕ n
v = 1 :: 2 :: 3 :: []
```

Then to see that the definition of v is well formed, i.e. is statically an acceptable text we need to know statically that n = 3. Which means that equality is a part of the judgement that a certain text is grammatically correct.

Now when I present Type Theory, i.e. in a book about Type Theory or in a lecture I am often switching between two modes: I am talking about Type Theory and then I am talking in Type Theory. Hence I will sometimes mention that certain expressions are definitionally equal using _=_ even though this is nothing I can talk about in Type Theory. And certainly if two expressions are definitionally equal then the values they denote are indeed propositionally equal just using refl.

6.4 Proving properties of addition

Ok, now we have the tools, let's do something with them. Actually let's build more tools. When reasoning about numbers we need to use algebra and what are the most basic laws of algebra? Here are two basic laws about addition on natural numbers: adding zero doesn't change anything. Why two laws? Since we haven't yet proven that x + y = y + x we can add zero on the left 0 + n ≡ n or on the right n + 0 ≡ n.

Before we look at them let's recall the definition of addition from section 4.1.2:

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

Now, let's prove the laws. Starting with adding zero on the left:

```
lneutr+ : {n : ℕ} → 0 + n ≡ n
```

It turns out that this law is trivial, because this follows directly from our definition of _+_. Hence we just need to say

```
lneutr+ = refl
```

and we are done.

The other direction is less straightforward but more interesting.

```
rneutr+ : {n : ℕ} → n + 0 ≡ n
```

We cannot say rneutr+ = refl because the two sides are not identical or compute to the same expression. However, we can still prove it by using recursion. For this purpose first of all lets make the parameter visible

```
rneutr+ {n} = ?
```

Ok, we have assumed n : ℕ what can we do with it? Exactly, we can pattern match on it:

```
rneutr+ {zero} = ?
rneutr+ {suc n} = ?
```

The first goal is zero + zero ≡ zero and this is easy to prove again, because we know from the definition of _+_ that zero + n = n, hence

```
rneutr+ {zero} = refl
```

In the suc n case our goal is suc n + zero ≡ suc n. While here we can't reduce to identical expressions we can compute a bit exploiting the second line in the definition of _+_ which tells us that suc m + n ≡ suc (m + n). Applying this to the left side of our goal it simplifies to suc (n + zero) ≡ suc n. This suggest we could recursively prove n + zero ≡ n but we need to get rid of the pesky suc. But we can, we just need to use cong suc:

```
rneutr+ {suc n} = cong suc ?
```

And now indeed the goal is n + zero ≡ n and we can use a recursive call:

```
rneutr+ {suc n} = cong suc (rneutr+ {n})
```

What has actually happened? We just used recursion in a proof! And it meant that to prove that some properties holds for all natural numbers it is enough to prove it for zero and prove it for suc n using that it holds for n. This proof principle has a name, it is called *the principle of induction.* This is a nice consequence of propositions as types: induction is recursion.

Let's move on. Another very useful property of addition is that it doesn't matter how we bracket several additions, that is:

```
assoc+ : {l m n : ℕ} → (l + m) + n ≡ l + (m + n)
```

This property is called *associativity.* Again our strategy is to use pattern matching and recursion but this time we have three numbers to choose from: l, m and n. It turns out that the first choice l is the right one, but why? Looking at the definition of addition we notice that an expression of the form $e_1 + e_2$ computes if e_1 starts with a constructor. Hence it is a good idea to turn expressions on the left into constructors and l is best positioned to achieve this.

```
assoc+ {zero} = ?
assoc+ {suc l} = ?
```

The first goal is to show (zero + m) + n ≡ zero + (m + n) and using the definition of _+_ we can see

```
(zero + m) + n = m + n
zero + (m + n) = m + n
```

Hence we just need to say:

```
assoc+ {zero} = refl
```

Ok, what about the suc l case? We need to prove (suc l + m) + n ≡ suc l + (m + n). Again we can use the definition of _+_:

```
(suc l + m) + n = suc (l + m) + n
                = suc ((l + m) + n
(suc l) + (m + n) = suc (l + (m + n)
```

Hence all we need to do is to use cong suc and then we can use recursion:

```
assoc+ {suc l} = cong suc (assoc+ {l})
```

In algebra there is a name for an operation (like _+_) and an element which behaves like 0: it is called *a monoid*:

```
record IsMonoid
  (e : A) (_•_ : A → A → A) : prop where
```

```
    field
      lneutr : {x : A} → e • x ≡ x
      rneutr : {x : A} → x • e ≡ x
      assoc : {x y z : A} → (x • y) • z ≡ x • (y • z)
```

and we can summarize our results so far:

```
+isMonoid : IsMonoid zero _+_
+isMonoid = record {
  lneutr = lneutr+;
  rneutr = rneutr+;
  assoc = λ {x} {y} {z} → assoc+ {x} {y} {z}}
```

However, addition is one more useful property: it is commutative m + n ≡ n + m. How can we prove this?

```
comm+ : {m n : ℕ} → m + n ≡ n + m
```

Again the question is over which argument to do the induction (aka recursion). However,due to the symmetry of the equation it doesn't matter - hence let's use the first.

```
comm+ {zero} {n} = ?
comm+ {suc m} {n} = ?
```

We get two goals:

```
zero + n ≡ n + zero
suc m + n ≡ n + suc m
```

In both cases we can simplify the left hand side, but there is no simplification possible for the righthand sides, because they are stuck on n.

```
n ≡ n + zero
suc (m + n) ≡ n + suc m
```

The first goal can be handled with rneutr+ but it is not clear how to make progress with the 2nd. We need a way to move the successor form the 2nd argument to the front - while the definition only provides a way to move the successor from inside the first argument. Hence we suspend this proof and first prove a lemma:

```
suc+ : {m n : ℕ} → suc (n + m) ≡ n + suc m
```

In this case it is clear that the induction should use n. The rest is straight forward:

```
suc+ {m} {zero} = refl
suc+ {m} {suc n} = cong suc (suc+ {m} {n})
```

Back to our original problem:

```
comm+ : {m n : ℕ} → m + n ≡ n + m
```

While as remarked the zero case can be handled with renutr+ there is a slight inconvenience that the equation is the wrong way around: rneutr proves n + zero ≡ n but we need n ≡ n + zero. This can be easily dealt with using sym:

```
comm+ {zero} {n} = sym rneutr+
```

For the second case we have to combine the lemma suc+ we have shown and the recursive call. That is to show

```
suc m + n ≡ n + suc m
```

We use

$$\begin{aligned} \mathsf{suc\ m} + \mathsf{n} &= \mathsf{suc\ (m + n)} && \text{The definition of } + \\ &= \mathsf{suc\ (n + m)} && \text{The induction hypothesis for } \mathsf{m} \\ &= \mathsf{n} + \mathsf{suc\ m} && \text{The lemma } \mathsf{suc+} \end{aligned}$$

Using trans and cong we can translate this into agda code

```
comm+ {suc m} {n} = trans (cong suc (comm+ {m} {n})) suc+
```

It is clear that this presentation of equational proofs will become very unreadable very quickly. The standard agda library defines some syntactic sugar which provides a nicer presentation which shows the intermediate goals and enables us to layout the derivation nicely but which actually reduces to the same proof.

```
comm+ {suc m} {n} = begin
  suc m + n
    ≡⟨ refl ⟩
  suc (m + n)
    ≡⟨ cong suc (comm+ {m} {n}) ⟩
  suc (n + m)
    ≡⟨ suc+ ⟩
  n + suc m
  ∎
```

However, we still have to spend a lot of space for mathematically trivial steps. This is can be addressed using a solver which automatically computes solutions.

To summarise we have shown that _+_ forms a *commutative monoid*:

```
record IsCommMonoid
  (e : A) (_•_ : A → A → A) : prop where
  field
```

```
    isMonoid : IsMonoid e _•_
    isComm : {x y : A} → x • y ≡ y • x


+isCommMonoid : IsCommMonoid zero _+_
+isCommMonoid = record {
  isMonoid = +isMonoid;
  isComm = λ {x} {y} → comm+ {x} {y}}
```

6.5 Extending the negative translation

We have introduced the negative translation in section 3.4 as an alternative to the boolean interpretation of classical logic. Now moving to predicate logic we cannot actually use the boolean interpretation because given P : A → Bool how can we determine a boolean to ∀[x : A] P x. In particular it may be that A is infinite, e.g. A = ℕ.

However, the negative translation which translates any proposition P so that RAA P = ¬ (¬ P) → P holds still works as we will see. First of all we need to show that RAA holds for ∀[x ∈ A] P x if for all x : A we have that RAA (P x) holds.

```
all-neg : ((x : A) → RAA (P x)) → RAA (∀[ x ∈ A ] P x)
all-neg raa h x = raa x (λ g → h (λ k → g (k x)))
```

What about ∃[x ∈ A] P x? Like P ∨ Q this is a positive statement, i.e. it contains information, namely the choice of witness. Hence as for _∨_ we translate ∃ using the classical equivalence that ∃[x ∈ A] P x is equivalent to saying that it is not the case the property isn't false for all x : A.

```
Ex-c : (A : Set) (P : A → prop) → prop
Ex-c A P = ¬ (∀[ x ∈ A ] ¬ (P x))
```

I am going to write $\exists^c$[x ∈ A] P for classical existence. As for _ $\vee^c$ _. It should be clear that we have:

neg-ex-c : RAA ($\exists^c$[x ∈ A] P)

because we have already shown that all negated propositions satisfy RAA.

We need to show that the basic principles for existentials are provable, that is we need to show:

ex^c−i : (a : A) → P a → ∃c[x ∈ A] P x
ex^c−e : RAA R → ((x : A) → P x → R) → $\exists^c$[x ∈ A] P x → R

What about equality? In turns out that most equalities we encounter are already negative, for example _≡_ {ℕ} satisfies RAA because it is decidable as we a re going to show in the next chapter. However, we don't need to assume this but we can define classical equality as the double negation of the usual equality:

$$\begin{array}{l} \_ \equiv^c \_ \; : \; \mathsf{A} \to \mathsf{A} \to \mathsf{prop} \\ \mathsf{a} \equiv^c \mathsf{b} \; = \; \neg\, (\neg\, (\mathsf{a} \equiv \mathsf{b})) \end{array}$$

and as before we have:

$$\mathsf{ne\text{-}eq\text{-}c} \; : \; \{\mathsf{a}\ \mathsf{b} \; : \; \mathsf{A}\} \to \mathsf{RAA}\ (\mathsf{a} \equiv^c \mathsf{b})$$

We can derive some the basic principles associated with equality for $\_ \equiv^c \_$:

$$\begin{array}{l} \mathrm{refl}^c \; : \; \{\mathsf{a} \; : \; \mathsf{A}\} \to \mathsf{a} \equiv\text{-}\mathsf{c}\ \mathsf{a} \\ \mathrm{subst}^c \; : \; (\mathsf{P} \; : \; \mathsf{A} \to \mathsf{prop}) \to (\{\mathsf{x} \; : \; \mathsf{A}\} \to \mathsf{RAA}\ (\mathsf{P}\ \mathsf{x})) \\ \quad \to \{\mathsf{a}\ \mathsf{b} \; : \; \mathsf{A}\} \to \mathsf{a} \equiv\text{-}\mathsf{c}\ \mathsf{b} \to \mathsf{P}\ \mathsf{a} \to \mathsf{P}\ \mathsf{b} \end{array}$$

I am leaving the verification of these properties as an exercise.

Hence it seems that classical logic is just the logic of the negative fragment of intuitionistic logic where we use negative, i.e. classical versions, of disjunction and existence. Almost, but there is another important aspect of classical Mathematics: the axiom of choice which we are going to discuss later. The negative translation is not going to help us here because the negative translation of the axiom of choice is not provable intuitionistically.

6.6 History

An early form of predicate logic was introduced by Frege in the late 19th century [Fre79]. His basic laws allowed quantification over all sets and hence was inconsistent as Russell pointed out. However, his attempt to make the rules of reasoning precise lead to the formal system of predicate logic. This is also often called first order logic, because we don't allow quantification over propositions themselves.

While Frege tried to capture set theory and logic at the same time, later it became clear that it is a good idea to separate the system of reasoning and the particular basic assumptions, i.e. axioms. Hence for example Peano Arithmetic and Zermelo-Fraenkel set theory can be formulated in the same framework. It was Hilbert and Ackermann who clearly formulated first order logic in the 1930ies, (there is a recent english translation: [HA99]).

Gödel proved a completeness theorem for classical predicate logic which shows that the formal system is complete for the classical notion of a model, i.e. if a proposition holds in all models it is also provable [Kur29]. However, this is done for classical logic in a classical metatheory and in this form it is not relevant in our context.

Unlike set theory type theory doesn't use the framework of predicate logic it stands on its own. However, many of the idioms of predicate logic are used and we can understand them via their translation into predicate logic as we have shown. However, the interpretation of disjunction ∨ and existence ∃ we have represented are debatable sine the types don't really behave like propositions. We will return to this issue later and present an alternative view what is a proposition.

6.7 Exercises

1. We are playing logic poker again. But this time with predicate logic!

   Consider the following propositions:

   ```
   P01 = {A B : Set} {R : A → B → prop} →
     ((∀[ x ∈ A ] ∃[ y ∈ B ] R x y)) →
     (∃[ y ∈ B ] ∀[ x ∈ A ] R x y)
   P02 = {A B : Set} {R : A → B → prop} →
     (∃[ y ∈ B ] (∀[ x ∈ A ] R x y))
      → (∀[ x ∈ A ] (∃[ y ∈ B ] R x y))
   P03 = {A : Set} {P : A → prop} →
     ¬ (∃[ x ∈ A ] P x) → ∀[ x ∈ A ] ¬ (P x)
   P04 = {A : Set} {P : A → prop} →
     (∀[ x ∈ A ] ¬ (P x)) → ¬ (∃[ x ∈  A ] P x)
   P05 = {A : Set} {P : A → prop} →
     (¬ (∀[ x ∈ A ] P x)) → ∃[ x ∈ A ] ¬ (P x)
   P06 = {A : Set} {P : A → prop} →
     (∃[ x ∈ A ] ¬ (P x)) → (¬ (∀[ x ∈ A ] P x))
   P07 = {A : Set} {P : A → prop} →
     (¬ (¬ (∀[ x ∈ A ] P x))) → ∀[ x ∈ A ] ¬ (¬ (P x))
   P08 = {A : Set} {P : A → prop} →
     (∀[ x ∈ A ] ¬ (¬ (P x))) → (¬ (¬ (∀[ x ∈ A ] P x)))
   P09 = {A : Set} {P : A → prop} →
     (∃[ x ∈ A ] ⊤) → (∃[ x ∈ A ] P x) → ∀[ x ∈ A ] P x
   P10 = {A : Set} {P : A → prop} →
     (∃[ x ∈ A ] ⊤) → (∃[ y ∈ A ] (P y → ∀[ x ∈ A ] P x))
   ```

   Here is a quick summary of the rules of logic poker:

   (a) You can prove it directly (intuitionistically) - just provide pi : Pi.

   (b) You cannot prove it with intuitionistic logic but you can prove it with classical logic. In this case prove pi : CLASS → Pi where

   ```
   CLASS = {P : prop} → RAA P
   ```

   (c) It isn't even true in classical logic. In this case prove pi : Pi → ⊥. Note that you will have to come up with predicates and relations to refute the proposition.

2. Show that all the properties of equality we have shown are derivable from subst. That is prove

   ```
   sym-s : {a b : A} → a ≡ b → b ≡ a
   trans-s : {a b c : A} → a ≡ b → b ≡ c → a ≡ c
   cong-s : (f : A → B) → {a b : A} → a ≡ b → f a ≡ f b
   ```

   using only:

   ```
   subst : (P : A → prop) → {a b : A} → a ≡ b → P a → P b
   ```

   instead of pattern matching.

3. Verify the laws of boolean algebra we discussed in section 3.1.

   ```
   com-& : {x y : Bool} → x & y ≡ y & x
   distr-&| : {x y z : Bool} → x & (y | z) ≡ x & y | x & z
   dm1 : {x y : Bool} → ! (x | y) ≡ ! x & ! y
   dm2 : {x y : Bool} → ! (x & y) ≡ ! x | ! y
   ```

4. Show that the natural numbers ℕ with addition and multiplication form a commutative semiring. We have already shown that addition forms a commutative monoid, hence it is left to show:

   ```
   *isCommMonoid : IsCommMonoid 1 _*_
   null-l : {n : ℕ} → 0 * n ≡ 0
   distr-l : {l m n : ℕ} → (l + m) * n ≡ l * n + m * n
   ```

   Note that I have mitted the symmetric forms of the distributivity laws:

   ```
   null-r : {n : ℕ} → n * 0 ≡ 0
   distr-r : {l m n : ℕ} → l * (m + n) ≡ l * m + l * n
   ```

   since they follow from commutativity but it may be useful to prove them before establishing commutativity.

5. Continuing from the previous question: show that the natural numbers form an exponential semi-ring, that is prove the laws of high school algebra:

   ```
   exp-plus : {l m n : ℕ} → l ↑ (m + n) ≡ l ↑ m * l ↑ n
   exp-times : {l m n : ℕ} → l ↑ (m * n) ≡ (l ↑ m) ↑ n
   ```

6. Prove the missing lemmas about the negative translation from section 6.5.

   ```
   all-neg : ((x : A) → RAA (P x)) → RAA (∀[ x ∈ A ] P x)
   ex^c−i : (a : A) → P a → ∃^c[ x ∈ A ] P x
   ex^c−e : RAA R → ((x : A) → P x → R) → ∃^c[ x ∈ A ] P x → R
   refl^c : {a : A} → a ≡^c a
   subst^c : (P : A → prop) → ({x : A} → RAA (P x))
      → {a b : A} → a ≡^c b → P a → P b
   ```