A Formal Verification of Reversible Primitive Permutations

Giacomo Maletto

1. The definition

1.1 Reversible computing

Reversible computing is a model of computation in which every process can be run backwards. Simply put, in a reversible setting any program takes inputs and gives outputs (like usual), but can also go the other way around: provided the output it can reconstruct the input. In a mathematical sense, every function is expected to be invertible.

Why do we care about such a thing?

Firstly, having a programming language in which every function (or even a subset of functions) is reversible could lead to interesting and practical applications.

But we can also imagine reversible computers, in which the underlying architecture is inherently reversible: Toffoli gates provide a way to do so. The opposite of reversibility is loss of information, which (for thermodynamic reasons) leads to loss of energy and heat dissipation. This means that a non-reversible gate dissipates energy each time information is discarded, while in principle a reversible computer wouldn't.

Lastly, reversible computing is directly related to quantum computing, as each operation in a quantum computer must be reversible.

1.2 Reversible Primitive Permutations

In the article we decided to formalize, the authors focus on providing a functional model of reversible computation. They develop an inductively defined set of functions, called **Reversible Primitive Permutations** or **RPP**, which are expressive enough to represent all Primitive Recursive Functions - that is to say, RPP is PRF-complete (we talk about what this means in section ?). Here is the definition that we will use:

Definition 1 (Reversible Primitive Permutations)**.** The class of **Reversible Primitive Permutations** or RPP is the smallest subset of functions $\mathbb{Z}^n \to \mathbb{Z}^n$ satisfying the following conditions:

- The n-ary **identity** $\mathsf{Id}_n(x_1, \ldots, x_n) = (x_1, \ldots, x_n)$ belongs to RPP, for all $n \in \mathbb{N}$.

$$\begin{array}{c} x_1 \\ \vdots \\ x_n \end{array} \;\boxed{\mathsf{Id}_n}\; \begin{array}{c} x_1 \\ \vdots \\ x_n \end{array}$$

  The meaning of these diagrams should be fairly obvious: if the values on the left of a function are provided as inputs to that function, we get the values on the right as outputs.

- The **sign-change** $\mathsf{Ne}(x) = -x$ belongs to RPP.

$$x \;\boxed{\mathsf{Ne}}\; -x$$

- The **successor function** $\mathsf{Su}(x) = x + 1$ belongs to RPP.

$$x \;\boxed{\mathsf{Su}}\; x+1$$

- The **predecessor function** $\mathsf{Pr}(x) = x - 1$ belongs to RPP.

$$x \;\boxed{\mathsf{Pr}}\; x-1$$

- The **swap** $\mathsf{Sw}(x, y) = (y, x)$ belongs to RPP.

$$\begin{array}{c} x \\ y \end{array} \;\boxed{\mathsf{Sw}}\; \begin{array}{c} y \\ x \end{array}$$

- If $f : \mathbb{Z}^n \to \mathbb{Z}^n$ and $g : \mathbb{Z}^n \to \mathbb{Z}^n$ belongs to RPP, then the **series composition** $(f \fatsemi g) : \mathbb{Z}^n \to \mathbb{Z}^n$ belongs to RPP and is such that:

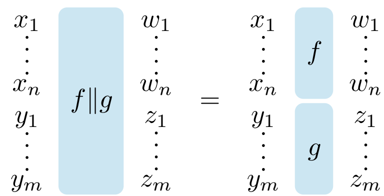
$$(f \fatsemi g)(x_1, \ldots, x_n) = g(f(x_1, \ldots, x_n)) = (g \circ f)(x_1, \ldots, x_n).$$

  We remark that $f \fatsemi g$ means that f is applied first, and then g, in opposition to the standard functional composition (denoted by $\circ$).

$$\begin{array}{c} x_1 \\ \vdots \\ x_n \end{array} \;\boxed{f \fatsemi g}\; \begin{array}{c} z_1 \\ \vdots \\ z_n \end{array} \;=\; \begin{array}{c} x_1 \\ \vdots \\ x_n \end{array} \;\boxed{f}\; \begin{array}{c} y_1 \\ \vdots \\ y_n \end{array} \;\boxed{g}\; \begin{array}{c} z_1 \\ \vdots \\ z_n \end{array}$$

- If $f : \mathbb{Z}^n \to \mathbb{Z}^n$ and $g : \mathbb{Z}^m \to \mathbb{Z}^m$ belongs to RPP, then the **parallel composition** $(f \| g) : \mathbb{Z}^{n+m} \to \mathbb{Z}^{n+m}$ belongs to RPP and is such that:

$$(f \| g)(x_1, \ldots, x_n, y_1, \ldots, y_m) = (f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)).$$



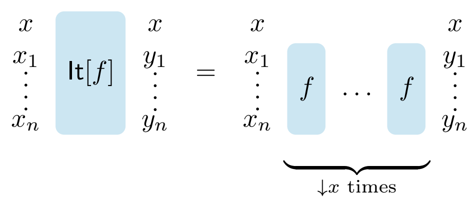
- If $f : \mathbb{Z}^n \to \mathbb{Z}^n$ belongs to RPP, then then **finite iteration** $\mathsf{It}[f] : \mathbb{Z}^{n+1} \to \mathbb{Z}^{n+1}$ belongs to RPP and is such that:

$$\mathsf{It}[f](x, x_1, \ldots, x_n) = (x, (\overbrace{f \circ \cdots \circ f}^{\downarrow x \text{ times}})(x_1, \ldots, x_n))$$

  where $\downarrow (\cdot) : \mathbb{Z} \to \mathbb{N}$ is defined as

$$\downarrow x = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}.$$

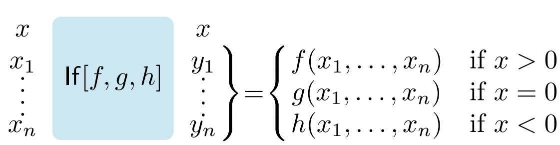
  This means that the function f is applied $\downarrow x$ times to $(x_1, \ldots, x_n)$.



- If $f, g, h : \mathbb{Z}^n \to \mathbb{Z}^n$ belong to RPP, then the **selection** $\mathsf{If}[f, g, h] : \mathbb{Z}^{n+1} \to \mathbb{Z}^{n+1}$ belongs to RPP and is such that:

$$\mathsf{If}[f, g, h](x, x_1, \ldots, x_n) = \begin{cases} (x, f(x_1, \ldots, x_n)), & \text{if } x > 0 \\ (x, g(x_1, \ldots, x_n)), & \text{if } x = 0 \\ (x, h(x_1, \ldots, x_n)), & \text{if } x < 0 \end{cases}.$$

  We remark that the argument x which determines which among f, g and h must be used cannot be among the arguments of f, g and h, as that would break reversibility.



Remark 1. If we have two functions of different arity, for example $f : \mathbb{Z}^3 \to \mathbb{Z}^3$ and $g : \mathbb{Z}^5 \to \mathbb{Z}^5$, then we will still write $f \fatsemi g$ to mean the function with arity $\max(3, 5) = 5$ given by $(f \| \mathsf{Id}_2) \fatsemi g$. In general, the arity of the "smaller" function can be enlarged by a suitable parallel composition with the identity. The same goes for the arguments of the selection $\mathsf{If}[f, g, h]$.

$$\begin{array}{ccc} x_1 & & z_1 \\ x_2 & & z_2 \\ x_3 & \boxed{f \,\mathbin{⨾}\, g} & z_3 \\ x_4 & & z_4 \\ x_5 & & z_5 \end{array} \;=\; \begin{array}{ccccc} x_1 & & y_1 & & z_1 \\ x_2 & \boxed{f} & y_2 & & z_2 \\ x_3 & & y_3 & \boxed{g} & z_3 \\ x_4 & & x_4 & & z_4 \\ x_5 & & x_5 & & z_5 \end{array} \;=\; \begin{array}{ccccc} x_1 & & y_1 & & z_1 \\ x_2 & \boxed{f} & y_2 & & z_2 \\ x_3 & & y_3 & \boxed{g} & z_3 \\ x_4 & \boxed{\mathsf{Id}_2} & x_4 & & z_4 \\ x_5 & & x_5 & & z_5 \end{array}$$

1.3 Some examples

In order to get accustomed to this definition, let's see some examples.

Increment and decrement Let's try to imagine what addition should look like in RPP. Of course, addition is usually thought of as a function which takes two inputs and yields their sum: something like $\mathtt{add}(x, y) = x + y$. But notice that this operation is not reversible: given only the output (the value $x + y$) it is impossible to obtain the original values (x, y). As we will see, every function in RPP is reversible, so we will not be able to define addition in this way.

Instead, we can define a function inc in RPP which, given $n \in \mathbb{N}$ and $x \in \mathbb{Z}$, yields

$$\begin{array}{ccc} n & \boxed{\mathsf{inc}} & n \\ x & & x + n \end{array}$$

If n is negative the output is just (n, x). The fact that the above diagram is only valid for $n \in \mathbb{N}$ might bother some of you; we'll explain later why it is so, and how we can also make it work for $n \in \mathbb{Z}$.

For now let's focus on the output: we don't just have $x + n$ but also n, and indeed, given both n and $x + n$ we can reconstruct n (obviously) and x (by $(x + n) - n$). As a matter of fact, the following function dec also belongs to RPP:

$$\begin{array}{ccc} n & \boxed{\mathsf{dec}} & n \\ x & & x - n \end{array}$$

and if we try to compose inc and dec we get this remarkable result:

$$\begin{array}{ccccc} n & \boxed{\mathsf{inc}} & n & \boxed{\mathsf{dec}} & n \\ x & & x + n & & x \end{array}$$

and similarly for $\mathsf{dec} \mathbin{⨾} \mathsf{inc}$. So indeed dec is the inverse of inc, and we can write $\mathsf{dec} = \mathsf{inc}^{-1}$.

But we haven't said how to actually define inc. Well, just like this:

$$\mathsf{inc} = \mathsf{It}[\mathsf{Su}]$$

This means that we apply the successor function Su to the value x, for $\downarrow n$ times. If $n \in \mathbb{N}$ then $\downarrow n = n$, so we effectively add n to the value x. If instead n is negative then $\downarrow n = 0$ and nothing changes.

Can you guess how dec is defined?

In a very similar manner, using the predecessor function:

$$\mathsf{dec} = \mathsf{It}[\mathsf{Pr}]$$

and as we will shortly see, finding the inverse is not something that we have to do by hand.

Multiplication and square We now turn our attention to multiplication. The elementary-school way to define multiplication is by repeated addition, and we can define mul exactly like that:

$$\mathsf{mul} = \mathsf{It}[\mathsf{inc}].$$

As inc had arity 2, mul has arity $2 + 1 = 3$. If $n, m \in \mathbb{N}$ and $x \in \mathbb{Z}$ then we have

$$\begin{array}{c} n \\ m \\ x \end{array} \boxed{\mathsf{mul}} \begin{array}{c} n \\ m \\ x + n \cdot m \end{array}$$

because we're essentially "incrementing by m" n times; so in this case we preserve both inputs and increase a certain variable x.

What is the inverse mul^{-1}? Does it perform division? Well, the truth is rather disappointing:

$$\begin{array}{c} n \\ m \\ x \end{array} \boxed{\mathsf{mul}^{-1}} \begin{array}{c} n \\ m \\ x - n \cdot m \end{array}$$

We will see a way to calculate division in RPP, but this is not it.

We're now ready to define the function square which is used to calculate the square of a number:

$$\mathsf{square} = (\mathsf{Id}_1 \| \mathsf{Sw}) \mathbin{;} \mathsf{inc} \mathbin{;} \mathsf{mul} \mathbin{;} \mathsf{dec} \mathbin{;} (\mathsf{Id}_1 \| \mathsf{Sw}).$$

That might look like a very complicated expression; thankfully we can make use of diagrams to show what each step does. Given $n \in \mathbb{N}$ and $x \in \mathbb{Z}$ we have

$$\begin{array}{ccccccccccc} n & & n & \boxed{\mathsf{inc}} & n & & n & \boxed{\mathsf{dec}} & n & & n \\ x & \boxed{\mathsf{Sw}} & 0 & & n & \boxed{\mathsf{mul}} & n & & 0 & \boxed{\mathsf{Sw}} & x + n \cdot n \\ 0 & & x & & x & & x + n \cdot n & & x + n \cdot n & & 0 \end{array}$$

so we add the result $n \cdot n$ to a variable x; we also require an additional value initialized to 0. We will make frequent use of variables initially set to 0 and which come back to 0 after the calculation; these are traditionally called **ancillary arguments** or **ancillaes**, from the latin term used to describe female house slaves in ancient Rome.

You might be wondering what would happen if $n < 0$ or the ancilla was different from 0. The truth is, we don't really care. We will often specify the behaviour of these functions given some initial values, and we won't need to know what happens for different initial values because we'll never use those functions in other ways.

1.4 Calculating the inverse

Earlier we hinted at the fact that every function in RPP is invertible and the inverse belongs to RPP; furthermore, we don't need to perform the calculation manually, case by case. In other words, there is an *effective procedure* which produces the inverse $f^{-1} \in \mathsf{RPP}$ given any $f \in \mathsf{RPP}$.

Proposition 1 (The inverse f^{-1} of any f)**.** *Let $f : \mathbb{Z}^n \to \mathbb{Z}^n$ belong to* RPP*. Then the inverse $f^{-1} : \mathbb{Z}^n \to \mathbb{Z}^n$ exists, belongs to* RPP *and, by definition, is*

- $\mathsf{Id}_n^{-1} = \mathsf{Id}_n$
- $\mathsf{Ne}^{-1} = \mathsf{Ne}$
- $\mathsf{Su}^{-1} = \mathsf{Pr}$
- $\mathsf{Pr}^{-1} = \mathsf{Su}$
- $\mathsf{Sw}^{-1} = \mathsf{Sw}$
- $(f \fatsemi g)^{-1} = g^{-1} \fatsemi f^{-1}$
- $(f \| g)^{-1} = f^{-1} \| g^{-1}$
- $\mathsf{It}[f]^{-1} = \mathsf{It}[f^{-1}]$
- $\mathsf{If}[f, g, h]^{-1} = \mathsf{If}[f^{-1}, g^{-1}, h^{-1}]$

Then $f \fatsemi f^{-1} = \mathsf{Id}_n$ and $f^{-1} \fatsemi f = \mathsf{Id}_n$.

Proof. By induction on the definition of f. □

Well, that was rather succint.

We invite the reader to check that every listed inverse does indeed make sense; for example, the function $\mathsf{It}[f](x, y_1, \ldots, y_n)$ applies $\downarrow x$ times the function f to the argument $(y_1, \ldots, y_n)$. If we want to "undo" this effect we just need to apply $\downarrow x$ times f^{-1} to the same argument, so $\mathsf{It}[f]^{-1} = \mathsf{It}[f^{-1}]$.

Of course, that reasoning only works if in turn f is also invertible and $f^{-1} \in \mathsf{RPP}$. This is the reason that the proof is by induction: given an arbitrary RPP, if we unfold one step of the definition we get one of the cases listed. We apply the appropriate step and then by inductive hypothesis we can assume that in turn its sub-terms are invertible.

Notice that in this way, if we try to calculate inc^{-1} we really do get dec, as we had hoped for.

Since RPP is inductively defined, any proposition involving RPP functions can be proven using induction. Not only that, but any function which has for an argument a generic RPP can be defined recursively, and indeed we can also see $(\cdot)^{-1} : \mathsf{RPP} \to \mathsf{RPP}$ as a recursive function. Now that we delve into the Lean theorem prover we will see that induction and recursion can be seen as really the same thing, and that's just one of many similarities between functions and proofs.

1.5 First steps with Lean

In this section we take a look at some of Lean's basic features. You don't have to understand every detail - just enough to have a vague sense of what it's like to define stuff in Lean.

In Lean we primarily do three things:

1. define data structures

2. define functions

3. prove theorems about data structures and functions

What sets Lean apart from your average functional programming language (like Haskell) is the third item on the list. Now we will instead focus on the first and second points.

You don't have to understand every detail of what will follow - a vague understanding of what's going on would be sufficient. The curious reader can run and play with most of the following snippets of code in the online editor `https://leanprover-community.github.io/lean-web-editor/`.

A simple example of a type Data is defined using the `inductive` keyword. Here is the typical example of data structure:

```
inductive weekday : Type
| monday : weekday
| tuesday : weekday
| wednesday : weekday
| thursday : weekday
| friday : weekday
| saturday : weekday
| sunday : weekday
```

This defines a *type* called `weekday`. Days of the week like `monday`, `tuesday`, etc. are elements of the type `weekday`. We can see the type of an element by using the `#check` command:

```
-- opening the scope weekday (otherwise to refer
-- to an element - for example tuesday - of weekday
-- we have to write weekday.tuesday)
open weekday

#check tuesday -- this outputs "weekday"
```

and we write this as

```
tuesday : weekday
```

Everything in Lean has a type (and only one). For example, natural numbers have type ℕ:

```
#check 3 -- ℕ
```

Even type themselves have a type[1]. Lean's type system is very expressive, and makes it possible to work with complex math in Lean.

We can define functions over the type `weekday` - for example, the function `next`:

```
-- Special characters like → will abound.
-- In VS Code and the Lean Web Editor,
-- arrows can be inserted by typing \to and hitting
-- the space bar.
def next : weekday → weekday
| monday    := tuesday
| tuesday   := wednesday
| wednesday := thursday
| thursday  := friday
| friday    := saturday
| saturday  := sunday
| sunday    := monday

#reduce next wednesday -- this outputs "thursday"
#check next -- next has type "weekday → weekday"
```

This function is defined by cases: if we have `monday`, output `tuesday`, if we have `tuesday`, output `wednesday`, and so on.

(Almost) every expression - like `next (next thursday)` or `3 * 5 + 2` - have a corresponding *reduced form* (respectively `saturday` and `17`) which can be displayed using the `#reduce` command, and is obtained by repeatedly applying functions to their arguments, until the full computation is carried out. In this sense, things like `next wednesday` and `next (next tuesday)` (or `2 + 2` and `1 + 3`) are *definitionally equivalent*, because they're reduced to the same expression.

An important remark on notation: in math it is customary to call functions by enclosing arguments in parenthesis and separating them with commas, i.e. $f(x, y, z)$. Languages like Lean follow a different convention: the arguments are simply written after the function name, like `f x y z`. So in our case, what we would write as $\mathsf{next}(\mathsf{next}(\mathsf{thursday}))$ is instead written `next (next thursday)`

[1] Types in Lean have a role similar to sets in math. Standard math axioms (like ZFC) dictates that everything is a set, including sets themselves. This basic notion can lead to some contradictory statements, like the famous Russell's paradox (let's consider the set of all sets that do not contain themselves; does this set contain itself?) and if one is not careful in defining types of types, the same thing could happen with type theory. But in fact, type theory was invented in the beginning of the 20th century by Bertrand Russell precisely to avoid Russell's paradox. The approach used in Lean is to define a cumulative hierarchy of universes `Type : Type 1 : Type 2 ...`, so that it's impossible to invoke objects like "the type of all types" or a type having itself as an element.

(writing `next next thursday` would be wrong because it would mean that the first argument to `next` is the function `next` itself, not `next thursday`). This leads to no ambiguity and often helps reducing clutter.

An example of an inductive type Right now you could be wondering why we used the keyword `inductive` to define `weekday`, when there's *clearly* no induction going on at all in its construction. First of all, it depends on what you mean by induction; but it is true that that was a particularly simple case. As an example of a more overtly inductive object, we can define the natural numbers like this:

```
inductive Nat : Type
| Zero : Nat
| Succ (n : Nat) : Nat
```

The name `Nat` and subsequent objects are capitalized in order to avoid conflict with the definition of `nat` already present in Lean. We can read this definition as "every element of the type `Nat` is either `Zero` or `Succ n` where `n : Nat`", which is basically the Peano definition of natural numbers. Some examples of elements of this type:

```
open Nat

-- all these outputs "Nat"
#check Zero -- represents 0
#check Succ Zero -- represents 1
#check Succ (Succ Zero) -- represents 2
#check Succ (Succ (Succ Zero)) -- represent 3
#check Zero.Succ.Succ.Succ -- also represents 3
                           -- alternative notation
```

Functions over `Nat` have the possibility of being truly recursive: for example, we can recursively define addition `Add m n` by induction over `n`.

- if `n = Zero` then `Add m Zero = m`
- if `n = Succ n'` for some `n' : Nat`,
  then `Add m (Succ n') = Succ (Add m n')`.

Note that by definition each element of `Nat` can be either `Zero` or `Succ n'` for some `n' : Nat`, so the two cases considered cover all possibilities. Written in Lean,

```
def Add : Nat → Nat → Nat
| m Zero      := m
| m (Succ n') := Succ (Add m n')
```

It may have struck you that the type of `Add` is not `Nat × Nat → Nat` but instead `Nat → Nat → Nat`. This is known as currying, and it's not as strange as it might look like at first. Consider this: we can think of `Add` as a function which takes a pair `(m,n) : Nat × Nat` and outputs `Add m n : Nat`, as is standard in mathematics. But we can also think of it as a function which takes just `m : Nat` and outputs the function `Add m : Nat → Nat`, which in turn given `n : Nat` outputs `Add m n : Nat`. We can think of `Add m` as a partially applied function, which becomes fully applied when it is given another argument `n`. From this point of view, `Add` is a function of type `Nat → (Nat → Nat)` which is the same as `Nat → Nat → Nat` because in Lean the arrow → is right associative.

In a certain sense, currying makes functions conceptually simpler; all functions are single variable, it's just that some return other functions.

Integers and lists Functions belonging to RPP have $\mathbb{Z}^n$ as their domain and codomain, so we need a way to represent and work with integer tuples.

The good news is that integers are already defined in Lean. Here is their definition:

```
inductive int : Type
| of_nat (n : ℕ) : int
| neg_succ_of_nat (n : ℕ) : int
```

The value `of_nat n` represents the natural number `n : ℕ` as an integer, while `neg_succ_of_nat n` represents the negative number `-(n+1)`. Of course it's not the definition we've just seen that gives this meaning to the `int`s; rather, it's the functions defined on them (like addition, subtraction etc.).

Immediately after the definition, some notation is introduced:

- ℤ stands for `int`
- in a context in which an integer is expected, if instead a natural number is supplied, the function `of_nat` will be automatically applied on the natural number. This convenient feature is called coercion.
- for every `n : ℕ`, `-[1+ n]` stands for `neg_succ_of_nat n`. This notation is almost never used.

```
notation `ℤ` := int

instance : has_coe nat int := ⟨int.of_nat⟩

notation `-[1+ ` n `]` := int.neg_succ_of_nat n
```

As an example of a function from ℤ to ℤ, this is negation:

```
def neg : ℤ → ℤ
| (of_nat n) := neg_of_nat n
| -[1+ n]    := succ n
```

We're interested not just in integers, but in tuples of integers. We can implement the concept of a tuple in many ways, but a particularly simple one is through the use of lists, a very common data structure in computer science.

Let's consider lists of natural numbers, i.e. the type `list ℕ`. This is a list of 5 elements:

```
open list
#reduce [4, 5, 7, 2, 5] -- [4, 5, 7, 2, 5]
```

The first element of a list is the *head*

```
#reduce head [4, 5, 7, 2, 5] -- 4
```

while the other elements are the *tail*.

```
#reduce tail [4, 5, 7, 2, 5] -- [5, 7, 2, 5]
```

Given `n : ℕ` and `l : list ℕ` we can obtain a new list `cons n l` (also written as `n :: l`) such that `head (n :: l) = n` and `head (n :: l) = l`

```
#reduce cons 2 [4, 5, 7, 2, 5] -- [2, 4, 5, 7, 2, 5]
#reduce 2 :: [4, 5, 7, 2, 5]   -- alternative notation
```

and ultimately, every list can be obtained by starting with `nil`, the empty list, and repeatedly using `cons`.

```
#reduce nil -- the empty list
#reduce []  -- alternative notation
#reduce cons 4 (cons 5 (cons 7 (cons 2 (cons 5 nil))))
            -- [4, 5, 7, 2, 5]
#reduce 4 :: 5 :: 7 :: 2 :: 5 :: []
            -- alternative notation
```

This might suggest a definition of lists of naturals: a `list_nat` is either the empty list `nil_nat`, or `cons_nat hd tl` where `hd : ℕ` and `tl : list_nat` are respectively the head and tail of the list:

```
inductive list_nat : Type
| nil_nat : list_nat
| cons_nat (hd : ℕ) (tl : list_nat) : list_nat
```

There's nothing special about using natural numbers. We can use the same procedure to define lists of integers:

```
inductive list_int : Type
| nil_int : list_int
| cons_int (hd : ℤ) (tl : list_int) : list_int
```

but having to define different types of lists for each type of element is pretty cumbersome. Instead, we can define lists for a generic type `T` using *dependent types*:[2]

```
inductive list (T : Type) : Type
| nil : list
| cons (hd : T) (tl : list) : list
```

[2] a Lean user would probably frown at this, because it would be best to choose an explicit universe `u` and work with `Type u`.

Rather than having `list_nat`, `list_int`... we use `list` $\mathbb{N}$, `list` $\mathbb{Z}$... `list` α where α is any type.

We can see how useful dependent types are by defining the function `length` which returns the number of elements of a list:

```
def length {α : Type*} : list α → ℕ
| []       := 0
| (a :: l) := length l + 1
```

Note that we can use `{α : Type*}` to refer to a generic type α. If instead we had stuck to `list_nat`, `list_int`... now we would have to define `length_nat`, `length_int`... separately for each type.

We will identity tuples of n elements $\mathbb{Z}^n$ with lists in `list` $\mathbb{Z}$ of length `n`.

1.6 The definition in Lean

Syntax and semantics Let's now ask ourselves: how can we define in a satisfactory way the class of functions RPP in Lean, using just types and functions? We'd like to be able to do proofs by induction over RPP, like in proposition 1, so we'll need to define an inductive type.

The key is thinking about RPP not as a class of functions, but as a small programming language. In this sense, we can write down "programs" like our square function

$$(\mathsf{Id}_1 \| \mathsf{Sw}) \fatsemi \mathsf{inc} \fatsemi \mathsf{mul} \fatsemi \mathsf{dec} \fatsemi (\mathsf{Id}_1 \| \mathsf{Sw})$$

but we should not view it only as a function belonging to $\mathbb{Z}^3 \to \mathbb{Z}^3$, but also as the sentence "$(\mathsf{Id}_1 \| \mathsf{Sw}) \fatsemi \mathsf{inc} \fatsemi \mathsf{mul} \fatsemi \mathsf{dec} \fatsemi (\mathsf{Id}_1 \| \mathsf{Sw})$" which can then be interpreted as the mathematical function belonging to $\mathbb{Z}^3 \to \mathbb{Z}^3$.

We thus separate between the *syntax* and the *semantics* of our language.

- The syntax are the rules which governs how to assemble well-structured sentences. For example, the selection symbol If should be followed by three other RPP functions; if we write $\mathsf{If}[\mathsf{Su}, \mathsf{Pr}] \fatsemi \mathsf{Ne}$ we get a non-valid sentence.

- The semantics is the meaning we give to (well-structured) sentences - in our case, they are intepreted as functions $\mathbb{Z}^n \to \mathbb{Z}^n$.

A possible way to define RPP functions in Lean is

- define the type RPP which has for elements syntactically-correct sentences of RPP

- define a function $\mathsf{evaluate} : \mathsf{RPP} \to (\mathbb{Z}^n \to \mathbb{Z}^n)$ which assigns to each RPP-sentence its intended meaning, namely a function $\mathbb{Z}^n \to \mathbb{Z}^n$.

Note that this is not the only way in which this task can be accomplished; we will discuss other methods at the end of this chapter.

We thus define the type RPP as follows:

```
inductive RPP : Type
| Id (n : ℕ) : RPP
| Ne : RPP
| Su : RPP
| Pr : RPP
| Sw : RPP
| Co (f g : RPP) : RPP
| Pa (f g : RPP) : RPP
| It (f : RPP) : RPP
| If (f g h : RPP) : RPP
```

and also introduce custom notation:

```
-- the numbers 50 and 55 denote the precedence -
-- simply put, Ne ;; Su ‖ Pr is intepreted as
-- Ne ;; (Su ‖ Pr), not (Ne ;; Su) ‖ Pr
infix ‘;;‘ : 50 := Co
infix ‘‖‘ : 55 := Pa
```

so it's now possible to write expressions like

```
#check It Su ;; (Id 1 ‖ If Sw Pr Su) -- RPP
```

Remember that by remark 1, it makes sense to consider the series composition of functions of different arity, as long as we give them the meaning specified in the remark.

Talking about arity, how do we deal with it? In order to define evaluate and give meaning to RPP, we must be able to define a concept of arity, otherwise we'll have trouble with parallel composition of two functions `f ‖ g` - the arity of `f` must be known, otherwise it's impossible to tell what to apply `g` to.

Luckily, we can reconstruct the arity of an `RPP` just by looking at its symbolic representation:

```
def arity : RPP → ℕ
| (Id n)      := n
| Ne          := 1
| Su          := 1
| Pr          := 1
| Sw          := 2
| (f ;; g)    := max f.arity g.arity
| (f ‖ g)     := f.arity + g.arity
| (It f)      := f.arity + 1
| (If f g h) := max (max f.arity g.arity) h.arity + 1
```

Note that `f.arity` is the same as `(arity f)`. This is a recursive function: there are 5 base cases and in the other 4 the value of `arity` is reconstructed from smaller sub-terms.

It's now possible to define some `RPP`-sentences in Lean

```
def inc := It Su
```

```
def dec := It Pr
def mul := It inc
def square := Id 1 ‖ Sw ;; inc ;; mul ;; dec ;; Id 1 ‖ Sw
```

and it's even possible to calculate their arity

```
#reduce square.arity -- outputs "3"
```

but we haven't yet given their meaning as functions.

The `evaluate` function We are now ready to define `evaluate` (`ev` for short). The function `ev` should take RPP-sentences and return functions $\mathbb{Z}^n \to \mathbb{Z}^n$, so in Lean we will define it as a function of type

```
RPP → (list ℤ → list ℤ)
```

which in Lean is the same as

```
RPP → list ℤ → list ℤ.
```

Here's how we do it:

```
def ev : RPP → list ℤ → list ℤ
| (Id n)     l                    := l
| Ne         (x :: l)             := -x :: l
| Su         (x :: l)             := (x + 1) :: l
| Pr         (x :: l)             := (x - 1) :: l
| Sw         (x :: y :: l)        := y :: x :: l
| (f ;; g)   l                    := ev g (ev f l)
| (f ‖ g)    l                    := ev f (take f.arity l) ++
                                     ev g (drop f.arity l)
| (It f)     (x :: l)             := x :: ((ev f)^[↓x] l)
| (If f g h) (0 :: l)             := 0 :: ev g l
| (If f g h) (((n : ℕ) + 1) :: l) := (n + 1) :: ev f l
| (If f g h) (-[1+ n] :: l)       := -[1+ n] :: ev h l
| _          l                    := l

notation `<` f `>` := ev f
```

We will write `<f>` to mean the function of type `list ℤ → list ℤ` given by `ev f`.

Here's a case-by-case analysis:

- `<Id n> l` is the original list `l`, unchanged.
- `<Ne> (x :: l)` reduces to `-x :: l`, which is same list but with the head of opposite sign.
- `<Su> (x :: l)` reduces to the same list but with the head incremented by one.

- <Pr> (x :: l) reduces to the same list but with the head decremented by one.

- <Sw> (x :: y :: l) reduces to the same list but with the first two elements swapped.

- <f ;; g> l successively applies <f> and <g> to the list.

- <f ‖ g> l applies <f> to the first f.arity elements of the list, applies <g> to the remaining elements of the list, and then joins the two parts through append (which is the (++) operator).

- <It f> (x :: l) leaves the head unchanged and applies <f> to the tail ↓x times, where ↓x is defined as in definition 1.

- <If f g h> (0 :: l) leaves the head unchanged and applies <g> to the tail.

- <If f g h> (((n : ℕ) + 1) :: l) is the case where the head is a positive number (a natural number plus 1), and as such leaves the head unchanged and applies <f> to the tail.

- <If f g h> (-[1+ n] :: l) is the case where the head is a negative number, and as such leaves the head unchanged and applies <h> to the tail.

- In all cases not considered (for example, applying <Ne> to an empty list) the whole list remains unchanged.

The reader is invited to compare this definition with the one given in definition 1.

Let's see some examples:

```
#check <It Su ;; (Id 1 ‖ If Sw Pr Su)> -- list ℤ → list ℤ
-- #eval is similar to #reduce
-- but in this case gives more readable output
#eval <inc> [3, 4] -- [3, 7]
#eval <square> [19, 0, 0] -- [19, 361, 0]
```

It magically works. We finally have our definition formalized in Lean.

It is worth noting that even though lists supplied to <f> are supposed to have length equal to f.arity, this is never enforced. So we are free to apply <f> to a list which is too short or too long. If it's too short, unspecified things will happen, we don't care. If it's too long, only the first f.arity items are utilized and affected, and this is guaranteed by theorem ev_split which we will prove in Lean. So, when we apply RPP functions to a list, we'll have to make sure that f.arity ≤ l.length.

The `inverse` function It's not hard to convert our proposition 1 into a function definition: `inv : RPP → RPP` which given `f : RPP` returns its inverse.

```
def inv : RPP → RPP
| (Id n)     := Id n
| Ne         := Ne
| Su         := Pr
| Pr         := Su
| Sw         := Sw
| (f ;; g)   := inv g ;; inv f
| (f ‖ g)    := inv f ‖ inv g
| (It f)     := It (inv f)
| (If f g h) := If (inv f) (inv g) (inv h)

notation f `⁻¹` := inv f
```

Now it's possible to define `dec` simply as $\texttt{inc}^{-1}$.

We will also prove in Lean that $\texttt{f}^{-1}$ really is the inverse (in the functional sense) of `f`, but it will require some work.

1.7 Differences with the original definition

The definition of `RPP` functions we've given differs quite a bit from the original one. Every change has been made in the name of simplicity: theorem proving in Lean is hard enough, we don't need to make it harder by choosing inconvenient definitions. Below is a list of changes, not only for completeness' sake but also to illustrate the kind of reasoning which goes on when formalizing definitions in Lean.

- In the original definition, in the iterator It and selection If the last element of the tuple is checked, not the first one (the head). It was more convenient to work with the first element because of the definition of lists: it's much easier to consider a list's head and tail than its last element and the elements before the last.

- We have defined Id_n as a n-ary function, while originally it was just unary. Having a n-ary identity function is very useful, because we can use parallel composition as in remark 1, and also because we have the possibility of having a 0-ary function, which is not useless in some cases.

- The original RPP functions are defined as the union $\cup_{n\in\mathbb{N}}\mathsf{RPP}^n$ where RPP^n are the n-ary RPP functions. A similar decision could've been made in Lean by definining `RPP n` as a dependent type with parameter `n : ℕ`, but it turned out that it was possible to calculate the arity of an `RPP` simply by looking at the corresponding `RPP`-sentence, which is what we did when we defined the function `arity`. This rendered superfluous using dependent types and separating `RPP` based on their arity.

There's a reason we tried to avoid dependent types wherever possible (which also led to the use of `lists` instead of `vectors`): at least in Coq (which is another proof assistants we used at the beginning of the project) working with dependent types is often painful, because Coq doesn't recognize that certain types are the same. For example, elements of `RPP (n + 1)` and `RPP (1 + n)` cannot be compared even though it is (demonstrably!) true that `n + 1 = 1 + n`. To get around this, it's possible to use something called John Major's Equality to state the equality of two objects with seemingly different types, but this involves the invocation of an additional axiom and is in general annoying to use. Other ways to deal with the problem exist, but our choice ended up being avoiding dependent types completely. As someone on the internet says,

> Coq has this really powerful type system, but... don't use it.

By extension, we also avoided them in Lean.

- When defining the iterator $\mathsf{It}[f](x, x_1, \ldots, x_n)$ it's not immediately clear what to do when $x < 0$. In our definition, nothing happens, as f in general is applied $\downarrow x = 0$ times. In the original definition, f is instead applied $|x|$ times - let's call this iterator Ita.

  Reversibility gifts us with a third option: if $x < 0$, we can apply f a negative amount of times - or in other words, we can apply f^{-1} for $-x$ times. Let's call this iterator Itr. Its usage leads to more natural definitions: for example, our function $\mathsf{inc}(n, x) = (\mathsf{It}[\mathsf{Su}])(n, x)$ returns $(n, x + n)$ only if $n \geq 0$. If instead we use Itr, suddenly $(\mathsf{Itr}[\mathsf{Su}])(n, x) = (n, x + n)$ for all values of $n \in \mathbb{Z}$.

  So why didn't we use Itr? Because our It is the most versatile option: we can define both Ita and Itr in terms of It, by using the fact that It doesn't do anything when the first argument is negative:

  $$\mathsf{Ita}[f] = \mathsf{It}[f] \fatsemi \mathsf{Ne} \fatsemi \mathsf{It}[f] \fatsemi \mathsf{Ne}$$
  $$\mathsf{Itr}[f] = \mathsf{It}[f] \fatsemi \mathsf{Ne} \fatsemi \mathsf{It}[f^{-1}] \fatsemi \mathsf{Ne}$$

  For example, in the case of $\mathsf{Ita}[f](x, x_1, \cdots, x_n)$, if $x \geq 0$ then the first It applies f for x times, then x changes sign and becomes $-x$ with Ne, then the second It doesn't do anything because $-x < 0$, and finally $-x$ changes sign again to x; if instead $x < 0$, only the second It does something.

  Another reason to prefer It over Itr is that in the definition of `evaluate`, using Itr it's hard to convince Lean (or Coq) that the function terminates (that is, it doesn't run on an infinite loop). Since every function in Lean must terminate (otherwise there would be consistency issues), Lean rejects the definition. There are ways to get around this - but once again we follow the path of least resistance and just get on with It.

After seeing all these changes you might ask yourself - is this still the original RPP? What's the point of formalizing a definition in Lean if in the process we change the definition completely?

We think that yes, we can still identify what we've constructed as the original functions, because in a way, the *essence* of what RPP is has not been altered: a class of functions which is reversible by construction and that is PRF-complete. We shouldn't view definitions as something unchanging and rigid, especially in rapidly evolving fields. Definitions should be molded and modified to fit our needs, because that's why we created them in the first place.

2. Theorem proving

Here we finally delve into the main characteristic which distinguishes Lean from usual programming languages: the possibility of formally proving results about the objects defined.

2.1 Some examples of Lean proofs

Reflexivity We define the type `Nat` similarly as before

```
inductive Nat : Type
| O : Nat -- it's a capital o, not a zero
| S (n : Nat) : Nat

open Nat
#reduce O -- represents 0
#reduce S O -- represents 1
#reduce O.S -- also represents 1
            -- we'll use this notation
#reduce O.S.S -- represents 2
```

together with addition

```
def Add : Nat → Nat → Nat
| m O      := m
| m (S n') := (Add m n').S

-- if n m : Nat then n + m is defined as Add n m
infix `+` := Add
```

Now let's prove some theorems about these objects. Let's start with something simple: we want to prove, beyond the shadow of a doubt, that `O = O`. We open our code editor and type this:

```
lemma O_eq_O : O = O :=
begin

end
```

If we now place the cursor between `begin` and `end` this appears in the sidebar:

```
1 goal

⊢ 0 = 0
```

This is called the **tactic state**. The line beginning with the turnstile `⊢ 0 = 0` is our **goal**. We can write commands called **tactics** which help us solve goals. In this case, the goal is an equality in which the left-hand side happens to coincide perfectly with the right-hand side, so we can solve our goal using the `refl` command

```
lemma 0_eq_0 : 0 = 0 :=
begin
  refl,
end
```

Placing the cursor just after the comma, a pleasant message appears:

```
goals accomplished
```

The name `refl` stands for "reflexivity", which is a property of equality (for any `a`, it is true that `a = a`). Let's try something more involved: $2 + 2 = 4$.

```
lemma two_plus_two : 0.S.S + 0.S.S = 0.S.S.S.S :=
begin

end
```

This time the left-hand side and the right-hand side do not look identical. However, there's something interesting to note:

```
#reduce 0.S.S.S.S -- outputs "0.S.S.S.S"
#reduce 0.S.S + 0.S.S -- also outputs "0.S.S.S.S"
```

that is, `0.S.S + 0.S.S` reduces to `0.S.S.S.S`. Since the left-hand side and the right-hand side reduce to the same element, they are **definitionally equivalent** and so we can use the tactic `refl` again:

```
lemma two_plus_two : 0.S.S + 0.S.S = 0.S.S.S.S :=
begin
  refl, -- goals accomplished
end
```

Remember that by definition of `Add`, for any `n m : Nat`, we have that `n + m.S = (n + m).S`. We can express this with another theorem

```
lemma plus_S (n m : Nat) : n + m.S = (n + m).S :=
begin
  refl, -- goals accomplished
end
```

which again can be solved using `refl`, because `n + m.S` reduces to `(n + m).S`. By the way, we could write `theorem` instead of `lemma`: the difference is only stylistic. Similarly, by definition of `Add`, `n + 0 = n` for all `n : Nat`

```
lemma plus_0 (n : Nat) : n + 0 = n :=
begin
  refl, -- goals accomplished
end
```

Induction and rewrite If instead we try to prove in the same way that `0 + n = n`,

```
def 0_plus (n : Nat) : 0 + n = n :=
begin
  refl,
end
```

something surprising happens:

```
invalid apply tactic, failed to unify
    0+n = n
  with
    ?m_2 = ?m_2
  state:
  n : Nat
  ⊢ 0+n = n
```

our trusted `refl` has, alas, failed. This is because `0 + n` is **not** definitionally equivalent to `n`: the function `Add` defines two definitional equivalences (`m + 0 = m` and `m + n.S = (m + n).S`) and there's nothing regarding `0 + n` when we have a generic `n : Nat`. However, two things can still be equal even if they are not definitionally equivalent.

To prove `0_plus` we need something stronger: the tactic `induction`. Let's try it:

```
def 0_plus (n : Nat) : 0 + n = n :=
begin
  induction n using n hn,
end
```

Now the tactic state has become

```
  2 goals

  case Nat.0
  ⊢ 0+0 = 0

  case Nat.S
  n: Nat
  hn: 0+n = n
  ⊢ 0+n.S = n.S
```

What happened is that we used induction: to prove a property `P n` (in this case, `0 + n = n`) for all `n : Nat`, it suffices to prove that `P 0` holds and that

`P n` implies `P n.S`. The first subgoal is the base case `0+0 = 0`, and can be solved using `refl`

```
lemma O_plus (n : Nat) : 0 + n = n :=
begin
  induction n with n hn,
  refl,
end
```

Only the second goal remains

```
1 goal

case Nat.S
n : Nat
hn : 0+n = n
⊢ 0+n.S = n.S
```

This means that `n` is an element of `Nat` and that we have an hypothesis named `hn` which tells us that `0+n = n`. Our goal is to prove that `0+n.S = n.S`. A proof of this fact would go somewhat like this:

1. by lemma `plus_S` we have `0+n.S = (0+n).S`
2. by the induction hypothesis `hn` we have `0+n = n` and by substitution we get `(0+n).S = n.S`

so `0+n.S = (0+n).S = n.S`, and this completes the proof.

We can capture this act of substituting a term in an equation with an equivalent one using the tactic `rw` ("rewrite"): for example, `rw plus_S` search in the goal for subterms of the form `0+n.S` and substitutes them with `(0+n).S`. More generally, given an equality `h : a = b`, calling `rw h` substitutes occurrences of `a` in the goal with `b`.

```
lemma O_plus (n : Nat) : 0 + n = n :=
begin
  induction n with n hn,
  refl,
              -- goal: 0+n.S = n.S
  rw plus_S,  -- goal: (0+n).S = n.S
  rw hn,      -- goal: n.S = n.S
              -- refl is automatically called with rw,
              -- so: goals accomplished!
end
```

Let's tackle one more theorem: the commutativiy of addition

```
lemma plus_comm (n m : Nat) : n + m = m + n :=
begin

end
```

Again, using `refl` doesn't work, so we use induction. We have a choice: using induction on `n` or `m`; note that doing induction on one or the other is not the same, because `n` and `m` have asymmetric roles in the definition of `Add`. In particular, the second argument gets "broken down" at each step (since `n + m.S = (n + m).S`) while the first argument doesn't change. Thus, in this case the best choice is induction on `m`.

```
lemma plus_comm (n m : Nat) : n + m = m + n :=
begin
  induction m with m hm, -- 2 goals
                         --
                         -- case Nat.O
                         -- n: Nat
                         -- ⊢ n+O = O+n
                         --
                         -- case Nat.S
                         -- nm: Nat
                         -- hm: n+m = m+n
                         -- ⊢ n+m.S = m.S+n
end
```

We can deal with the base case `n+O = O+n` by using `rw` with our two lemmas `plus_O : n + O = n` and `O_plus : O + n = n`:

```
  rw plus_O, rw O_plus, -- first goal vanquished
```

So now we have hypothesis `hm : n+m = m+n` and goal `n+m.S = m.S+n`. We can use `rw plus_S` to change the goal to `(n+m).S = m.S+n`, and if we could further rewrite it to `(n+m).S = (m+n).S` then we would use the hypothesis to solve the goal. Problem is, we don't have a theorem which states that `m.S+n = (m+n).S` - but we leave it as an exercise for the reader. Having called it `S_plus` we can thus conclude our proof

```
lemma plus_comm (n m : Nat) : n + m = m + n :=
begin
  induction m with m hm,
  rw plus_O, rw O_plus,
  rw plus_S, rw S_plus, rw hm, -- goals accomplished
end
```

A remark: since the rules `n + m.S = (n + m).S` and `n + O = n` are exactly the definition of `Add`, we don't actually need to write `rw plus_S` and `rw plus_O`, we can instead use `rw Add` which includes both these equalities.

After seeing some examples of tactics-based proofs, you might come to the conclusion that they are very unreadable and difficult to understand. That's not entirely false, but it's important to notice that the interactive component of Lean is vital to its usage: just reading tactics it's practically impossible to get what's going on, especially for more involved proofs. On the other hand seeing the hypotheses, the goals, and how they change at each steps of the proof immensely clarifies the process of understanding and usage.

So, we've now learned some basics about theorem proving in Lean, but we don't know anything yet about what proofs *are* and how they fit in the general scheme of things. There is a lot to be learned.

2.2 Curry-Howard correspondence

The following section is not necessary to understand the rest of the thesis, so the busy reader can skip it.

In Lean, things like propositions and proofs are not completely separated from data objects like types and elements of types. We previously stated that in Lean, everything has a type, and we can see what type a certain object has by using the `#check` command.

So, let's feed random stuff to `#check`.

```
#check O -- Nat
#check Nat -- Type
#check Type -- Type 1
#check Type 1 -- Type 2
              -- it's an infinite family of types
              -- for each u, Type u is an element of Type (u+1)
#check Add -- Nat → Nat → Nat
#check Nat → Nat → Nat -- Type

-- some theorem names...
#check two_plus_two -- O.S.S+O.S.S = O.S.S.S.S
#check O_plus -- ∀ (n : Nat), O+n = n
#check plus_comm -- ∀ (n m : Nat), n+m = m+n

-- ..and their statements
#check O.S.S+O.S.S = O.S.S.S.S -- Prop
#check ∀ (n : Nat), O+n = n -- Prop
#check ∀ (n m : Nat), n+m = m+n -- Prop

#check Prop -- Type
```

So, there is a type called `Prop` and its elements are propositions. Any proposition, like `∀ (n : Nat), O+n = n`, is in turn a type - but what exactly are its elements?

The elements of a proposition are proofs of that proposition. This means that what we have called `two_plus_two` is a proof of the fact that `O.S.S + O.S.S = O.S.S.S.S`. What we mean by proving a proposition, is finding an element whose type is that proposition. For example, here is the definition of the proposition `true`:

```
inductive true : Prop -- this is a Prop, not a Type
| intro : true
```

How do we know that `true` is true? Because it has an element, `true.intro` (another way to say it is that `true.intro` is a proof of the proposition `true`):

```
lemma true_is_true : true :=
begin
  exact true.intro, -- if we have an explicit element
                    -- of the proposition to be proven,
                    -- we can use the tactic exact
end
```

The proposition `false` is defined like this:

```
inductive false : Prop
```

It's a type with no elements, so `false` can't be proven.

Suppose that A is a proposition in classical logic. Then it is true that $A \Rightarrow A$ (we can derive it using natural deduction, for example). In Lean this fact can be expressed as the proposition `A → A`, and it's something provable:

```
lemma A_implies_A (A : Prop) : A → A :=
begin
  -- 1 goal
  -- A: Prop
  -- ⊢ A → A
end
```

At the left of the implication arrow we have `A`: we can thus turn `A` into an hypothesis using the `intro` tactic

```
lemma A_implies_A (A : Prop) : A → A :=
begin
  intro h, -- creates a hypothesis h : A
  -- 1 goal
  -- A : Prop
  -- h : A
  -- ⊢ A
  exact h, -- goals accomplished
end
```

Let's forget for a moment that `A` is a proposition, let's think of it as just a type. Then something funny happens: we see that `A → A` is just the type of functions from `A` to `A`. If we can exhibit an element of this type, then we have proven that `A → A`. But this is easy enough: we can use the identity function

```
def A_implies_A' (A : Prop) : A → A
| h := h
```

and this proves the proposition, but also defines a function. It can be interpreted like this: if we have a proof `h : A` and we have to prove `A` then we can just exhibit `h`; that's it.

Amazingly, it turns out all proofs are really just functions: we can see this using the `#print` command, which given a function prints out its definition.

```
#print Add
  -- def Add : Nat → Nat → Nat :=
  -- λ (n m : Nat), Nat.rec n (λ (m' n_m' : Nat), n_m'.S) m
#print A_implies_A
  -- theorem A_implies_A : ∀ (A : Prop), A → A :=
  -- λ (A : Prop) (h : A), h
#print O_plus
  -- theorem O_plus : ∀ (n : Nat), O+n = n :=
  -- λ (n : Nat),
  --   Nat.rec (eq.refl (O+O))
  --     (λ (n : Nat) (hn : O+n = n),
  --        (id (eq.rec (eq.refl (O+n.S = n.S))
  --                    (plus_S O n))).mpr
  --          ((id (eq.rec (eq.refl ((O+n).S = n.S)) hn)).mpr
  --            (eq.refl n.S)))
  --     n
```

When we proved `O_plus` we didn't explicitly write a function, we used tactics, but it's important to notice that the sequence of tactics *isn't* the proof - instead, tactics generate proofs (functions). To further illustrate this point, notice that we can also define the function `Add` using tactics:

```
def Add_tactic (n m : Nat) : Nat :=
begin
  -- 1 goal
  -- n m : Nat
  -- ⊢ Nat
  -- to "solve" the goal we have to provide a Nat
  induction m with m' n_m', -- induction on m
  exact n, -- base case, m=O: we return n
  exact n_m'.S, -- inductive case, Add n m' = n_m'
                -- we have to provide Add n m'.S:
                -- we return n_m'.S
                -- "goals accomplished"
end

#reduce Add_tactic O.S.S O.S.S.S.S -- O.S.S.S.S.S.S
```

And notice! `Add` is a recursive function, so to define it we had to use induction. Going back to `#print Add` and `#print O_plus`, we can see that both call a function called `Nat.rec`. Let's investigate:

```
#check Nat.rec
  -- Nat.rec : ?M_1 O →
  --           (Π (n : Nat), ?M_1 n → ?M_1 n.S) →
  --           Π (n : Nat), ?M_1 n
```

If we intepret `?M_1` as a proposition `P : Nat → Prop` which ranges over a `Nat`, then `Nat.rec` reads off as "if `P O` holds true and for all `n : Nat`, `P n` implies

P n.S, then P n holds for all n : Nat" which is the principle of induction. If instead we interpret ?M_1 as a function f : Nat → T with domain Nat and codomain a certain type T, then Nat.rec reads off as "if we define f 0 and for all n : Nat, given f n we define f n.S, then we have defined a function Nat → T" which is how we define recursive functions.

Nat.rec is called the **induction principle** of Nat, and is auto-generated as soon as Nat is defined. Every type generates an induction principle - even simple types like our earlier weekday - hence every type is defined using the inductive keyword. The same induction principle is used both for recursive functions and inductive proofs.

The remarkable thing about Lean is that the concepts of types and propositions, functions and proofs are united in a single mechanism, which results in a particularly simple foundation for mathematics and computing. This concept is called the **Curry-Howard correspondence** and it's not something unique to Lean - it's a common characteristic of many theorem prover, especially those based on (intuitionistic) type theory.

2.3 The simplification tactic

A hot topic in proof assistants is automatization. For many, being able to generate automatically new theorems and new math is the ultimate objective of theorem provers. Certainly Lean has value even before such a feat is accomplished, but where automation is already avaiable it would be a waste not to use it.

A simple tool which helps tremendously with theorem proving is the simp tactic. It automaticaly tries to apply already known theorems in order to simplify a given expression. We illustrate this with an example:

```
lemma many_adds (n : Nat) :
  (0 + (0 + (n.S + (n.S + (0 + n))))) = (n + (n + n)).S.S :=
begin
  rw O_plus,
  rw O_plus,
  rw O_plus,
  rw S_plus,
  rw S_plus,
  rw plus_S, -- goals accomplished
end
```

Using our previous lemmas it's not hard to prove this, but there's a lot of repetition and any slight change to the statement probably results in some misapplied tactics. Equalities like O_plus : 0 + n = n makes our expression strictly simpler, so usually there wouldn't be any reason to not rewrite it automatically. We can do such a thing by marking the theorem O_plus with the @[simp] tag

```
@[simp] lemma O_plus (n : Nat) : 0 + n = n :=
begin
```

```
  ...
```

and then using the simp tactic in the proof of many_adds:

```
lemma many_adds (n : Nat) :
  (0 + (0 + (n.S + (n.S + (0 + n))))) = (n + (n + n)).S.S :=
begin
  simp, -- 1 goal
        -- n : Nat
        -- ⊢ n.S+(n.S+n) = (n+(n+n)).S.S
end
```

If we also mark lemmas plus_S and S_plus with @[simp], then we can conclude many_adds with a single use tactic:

```
lemma many_adds (n : Nat) :
  (0 + (0 + (n.S + (n.S + (0 + n))))) = (n + (n + n)).S.S :=
begin
  simp, -- goals accomplished
end
```

We can mark with @[simp] theorems which have an equality or a bi-implication as thesis. Notice that we put @[simp] besides theorems that we want to be later utilized by simp, not in theorems where we want to use the tactic simp.

We can also mark definitions and functions, so that direct consequences of the the definition are automatically simplified. For example, marking Add with @[simp] is the same as marking plus_0 and plus_S.

This tactic has some more functionalities: if we want to mark certain theorems or definitions th1, th2, def1, ... just for one simp call, we can write simp[th1, th2, def1, ...] and we can also mark every local hypothesis by using the asterisk simp[*]. Finally, hypotheses themselves can be the target of simplification: if we want to simplify a hypothesis h we write simp at h.

Using simp and marking theorems often starts off an avalanche effect: each new theorem makes simp stronger, which helps it prove new theorems. In the following section we will make heavy use of simp on theorems about RPP.

It's usually not a good idea to indiscriminately mark every equality with @[simp], however. That's because simp is very eager to apply every theorem it can each time it has the opportunity. This means that if we mark something like our add_comm : ∀ (n m : Nat), n+m = m+n then when simp meets a sum n+m, it gets stuck in an infinite loop, endlessly "simplifying" it to m+n and back to n+m. Hence, when we tag an equality as @[simp], it's good practice to make sure that the right-hand side is strictly simpler than the left-hand side, and that no infinite loops can occur.

2.4 Basic theorems about RPP

2.5 A library of functions

2.6 PRF-completeness

2.7 Alternative ways to define RPP in Lean

3. Conclusions

3.1 Future work