Recursive types and structural recursion

The *algebraic* types we have looked at so far are essentially *finite*. Looking at the correspondence with algebra or sets we can see that

- `Void` has 0 elements
- `Unit` has 1 element
- if `A` has $`m`$ elements, `B` has $`n`$ elements, then `Sum A B` has $`m + n`$ elements
- if `A` has $`m`$ elements, `B` has $`n`$ elements, then `Prod A B` has $`m \times n`$ elements
- if `A` has `m` elements, `B` has `n` elements, then `A -> B` has $`n^m`$ elements

However, programming becomes algorithmically interesting when we are dealing with *infinite* types such as *natural numbers* or *lists*. A very broad range of infinite types can be defined using *recursive definitions*.

Let us start with a simple but very important such data type: natural numbers. In mathematics you have seen so-called *Peano numbers* which are defined recursively:

- `zero` is a Natural number
- there is an operation `suc`(cessor) such that if `n` is a Natural then `suc n` is a natural.

Above, `zero` and `suc` can be used as *type constructors*:

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

Let us recapitulate the Agda syntax:

- `data` is a keyword indicating a type definition
- `Nat` is the identifier we choose for the type
- `Set` is the built-in identifier for types
  (`data Nat : Set` says *we are defining a type called `Nat`*.)
- `where` is introducing the definitions, which need to be indented
- `zero` is the name of one of the constructors
- `suc` is a constructor that takes a `Nat` as an argument and produces another `Nat`.

For example:

- `zero` is what you would normally write 0
- `suc zero` is what you would normally write 1
- `suc (suc zero)` is 2
- `suc (suc (suc zero)))` is 3

and so on.

> ***Obs:*** *This way of defining natural numbers is theoretically interesting, but not very efficient. For fast programs it is better to*

> *use the built-in "integer" type that is operated on directly by the hardware.*

Let us now consider the basic functions on naturals.

Zero test

Let us start with a very simple function

```
is-zero : Nat -> Boolean
```

which returns `true` if the argument is `zero` and `false` otherwise. This siple function will illustrate pattern-matching on the `Nat`s: two constructors can give us two cases:

```
is-zero : Nat -> Boolean
is-zero zero    = true
is-zero (suc n) = false
```

Above `n` is a *variable* which will be instantiated to the argument of *suc*. Since we are not actually using *n* we could instead write:

```
is-zero : Nat -> Boolean
is-zero zero    = true
is-zero (suc _) = false
```

or even (taking into account top-down pattern-matching priority):

```
is-zero : Nat -> Boolean
is-zero zero  = true
is-zero _     = false
```

For example,

```
is-zero (suc (suc zero)) = false
         ^^^  ^^^^^^^^
         suc      n
```

because the outer `suc` is pattern-matched against the constructor, and `suc zero` used to instantiate `n` (and ignored).

Doubling

Let us continue with a rather boring but simple function: doubling a `Nat`:

- 0 … 0
- 1 … 2
- 2 … 4
- 3 … 6
- …

This function takes a `Nat` into a `Nat`, i.e.

```
dbl : Nat -> Nat
```

Because the `Nat` type has two constructors we can define `dbl` by pattern-matching the constructors:

```
dbl : Nat -> Nat
dbl zero    = zero
dbl (suc n) = ?
```

The first case (`zero`) is obvious. But what about the second case (`suc`)? What we want is to double the number, which means doubling the `suc`s in the number. In the argument we have one `suc` so in the result we should have two `suc`s:

```
dbl : Nat -> Nat
dbl zero    = zero
dbl (suc n) = suc (suc ?))
```

But what about `n`? How do we double it up? By using `dbl` itself! This definition of a function in terms of itself is called *recursion* and you have seen it in the DSA module:

```
dbl : Nat -> Nat
dbl zero    = zero
dbl (suc n) = suc (suc (dbl n))
```

Remember that using recursion we can write functions that evaluate forever or which blow up the stack. However, such functions are not allowed in Agda? How is this reconciled with recursion? Agda has a rathar powerful *termination checker* which attempts to verify all functions for termination. This is not always possible (because it is a problem ultimately equivalent to the Halting Problem) but it many cases it is. In the case of `dbl` we can see that in the recursion call:

```
dbl (suc n) = suc (suc (dbl n))
     ^^^^^                 ^
```

the argument of `dbl` gets *strictly smaller* by “stripping a layer” of constructors. Since `dbl` can only be called with a *finite* `Nat` we can be guaranteed that eventually `zero` will be reached and the “base” non-recursive case will be executed:

```
dbl zero    = zero
```

Defining a recursive function this way is called ***structural recursion***. It is a restricted, simpler, form of recursion. We will stick to it in the rest of the module.

Addition

Let us now consider our first generally useful function, addition.

Addition requires two `Nat` arguments and produces a `Nat`. Because the `Nat` type has two constructors we can take two cases on each, so a total of four cases:

```
add : Nat -> Nat -> Nat
add zero    zero    = ?
add zero    (suc n) = ?
add (suc n) zero    = ?
add (suc m) (suc n) = ?
```

The first three, which are defining addition with 0 are immediate (and non-recursive):

```
add : Nat -> Nat -> Nat
add zero    zero    = zero
add zero    (suc n) = suc n
add (suc n) zero    = suc n
add (suc m) (suc n) = ?
```

The final case will involve recursion. Informally, `suc m` can be thought of as `1 + n`. To define addition in this case

`(1 + m) + (1 + n) = ?`

we need to re-organise the sum so that we can put `m + n` together, which will allow a recursive call:

`(1 + m) + (1 + n) = 1 + 1 + m + n`

If we re-bracket the RHS as follows, the implementation becomes apparent:

`(1 + m) + (1 + n) = (1 + (1 + (m + n)))`

```
add (suc m) (suc n) = suc (suc (add m n))
```

To put it all together:

```
add : Nat -> Nat -> Nat
add zero    zero    = zero
add zero    (suc n) = suc n
add (suc n) zero    = suc n
add (suc m) (suc n) = suc (suc (add m n))
```

Example: Let us revisit (yet again!) our favourite example, `1 + 1 = 2`.

```
1 + 1
  = add (suc zero) (suc zero)    ... pattern-match the fin
al case
             ^m^^       ^n^^
  = suc (suc (add zero zero)     ... pattern-match the fir
st case
  = suc (suc zero)               ... = 2
```

Another addition

Note that the definition of addition is obviously not unique. We could have defined it just as well as

```
add : Nat -> Nat -> Nat
add zero    zero    = zero
add zero    (suc n) = suc n
add (suc n) zero    = suc n
add (suc m) (suc n) = suc (suc (add n m))
```

because `(1 + m) + (1 + n) = (1 + (1 + (n + m)))`.

But it would have been a bit weird. When we *define* functions, particularly in this context where we focus on correctness, we prefer simpler definitions. Addition, as defined above, has 4 cases, which is actually excessive.

Could we define addition just by pattern matching the first argument, and taking the second as a generic variable? (Noting that variables match any pattern.)

```
add : Nat -> Nat -> Nat
add zero    n = n
add (suc m) n = ?
```

The first case is immediate. In the second case we need to rearrange the operations so that we can perform the recursive call, just as above:

`(suc m) + n = (1 + m) + n = 1 + m + n = 1 + (m + n) = suc (m + n)`

which gives us:

```
add : Nat -> Nat -> Nat
add zero    n = n
add (suc m) n = suc (add m n)
```

Note that in the recursive call only the first argument becomes smaller, from `suc m` to `m`, whereas the second `n` stays the same. This is still good enough to guarantee termination: *something* gets smaller, so in time we will reach the `zero`.

The second definition is simpler, and is the standard addition for Peano numbers.

Exercises:

- Define the *halving* operation, approximating 1/2 as 0.
- Define an equality test `eq : Nat -> Bool`.
- Define the *multiplication* operation, using addition, as defined above.

  Hint:

`m * 1 = m`

`m * (1 + n) = m * 1 + m * n = m + (m * n)`

- Define isomorphisms between the types below:
  - `Nat ~ Sum Nat Unit`
  - `Nat ~ Sum Nat Nat`
  - (**Hard**) `Nat ~ Prod Nat Nat`

    Note: Just the functions, no proofs required.

> ***Note:*** *A correspondence between types and logic (and algebra) can be maintained for recursive types but it is beyond the scope of this module. It will be explored in the Year 2 module Functional Programming. For now we will concentrate on recursive types only.*