Contents

1 Kinds of induction

1.1 Stepwise

Stepwise induction, or more commonly understood as just mathematical induction refers to a proof technique used to establish the truth of an infinite sequence of statements, typically indexed by natural numbers. In plain english its based on two main steps

1. Base case: Prove that the statement holds for some initial condition.
2. Inductive step: Prove that if the statement holds for some arbitrary subsequent case then the following case also holds.

Example 1.1. Dominos
A nice example out of the book 'Proofs' by Jay Cummings is to imagine a line of dominos standing upright. To prove that all the dominos will fall, we can use stepwise induction:

- Base case: Prove that the first domino falls when pushed.
- Inductive step: Prove that if any domino k falls, then the next domino $k+1$ will also fall.

By establishing these two steps, we can conclude that all the dominos in the line will eventually fall.

We can express this more formally as follows

Definition 1.1. Stepwise induction
If we consider a sequence of mathematical statements $P(n)$ indexed by natural numbers $n \in \mathbb{N}$, then to prove that $P(n)$ holds for all natural numbers n, we need to show:

1. Base case: $P(0)$ is true.
2. Inductive step: For every $k \in \mathbb{N}$, if $P(k)$ is true, then $P(k+1)$ is also true.

If both conditions are satisfied, then by the principle of mathematical induction, we can conclude that $P(n)$ holds for all natural numbers n. An equivalent compressed formulation of this principle can be expressed as follows

$$(P(0) \land \forall k \in \mathbb{N},\ (P(k) \to P(k+1))) \to \forall n \in \mathbb{N},\ P(n) \qquad \text{(Induction Principle)}$$

Example 1.2. Peano Arithmetic
One canonical example where we see the principle of stepwise induction appear is as an axiom in Peano arithmetic, which is a foundational system for the natural numbers. In Peano arithmetic, the principle of mathematical induction is used to define properties of natural numbers and to prove statements about them. Here we define the signature for the theory of Peano arithmetic as follows

$$\Sigma_{PA} : \{0, 1, +, \times, =\}$$

with the following axioms

$$\forall x.\neg(0 = x + 1) : 0 = \text{MIN}(\mathbb{N}) \qquad (\textit{zero})$$
$$\forall x.x + 0 = x \qquad (\textit{plus zero})$$
$$\forall x.\forall y.x + (y + 1) = (x + y) + 1 \qquad (\textit{plus successor})$$
$$\forall x.x \times 0 = 0 \qquad (\textit{times zero})$$
$$\forall x.\forall y.x \times (y + 1) = (x \times y) + x \qquad (\textit{times successor})$$
$$\forall x.\forall y.(x = y) \to (x + 1 = y + 1) \qquad (\textit{equality})$$
$$(F(0) \land (\forall x.F(x) \to F(x + 1))) \to (\forall x.F(x)) \qquad (\textit{induction})$$

Example 1.3. Sum of $n \in \mathbb{N}$
Lets try to prove the following statement

$$P(n) : \sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

We can prove this using stepwise induction as follows

- Base case: Since we are summing from 1 to n, we start with $n = 1$. We need to show that $P(1)$ holds:

$$P(1) : \sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2} = 1$$

  Thus, the base case holds.

- Inductive step: Firstly here we are provided with the inductive hypothesis, namely that $P(k)$ holds for some arbitrary $k \in \mathbb{N}$, so we assume that the following holds

$$P(k) : \sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

  Our goal is to show that $P(k+1)$ also holds, which means we need to prove

$$P(k+1) : \sum_{i=1}^{k+1} i = \underbrace{1 + 2 + \ldots + k}_{= \frac{k(k+1)}{2} \text{ via IH}} + (k+1) = \frac{(k+1)(k+2)}{2}$$

  as already annotated in the above expression we know that up to k the sum holds via the inductive hypothesis, so we can substitute that in and simplify as follows

$$\begin{aligned}
\sum_{i=1}^{k+1} i &= \frac{k(k+1)}{2} + (k+1) && \text{(by IH)} \\
&= \frac{k(k+1) + 2(k+1)}{2} && \left(\tfrac{(x+y*z)}{y} = \tfrac{x}{y} + z\right) \\
&= \frac{(k+1)(k+2)}{2} && \text{(factoring out } (k+1)\text{)}
\end{aligned}$$

  Thus the inductive step holds.

We can construct this equivalent proof in Lean as follows

Example 1.4. Sum of n in Lean As we are now working in a proof assistant we need to first define how we encode the notion of summation. We will be using the `Finset` library which provides us with the ability to describe a finite set of distinct elements. In particular we will be using the `Finset.sum` function which allows us to sum over a finite set. So the basic dynamics are

```
def s : Finset ℕ := {1, 2, 3}
#eval Σ i ∈ s, i -- evaluates to 6
```

As a reminder what we want to prove is that

$$\forall n \in \mathbb{N}, \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

We can encode this in Lean as follows

```
theorem sum_n (n : ℕ) : Σ i ∈ Finset.range (n + 1), i = n * (n + 1) / 2 := ...
```

where `Finset.range (n + 1)` generates the finite set $\{0, 1, 2, \ldots, n\}$ so we will need to adjust our summation to start from 1 instead of 0. The general structure of any inductive proof on natural numbers is always

```
induction n with
| zero => ...
| succ k ih => ...
```

where in the `zero` case we prove the base case and in the `succ k ih` case we prove the inductive step with `ih` being the inductive hypothesis. So our full proof looks like this

- Base case: Here we can rely on simplification to show that both sides evaluate to 1

```
induction n with
| zero => simp
```

- Inductive step: The inductive case becomes more interesting. Something we sort of take for granted when doing pen-and-paper proofs is the ability to just freely manipulate expressions algebraically. In Lean however we need to be a bit more explicit about this. Our full proof looks as follows

```
| succ k ih =>
calc
    (Σ i ∈ Finset.range ((k + 1) + 1), i) -- P(k+1)
      = (Σ_n(k)) + (k + 1)                := by rw [Finset.sum_range_succ]
    _ = k * (k + 1) / 2 + (k + 1)         := by rw [ih]
    _ = (k * (k + 1) + 2 * (k + 1)) / 2   := by rw [Nat.add_mul_div_left _ _ (by norm_num)]
    _ = ((k + 1)) * ((k + 1) + 1) / 2     := by ring_nf
```

Here we use the `calc` block which allows us to chain together a series of equalities. Breaking down the lemmas we use each step

1. `Finset.sum_range_succ`: This can be expressed as
$$\sum_{i=0}^{n} i = \left(\sum_{i=0}^{n-1} i\right) + n$$
so in our case it preforms the rewrite
$$\sum_{i=0}^{k+1} i = \left(\sum_{i=0}^{k} i\right) + (k+1)$$
2. `ih`: Here we simply apply the inductive hypothesis to rewrite $\sum_{i=0}^{k} i$ as $\frac{k(k+1)}{2}$ in full our induction hypothesis states that
3. `Nat.add_mul_div_left`: This lemma states that for natural numbers a, b, c with $c \neq 0$, the following holds
$$\frac{a}{c} + b = \frac{a + b \cdot c}{c}$$
so in our case we have the rewrite
$$\frac{k(k+1)}{2} + (k+1) = \frac{k(k+1) + 2(k+1)}{2}$$
4. `ring_nf`: This tactic is used to normalize expressions in a ring, which includes simplifying polynomial expressions. In our case it helps us factor out $(k+1)$ from the numerator to get the final desired form
$$\frac{k(k+1) + 2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$$

1.2 Strong / Complete

We can define strong or complete induction intuitively by going back to our example of dominos. In stepwise induction the idea was that if we could demonstrate that the first domino falls and that each domino at position k causes the next domino at position $k+1$ to fall, then all the dominos would eventually fall. In complete or strong induction we modify the inductive step to be a bit stronger. Instead of just showing that domino k causes domino $k+1$ to fall, we show that if all dominos up to position k have fallen, then domino $k+1$ will also fall. This means that the state of all previous dominos influences the next one, rather than just the immediate predecessor.

Definition 1.2. Complete induction
We can once again introduce an inductive schema

$$(\forall n.\ (\forall k < n \to P(k)) \to P(n)) \to (\forall n.\ P(n)) \qquad \text{(Complete Induction Principle)}$$

One thing we can immediately observe here is that we don't have a base case. Reason for this being that if we instantiate n to be 0 we get

$$(\forall k < 0 \to P(k)) \to P(0) \equiv (\text{vacuous}) \to P(0) \equiv P(0)$$

so the base case is just implicit in the inductive schema. Within lean we see this defined as follows

```
@[elab_as_elim]
protected theorem strong_induction_on {p : ℕ → Prop} (n : ℕ)
    (h : ∀ n, (∀ m, m < n → p m) → p n) : p n :=
Nat.strongRecOn n h
```

Example 1.5. Strong induction in T^*_{PA}
We introduce some additional axioms into Peano arithemtic

$$\forall x, y.\ x < y \to \text{quot}(x, y) = 0 \qquad \text{(\textit{less than quotient})}$$
$$\forall x, y.\ y > 0 \to \text{quot}(x + y, y) = \text{quot}(x, y) + 1 \qquad \text{(\textit{quotient successor})}$$
$$\forall x, y.\ x < y \to \text{rem}(x, y) = x \qquad \text{(\textit{less than remainder})}$$
$$\forall x, y.\ y > 0 \to \text{rem}(x + y, y) = \text{rem}(x, y) \qquad \text{(\textit{remainder successor})}$$

Some important properties we have about these axioms are

$$\forall x, y.\ y > 0 \to \text{rem}(x, y) < y \qquad \text{(Remainder less than divisor)}$$

this property suggests that we apply strong induction to prove

$$P(x) : \forall y.\ y > 0 \to \text{rem}(x, y) < x$$

thus for the IH we can assume that for an arbitrary $x \in \mathbb{N}$

$$\forall k.\ k < x \to \underbrace{\forall y.\ y > 0 \to \text{rem}(k, y) < y}_{P(k)} \qquad \text{(IH)}$$

and we need to show that

$$\forall y.\ y > 0 \to \text{rem}(x, y) < y$$

So we proceed by case analysis on whether $x < y$ or not

- Case $x < y$ - here we have

$$\begin{aligned} x < y &\to \text{rem}(x, y) = x && \text{(by \textit{less than remainder})} \\ &\to \text{rem}(x, y) < y && \text{(since } x < y\text{)} \end{aligned}$$

- Case $x \geqslant y$ - here we must have $n \in \mathbb{N}$ s.t. $n < x \land x = n + y$. Thus we can derive

$$\begin{aligned} \text{rem}(x, y) &= \text{rem}(n + y, y) && \text{(since } x = n + y\text{)} \\ &= \text{rem}(n, y) && \text{(by } \textit{remainder successor}\text{)} \\ &< y && \text{(by IH since } n < x\text{)} \end{aligned}$$

Another important property is

$$\forall x, y.\ y > 0 \to x = \text{quot}(x, y) \times y + \text{rem}(x, y) \qquad \text{(Division Algorithm)}$$

this suggests that we apply complete induction to prove

$$G(x) : \forall y.\ y > 0 \to x = \text{quot}(x, y) \times y + \text{rem}(x, y)$$

thus for the IH we can assume that for an arbitrary $x \in \mathbb{N}$

$$\forall k.\ k < x \to \underbrace{\forall y.\ y > 0 \to k = \text{quot}(k, y) \times y + \text{rem}(k, y)}_{G(k)} \qquad \text{(IH)}$$

again we proceed by case analysis on whether $x < y$ or not

- Case $x < y$ - here we have

$$\begin{aligned} x < y &\to \text{quot}(x, y) = 0 && \text{(by } \textit{less than quotient}\text{)} \\ &\to x = 0 * y + \text{rem}(x, y) && \text{(by rearranging)} \end{aligned}$$

- Case $x \geqslant y$ - here we must have $n \in \mathbb{N}$ s.t. $n < x \land x = n + y$. Thus we can derive

$$\begin{aligned} \text{quot}(x, y) \times y + \text{rem}(x, y) &= y \times \text{quot}(n + y, y) + \text{rem}(n + y, y) && \text{(since } x = n + y\text{)} \\ &= y \times (\text{quot}(n, y) + 1) + \text{rem}(n, y) && \text{(by } \textit{quot and rem successor}\text{)} \\ &= y + (y \times \text{quot}(n, y) + \text{rem}(n, y)) && \text{(rearranging)} \\ &= y + n && \text{(by IH since } n < x\text{)} \\ &= x && \text{(since } x = n + y\text{)} \end{aligned}$$

1.3 Well-founded

We start by defining what a well-founded relation is

Definition 1.3. Well-founded relation
A binary relation or predicate $\prec$ over a set S is a *well-founded relation* iff there does not exist an infinite sequence of elements such that each successive element is less than its predecessor according to the relation $\prec$. Formally this can be expressed as follows

$$\neg \exists x_0, x_1, x_2, \ldots \in S.\ x_1 \prec x_0 \land x_2 \prec x_1 \land x_3 \prec x_2 \land \ldots$$

in other words each sequence of elements of S that decreases according to the relation $\prec$ must eventually terminate i.e. is finite.

Example 1.6. $<$ over $\mathbb{N}$
A classic example of a well-founded relation is the standard less-than relation $<$ over the natural numbers $\mathbb{N}$. To see why this is well-founded, consider any sequence of natural numbers $n_0, n_1, n_2, \ldots$ such that each successive number is less than its predecessor according to the relation $<$. That is, we have

$$n_1 < n_0, \quad n_2 < n_1, \quad n_3 < n_2, \quad \ldots$$

Since the natural numbers are well-ordered and have a minimum element (which is 0), any decreasing sequence must eventually reach this minimum element.

Example 1.7. $<$ over $\mathbb{Q}$
However not all less-than relations are well-founded. For instance, consider the less-than relation $<$ over the rational numbers $\mathbb{Q}$. In this case, we can construct an infinite decreasing sequence of rational numbers. For example, consider the sequence

$$1, \quad \frac{1}{2}, \quad \frac{1}{4}, \quad \frac{1}{8}, \quad \dots$$

Here, each successive number is less than its predecessor according to the relation $<$. This sequence does not terminate and continues indefinitely, demonstrating that the less-than relation over the rational numbers is not well-founded.

Example 1.8. Theory of $\mathsf{T}^{\mathrm{PA}}_{\mathsf{cons}}$
The theory of $\mathsf{T}^{\mathrm{PA}}_{\mathsf{cons}}$ combines the theory of lists with Peano arithmetic. It includes the axioms of both theories in addition to the following

$$\forall \mathtt{atom}\ u, v.\ u \preceq_c v \leftrightarrow u = v \tag{$\preceq_c$ 1}$$
$$\forall \mathtt{atom}\ u, v.\ \forall v.\ \neg \mathtt{atom}(v) \to \neg(v \preceq_c u) \tag{$\preceq_c$ 2}$$
$$\forall \mathtt{atom}\ u, v.\ \forall v, w.\ u \preceq_c \mathtt{cons}(v, w) \leftrightarrow u = v \vee u \preceq_c w \tag{$\preceq_c$ 3}$$
$$\forall \mathtt{atom}\ u_1, v_1, u_2, v_2.\ \mathtt{cons}(u_1, v_1) \preceq_c \mathtt{cons}(u_2, v_2)$$
$$\leftrightarrow (u_1 = u_2 \wedge v_1 \preceq_c v_2) \vee \mathtt{cons}(u_1, v_1) \preceq_c v_2 \tag{$\preceq_c$ 4}$$
$$\forall x, y.\ x \prec_c y \to x \preceq_c y \wedge x \neq y \tag{$\prec_c$}$$
$$\forall \mathtt{atom}\ u.\ |u| = 1 \tag{*Atom length*}$$
$$\forall u, v.\ |\mathtt{cons}(u, v)| = |v| + 1 \tag{*List length*}$$

The first four axioms here define the *sublist relation* $\preceq_c$ which relates two lists if the first is a sublist of the second. The fifth axiom defines the *strict sublist relation* $\prec_c$ in terms of $\preceq_c$. The last two axioms define the length function $|\cdot|$ for lists, stating that the length of an atom is 1 and the length of a cons cell is one plus the length of its tail.

The strict sublist relation $\prec_c$ is well-founded. To see why, consider any sequence of lists $L_0, L_1, L_2, \dots$ such that each successive list is a strict sublist of its predecessor according to the relation $\prec_c$. That is, we have

$$L_1 \prec_c L_0, \quad L_2 \prec_c L_1, \quad L_3 \prec_c L_2, \quad \dots$$

Since each list in the sequence is a strict sublist of the previous one, the length of the lists must decrease with each step. Specifically, we have

$$|L_1| < |L_0|, \quad |L_2| < |L_1|, \quad |L_3| < |L_2|, \quad \dots$$

However, the length of a list is a natural number, and natural numbers are well-founded under the standard less-than relation. Therefore, the lengths of the lists in the sequence must eventually reach 0, at which point the sequence must terminate. This shows that there cannot be an infinite descending sequence of lists under the strict sublist relation $\prec_c$, demonstrating that $\prec_c$ is well-founded.

Well-founded induction is a *generalization* of complete induction in which we replace the less-than relation with an arbitrary well-founded relation. So it generalizes the notion of complete induction to an arbitrary theory T by allowing us to define induction over any well-founded relation $\prec$ defined within that theory.

Definition 1.4. Well-founded induction
Formally we can express the well-founded induction principle as follows

$$(\forall n.\ (\forall k.\ k \prec n \to P(k)) \to P(n)) \to (\forall n.\ P(n)) \qquad \text{(Well-founded Induction Principle)}$$

for some Σ-formulae $P(n)$ with one free variable n. So the IH we apply here is that for some arbitrary n and for all k such that $k \prec n$ $P(k)$ holds, we need to show that $P(n)$ also holds, i.e. is T-valid.

Example 1.9. Applying well-founded induction

Lets consider proving the following property in T^{PA}_{cons}

$$\forall x.\ |x| \geqslant 1$$

we apply well-founded induction on x using the relation $\prec_s$ to prove

$$P(x) : |x| \geqslant 1$$

for the IH we can assume that

$$\forall k.\ k \prec_s x \to \underbrace{|k| \geqslant 1}_{P(k)} \tag{IH}$$

we proceed by case analysis on whether x is an atom or not

- Case `atom` - here we have

  $$|x| = 1 \qquad \text{(by } \textit{Atom length}\text{)}$$

  thus $|x| \geqslant 1$ holds.

- Case $\neg$`atom` - here we must have u, v s.t. $x = \mathsf{cons}(u, v)$. Thus we can derive

  $$\begin{aligned} |x| &= |\mathsf{cons}(u, v)| && \text{(since } x = \mathsf{cons}(u, v)\text{)} \\ &= |v| + 1 && \text{(by } \textit{List length}\text{)} \\ &\geqslant 1 && \text{(since } |v| \geqslant 1 \text{ by IH as } v \prec_s x\text{)} \end{aligned}$$

  thus $|x| \geqslant 1$ holds.

1.4 Structural

Definition 1.5. Strict subformula relation
Given a formula ϕ in a formal language, a *strict subformula* of ϕ is any formula that is a proper part of ϕ, meaning it is contained within ϕ but is not equal to ϕ itself. The strict subformula relation, denoted by $\prec_{sf}$, relates two formulas ψ and ϕ such that $\psi \prec_{sf} \phi$ if and only if ψ is a strict subformula of ϕ. Formally, we can define the strict subformula relation as follows:

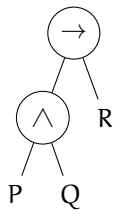
$$\psi \prec_{sf} \phi \iff \psi \text{ is a strict subformula of } \phi$$

This relation captures the hierarchical structure of formulas, where larger formulas are composed of smaller subformulas.

Example 1.10. Subformulas
Lets consider the FOL formula

$$F = (P \wedge Q) \to R$$

The syntax tree for this formula looks as follows



We can see here quite nicely that for F we have the following strict subformulas

$$P, Q, R, (P \wedge Q)$$

Definition 1.6. Structural induction
Similar to how complete induction is just well-founded induction instantiated on the less-than relation, structural induction is well-founded induction instantiated on the strict subformula relation. Thus we can express the structural induction principle as follows

$$(\forall \phi.\ (\forall \psi.\ \psi \prec_{sf} \phi \to P(\psi)) \to P(\phi)) \to (\forall \phi.\ P(\phi)) \qquad \text{(Structural Induction Principle)}$$

We state the inductive hypothesis (IH) here as follows: for some arbitrary formula ϕ, and for all formulas ψ such that ψ is a strict subformula of ϕ (i.e., $\psi \prec_{sf} \phi$), we assume that $P(\psi)$ holds. Our goal is then to show that $P(\phi)$ also holds, i.e., that the property P is valid for the formula ϕ.

Example 1.11. Equivalence of formulae
A classical example here is the notion that every propositional formula F is equivalent to one constructed only with the logical connectives $\{\top, \wedge, \neg\}$. We can prove this using structural induction on the formula F as follows

- Base case: We have 3 base cases two consider
  - $F = \top$ - here we are done as $\top$ is already in the desired form.
  - $F = P$ where P is an atomic proposition - here we are also done as atomic propositions are allowed in the desired form.
  - $F = \bot$ - here we can use the equivalence $\bot \equiv \neg\top$ to rewrite $\bot$ in the desired form.
- Inductive step: Here we consider the possible structures of F based on its main connective

  More specifically we consider 3 formulas G, G_1, G_2 and we consider via the IH that they are all equivalent to some formulas G', G_1', G_2' constructed only with $\{\top, \wedge, \neg\}$. We aim to show that each possible formula constructed only with one possible connective is equivalent to some formula constructed only with $\{\top, \wedge, \neg\}$.

  - $F = \neg G$ - by the IH we know that G is equivalent to some formula G' constructed only with $\{\top, \wedge, \neg\}$. Thus we have
    $$F = \neg G \equiv \neg G'$$
    which is also in the desired form.
  - $F = G_1 \wedge G_2$ - by the IH we know that both G_1 and G_2 are equivalent to some formulas G_1' and G_2' constructed only with $\{\top, \wedge, \neg\}$. Thus we have
    $$F = G_1 \wedge G_2 \equiv G_1' \wedge G_2'$$
    which is also in the desired form.
  - $F = G_1 \vee G_2$ - by the IH we know that both G_1 and G_2 are equivalent to some formulas G_1' and G_2' constructed only with $\{\top, \wedge, \neg\}$. We can use De Morgan's law to rewrite F as follows
    $$F = G_1 \vee G_2 \equiv \neg(\neg G_1 \wedge \neg G_2) \equiv \neg(\neg G_1' \wedge \neg G_2')$$
    which is also in the desired form.
  - $F = G_1 \to G_2$ - by the IH we know that both G_1 and G_2 are equivalent to some formulas G_1' and G_2' constructed only with $\{\top, \wedge, \neg\}$. We can use the equivalence $A \to B \equiv \neg A \vee B$ to rewrite F as follows
    $$\begin{aligned} F = G_1 \to G_2 &\equiv \neg G_1 \vee G_2 \\ &\equiv \neg G_1' \vee G_2' \\ &\equiv \neg(\neg\neg G_1' \wedge \neg G_2') \end{aligned}$$
    which is also in the desired form.

2 Recursors

Recursors can most succinctly be described as functions that allow us to define operations on inductively defined data types by specifying how to handle each constructor of the data type. They are closely related to the concept of induction, as they provide a way to define functions that operate on all elements of an inductively defined set by recursively applying the function to the components of each element.

2.1 On $\mathbb{N}$

We can define a simple grammar and operational semantics for recursors over the natural numbers as follows

$$\begin{aligned} \text{Types} \quad & \sigma, \tau ::= \tau_1 \to \tau_2 \mid \mathsf{nat} \\ \text{Expressions} \quad & e ::= x \mid \lambda x : \tau.\, e \mid e_1\ e_2 \mid \mathsf{zero} \mid \mathsf{succ}(e) \mid \mathsf{rec}_{\mathsf{nat}}[\tau](e, e_1, \mathsf{v.res}.(e_2)) \end{aligned}$$

We can define the typing rules as follows

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

$$\frac{}{(\lambda x : \tau.\ e_1)\ e_2 \to e_1[x \mapsto e_2]}\ \beta\textsc{-reduction} \qquad \frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}}\ \textsc{Zero} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(e) : \mathsf{nat}}\ \textsc{Succ}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \mathsf{nat} \\ \Gamma \vdash e_1 : \tau \\ \Gamma, \mathsf{v} : \mathsf{nat}, \mathsf{res} : \tau \vdash e_2 : \tau\end{array}}{\Gamma \vdash \mathsf{rec}_{\mathsf{nat}}[\tau](e, e_1, \mathsf{v.res.}(e_2)) : \tau}\ \textsc{NatRec}$$

We can define the operational semantics of recursion on natural numbers as follows

$$\frac{e \to e'}{\mathsf{rec}_{\mathsf{nat}}[\tau](e, e_1, \mathsf{v.res.}(e_2)) \to \mathsf{rec}_{\mathsf{nat}}[\tau](e', e_1, \mathsf{v.res.}(e_2))}\ \textsc{ERec} \qquad \frac{}{\mathsf{rec}_{\mathsf{nat}}[\tau](\mathsf{zero}, e_1, \mathsf{v.res.}(e_2)) \to e_1}\ \textsc{RecZero}$$

$$\frac{}{\mathsf{rec}_{\mathsf{nat}}[\tau](\mathsf{succ}(e), e_1, \mathsf{v.res.}(e_2)) \to e_2[\mathsf{v} \mapsto e][\mathsf{res} \mapsto \mathsf{rec}_{\mathsf{nat}}[\tau](e, e_1, \mathsf{v.res.}(e_2))]}\ \textsc{RecSucc}$$

Now lets try and define a simple function which just computes the double of a natural number using our recursor

$$\mathsf{double} = \lambda n : \mathsf{nat}.\ \mathsf{rec}_{\mathsf{nat}}[\mathsf{nat}](-, -, -)$$

Our recursor argument breaks down as follows

- First argument: Here we just pass in n as that is the natural number we want to double.
- Second argument: This corresponds to the base case when $n = 0$, so we just return 0 here as double of 0 is 0.
- Third argument: This corresponds to the inductive step when $n = \mathsf{succ}(k)$ for some k. Here we have two variables available to us - v which represents k and res which represents the result of doubling k. Thus to compute double of $\mathsf{succ}(k)$ we can just return $\mathsf{succ}(\mathsf{succ}(\mathsf{res}))$ as this effectively adds 2 to the double of k.

Thus our full definition of double looks as follows

$$\mathsf{double} = \lambda n : \mathsf{nat}.\ \mathsf{rec}_{\mathsf{nat}}[\mathsf{nat}](n, \mathsf{zero}, \mathsf{v.res.}(\mathsf{succ}(\mathsf{succ}(\mathsf{res}))))$$

Example 2.1. Double function
Lets evaluate the expression $\mathsf{double}\ (\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))$ which corresponds to doubling 2. We proceed as follows

$$\begin{aligned}
\mathsf{double}\ (\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))) &\to \mathsf{rec}_{\mathsf{nat}}[\mathsf{nat}](\mathsf{succ}(\mathsf{succ}(\mathsf{zero})), \mathsf{zero}, \mathsf{v.res.}(\mathsf{succ}(\mathsf{succ}(\mathsf{res})))) && \text{(unfolding double)} \\
&\to \mathsf{succ}(\mathsf{succ}(\mathsf{rec}_{\mathsf{nat}}[\mathsf{nat}](\mathsf{succ}(\mathsf{zero}), \mathsf{zero}, \mathsf{v.res.}(\mathsf{succ}(\mathsf{succ}(\mathsf{res})))))) && \text{(by RecSucc)} \\
&\to \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))) && \text{(simplifying)}
\end{aligned}$$

Thus we see that $\mathsf{double}\ 2$ evaluates to 4

A simple version of how we might define the recursor for natural numbers in lean is as follows

```
def recNat  {τ : Type} (e : Nat) (e₁ : τ) (e₂ : Nat → τ → τ) : τ :=
  match e with
  | Nat.zero    => e₁
  | Nat.succ v  => (e₂ v (recNat v e₁ e₂))
```

What we can see here is that we define a function `recNat` which takes in three arguments - a natural number `e`, a base case value `e1` of type τ, and an inductive step function `e2` which takes in a natural number and a value of type τ and returns a value of type τ. We then pattern match on the natural number `e` to handle the base case when it is zero and

the inductive case when it is a successor.

Something you might have noticed here is that this definition is very similar to the semantics of the inductive schema we defined earlier for natural numbers.

$$(\forall n.\ P(0) \land (\forall k.\ P(k) \to P(k+1))) \to (\forall n.\ P(n))$$

The connection here being that

- Base case $P(0)$ corresponds to the base case value `e1` in our recursor.
- Inductive step $\forall k.\ P(k) \to P(k+1)$ corresponds to the inductive step function `e2` in our recursor.

The distinction here being that the inductive schema is used to prove properties about all natural numbers, whereas the recursor is used to define functions that operate on natural numbers. So the recursor represents the computational form of induction. Another way to think about it is that it describes how to define a function out of an inductive type. We can see this more clearly when examining the actual recursor for naturals in lean

```
Nat.rec.{u} {motive : ℕ → Sort u}
    (zero : motive Nat.zero)                        -- P(0)
    (succ : (n : ℕ) → motive n → motive n.succ) -- ∀k. P(k) → P(k + 1)
    (t : ℕ) : motive t                               -- ∀n. P(n)
```

here we can again define the double function out of this recursor as follows

```
#eval Nat.rec -- outputs 6
  (motive := λ _ => Nat)                            -- function returns Nat
  (zero := 0)                                       -- base case double(0) = 0
  (succ := λ _ rec => Nat.succ (Nat.succ rec)) 3    -- inductive step double(n+1) = double(n) + 2
```

An important thing to understand here is the idea of what the motive function represents. For the case of the natural number recursor the motive describes what we want to show about each natural number. When the motive is non-dependent i.e. it describes a plain recursor returning a type like `nat`, it means we want to define a function that maps each natural number to a natural number. However when the motive is dependent i.e. it describes a recursor returning a type that depends on the input natural number, it means we want to define a property or predicate that holds for each natural number. Which is precisely what we do when we use induction to prove properties about natural numbers. So for example we can define

$$P(n) : n + 0 = n$$

in the following two equivalent ways, first using our classical induction on the structure of natural numbers

```
theorem add_zero : (n : Nat) → n + 0 = n := by
  intro n
  induction n with
   zero => rfl
   succ k ih =>
    calc
      Nat.succ k + 0 = Nat.succ (k + 0) := by rfl
      _ = Nat.succ k                    := by rw [ih]
```

Or equivalently we can define our motive to be

$$\texttt{motive} := \lambda n : \mathbb{N}.\ n + 0 = n$$

in other words for each natural number n we want to show that the property $n + 0 = n$ holds. Thus we can define our proof using the recursor as follows

```
theorem add_zero : (n : Nat) → n + 0 = n :=
  Nat.rec
    (motive := λ n => n + 0 = n)  -- for each n we want to show n + 0 = n
    (zero := by rfl)              -- base case 0 + 0 = 0
    (succ := λ _ ih => by simp)   -- inductive step n + 0 = n implies (n+1) + 0 = n + 1
```

As recursors are just the computational form of induction, we can also define recursors for other inductively defined data types such as lists, trees, etc.

Example 2.2. List recursors
A list is inductively defined as either being empty or being a cons cell containing a head element and a tail list. In lean this is encoded as

```
inductive List (α : Type u) where
| nil : List α
| cons (head : α) (tail : List α) : List α
```

The recursors for lists in lean can be defined as follows

```
List.rec.{u_1, u} {α : Type u}
  {motive : List α → Sort u_1}
  (nil : motive [])                    -- base case P(nil)
  (cons : (head : α) → (tail : List α)
    → motive tail
    → motive (head :: tail))           -- inductive step ∀ head tail. P(tail) → P(head :: tail)
  (t : List α) : motive t              -- ∀ l. P(l)
```

Similar as before if we wanted to then have a function which for example reverses a list we could either define it using the classical functional approach

```
def reverse {α : Type} (l : List α) : List α :=
  match l with
   [] => []
   h :: t => reverse' t ++ [h]
```

or we can define it using the recursor as follows

```
def reverse {α : Type} (l : List α) : List α :=
  List.rec
    (motive := λ _ => List α)
    (nil := [])
    (cons := λ h _ rec => rec ++ [h]) l
```

And again if we make the motive dependent we can prove properties about lists using the recursor. For example we can prove that for any list l its length is always greater than or equal to 0 as follows

```
theorem list_len_geq_zero {α : Type} (l : List α) : l.length ≥ 0 :=
  List.rec
    (motive := λ _ => l.length ≥ 0)
    (nil := by simp)
    (cons := λ _ _ ih => by simp [ih]) l
```

3 Lean Notes

A subtlety regarding induction proofs is cases when your goal depends on *other variables* that in turn depend on the variable you are inducting over. To explain what that means lets consider the following example

```
lemma add_sub_cancel' (x n : ℕ) (h : x ⩽ n) : x + (n - x) = n := by
  induction n generalizing x
  | zero       => cases h; simp
  | succ n' ih =>
    cases x with
    | zero =>
      simp
    | succ x' =>
      simp [Nat.succ_add x' (n' - x')]          -- (a + 1) + b = (a + b) + 1
      apply (ih x' (Nat.le_of_succ_le_succ h))  -- apply the induction hypothesis
```

One aside here on conventions, in Lean you have alot of ways of essentially doing the same thing, the proof above is probably on the more verbose side, once nice thing to use here is the `rcases` strategy.

A noticable thing is that the `zero` constructor has no actual variables and for the `succ` constructor we really only care about the predecessor value `x'`. So a more concise way of doing the case analysis on `x` would be to use `rcases` as follows

```
rcases x with (_ | x')
```

here the underscore _ indicates that we don't care about the variable for the `zero` case, and for the `succ` case we name the predecessor `x'`. The general pattern for an inductive type with multiple constructors is

```
inductive T where
| C1 (a1 : τ₁) ... (am : τₘ)
| C2 (b1 : σ₁) ... (bn : σₙ)
| C3 : τ
...

-- in some proof
rcases t with (a1 ... am | b1 ... bn | _ | ...)
```

Something we can notice here is that the property we are trying to prove now has two variables x and n, now if we just do regular induction our base case is fine, so this works

```
induction n with
| zero       => cases h; simp
```

but when we consider the inductive step we run into a problem. We can rewrite our expression as before

```
| succ x' =>
  simp [Nat.succ_add x' (n' - x')]  -- (a + 1) + b = (a + b) + 1
```

but now the key step. We can notice our goal is aligned with what we are trying to prove, namely

```
x' + (n' - x') = n'
```

but then we see that our induction hypothesis looks like this

```
x' + 1 ⩽ n' → x' + 1 + (n' - (x' + 1)) = n'
```

which isn't very usable, now we could try changing the induction variable to x instead and swap the case analysis but that would give us

```
n' ⩽ x' + 1 → n' + (x' + 1 - n') = x' + 1
```

but again we run into the issue that our inductive hypothesis appears to be misaligned with our goal. In an abstract sense our IH should be something which takes in our hypothesis for the inductive case and returns our goal, but in both cases it appears to only partially do so.

The solution to this is to use the `generalizing` keyword when performing induction. What this does is it tells lean to generalize over the variable x when performing induction on n. This effectively means that lean will treat x as an arbitrary variable that can take on any value, rather than being fixed to a specific value. Thus when we perform induction on n, lean will generate an induction hypothesis that is valid for all possible values of x. This allows us to align our IH with our goal properly as follows

```
induction n generalizing x
```

Now our IH looks like this

```
∀ x' : ℕ, x' ⩽ n' → x' + (n' - x') = n'
```

which is exactly what we want. Thus we can now apply our IH properly in the inductive step as follows

```
apply (ih x' (Nat.le_of_succ_le_succ h))  -- le_of_succ_le_succ does x' + 1 ⩽ n' + 1 to x' ⩽ n'
```