Natural Numbers
by Coinductive Construction

by Sven Nilsen, 2018

In path semantics, the `[:] x` sub-type means `[x] x`. It is a function that returns itself.

One can write the following:

```
nat : [:] zero
```

This works because `zero` is an atomic function:

```
zero(nat) → zero
zero : nat
```

Therefore:

```
nat : [zero] zero <=> nat : [:] zero
```

Now, I will use this to define natural numbers:

```
nat : [:] zero ∧ [:] [succ] (x : nat)
```

This is called an “inductive construction” because it defines the type `nat` from the solution.
I interpret this as “`succ` returns some natural number”, without specifying what it returns.

The key is to understand what the following means:

```
[:] [succ] (x : nat)
```

Here, the input to `succ` must be a member of `nat`, so `succ` has the type:

```
succ : nat → nat
```

Other ways to write this is:

```
succ(nat → nat) → succ
nat → nat : [:] succ
```

One such solution is this:

```
succ [:] (zero) : nat
succ [:] (zero) (nat) → succ [:] (zero)
```

It does not say which natural number it returns, but it says that it is a natural number.

A coinductive construction means that if one does not know what it is, it might be a new thing.
For every natural number `x`, there are multiple ways of defining `succ [:] (x)` of that natural number.
So, instead of choosing a particular `succ [:] (x)`, one can define it to be a new natural number.

An inductive construction means that a definition is defined by recursion on constructors, but this is not what I specified here. Instead, I do not assume that anything like a constructor exist, but that I can talk about the function output. This is what makes the construction coinductive.

The power of coinductive construction means that one can talk about natural numbers in many ways. They can map to other natural numbers by narrowing down the definition, such as in modular spaces, but they also can map to e.g. prime numbers. One can narrow down the definition by using sub-types of natural numbers.

With inductive construction is not possible to say that natural numbers can mean many things, so I think that the coinductive construction is more accurate, because it captures the properties of natural numbers that we find intuitive in mathematics.

There are no other ways to get new things without using `zero`. Assume one has the following so far:

```
zero : nat
succ [:] (zero) : nat
```

Notice that the `succ [:]` notation is used because I use atomic functions and normal functions here.
This notation is to make sure that I do not accidentally mix up atomic and normal functions.

Now, one can use `succ [:] (zero)` as a new input:

```
succ [:] (succ [:] (zero)) : nat
```

If one uses `zero`, one just gets `succ [:] (zero)`, which has already been found.
The only new construction is `succ [:] (succ [:] (zero))`.

This gets messy pretty quickly, but luckily, path semantics permits replacing arbitrary symbols with another, as long as it is understood that the symbols are identical:

```
zero          <=>  0
succ [:] (0)  <=>  1
succ [:] (1)  <=>  2
succ [:] (3)  <=>  3
…
```

This can also be interpreted as the natural numbers `succ [:]` maps to.

It also points to a familiar function:

```
succ <=> inc

inc := \(x : nat) = x + 1
```