Chapter 2

Simple Types

Before jumping straight into dependent types with all its beautiful complexities we will first have a look at something simpler, appropriately named *simple types*. We start of with the most fundamental type of all, the function type $\_\to\_$ (section 2.1).

One issue with simple types (which is repeated next chapter when we look at propositional logic) is that it is not clear what to start with since we haven't yet got any types to built functions with. For examples I will use types line $\mathbb{N}$. We will also look at constructions which work for any type, called Set in Agda. That is we will be using *polymorphic* constructions without introducing a formal account for polymorphism which we will do later once we have dependent types. A particular example of polymorphic constructions are identity id and composition $\_\circ\_$ (section 2.2) which give us a first taste of category theory.

We will have a superficial look at λ-calculus (section 2.3), by which I always mean typed λ-calculus because the untyped one doesn't make any sense to me. At least enough to see that variables are causing a bit of a headache, something which can be avoided my using combinatory logic which features just two combinators (S and K), but the price we have to pay is the creation fo some completely unreadable combinator code. However, I think it is worthwhile to know about them and we will return to combinators later from a more formal perspective.

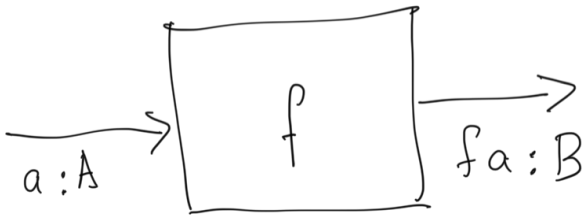
Functions are not all there is to simple types and hence we are going to look at sums $(\_\uplus\_)$ and products $(\_\times\_)$ (section 2.5 which let us encode finite types. Finally we will meet some old acquaintances from high school in the context of type theory.

2.1 Functions

In Type Theory functions are a fundamental concept. This is different from set theory where functions are viewed as a special kind of relations, namely relations that assign to each element of the domain (the input) exactly one element of

the codomain (the output). In Type Theory it is the other way around: we start with functions and actually we need functions to say what a relations is, namely a relation between A and B is a function from A and B to the type of propositions. But lets not get ahead of ourselves.

Intuitively, a function between types A and B is a black box where we can input elements of $\mathsf{a} : \mathsf{A}$ and out come elements of $\mathsf{f\,a} : \mathsf{B}$.



You may notice that we don't write $f(a)$ as usual in Mathematics and in many programming languages but instead save brackets and just write $\mathsf{f\,a}$, which is common in functional programming languages like Haskell. We write $\mathsf{A} \to \mathsf{B}$ for the type of functions from A to B and hence we write $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ to express that f is such a function.

I was saying *black box*, even though for reasons of readability I didn't actually draw a black box. The meaning here is rather metaphorical, it means we cannot look into the box. If somebody gives you a function, all you can do it is to feed it elements and observe the output. You have no way to explore the mechanism (this is a sort of digital rights management for functions). This aspect will become important when we discuss extensionality later.

Let's look at some examples! For the purpose of illustration I will use some types like the natural numbers $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ [1] [2] and the booleans $\mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$ even though we will only introduce them later. We will also use some basic operations such as addition $(\_+\_)$ on natural numbers.

Let's define a function which adds 2 to its input

$$\begin{aligned}&\mathsf{add2} : \mathbb{N} \to \mathbb{N}\\ &\mathsf{add2\,x} = \mathsf{x} + 2\end{aligned}$$

To define $\mathsf{add2}$ we introduce a parameter x. Now we can apply $\mathsf{add2}$ to a number as in $\mathsf{3}$ and observe $\mathsf{add2\ 3}$ which is (no surprise) $\mathsf{5}$. Lets do this step by step:

$$\begin{aligned}\mathsf{add2\ 3} &= \mathsf{3 + 2} && \text{Use the definition of } \mathsf{add2}\\ &= \mathsf{5} && \text{Calculating the sum.}\end{aligned}$$

[1] In computer science unlike in mathematics we start counting with 0, which enables us to answer questions like *How many elephants are in your fridge?*

[2] You may think I have succumbed to set theory already by using $\{\dots\}$. But this is not the case, this is just a convenient notation for labelled coproducts which we will introduce soon.

In the first step we replace the parameter x with the actual argument 2 and then we use our knowledge about addition to conclude. The first step, replacing the parameter with the argument, is called β-reduction.

The definition above combines naming and defining the function. If we just want to write a function without giving it a name we can use λ-abstraction. That is we could have defined the same function as follows:

$$\begin{aligned}&\mathsf{add2} : \mathbb{N} \to \mathbb{N}\\ &\mathsf{add2} = \lambda\,\mathsf{x} \to \mathsf{x} + 2\end{aligned}$$

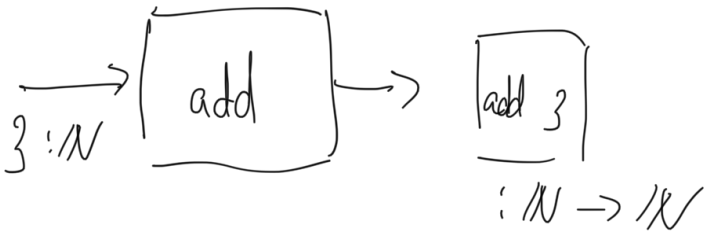
We can view the first definition as an abbreviation for the 2nd. We can now give a more detailed derivation where we differentiate the unfolding of a definition and β-reduction.

$$\begin{aligned}\mathsf{add2}\ 3 &= (\lambda\,\mathsf{x} \to \mathsf{x} + 2)\ 3 && \text{Unfolding the definition of } \mathsf{add2}.\\ &= 3 + 2 && \beta\text{-reduction}\\ &= 5 && \text{Calculating the sum.}\end{aligned}$$

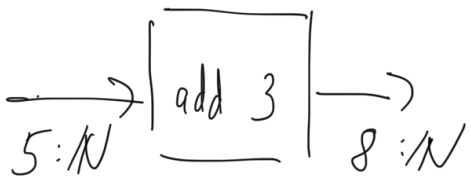
Using λ we can only define a functions with one parameter. To define a function with several parameters we use *currying*, that is a function that returns a function

$$\begin{aligned}&\mathsf{add} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})\\ &\mathsf{add} = \lambda\,\mathsf{x} \to (\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y})\end{aligned}$$

So $\mathsf{add}\ 3 : \mathbb{N} \to \mathbb{N}$ is the function $\lambda\ \mathsf{y} \to 3 + \mathsf{y}$ that is the function that adds 3, and $(\mathsf{add}\ 3)\ 5$ reduces to $5 + 3$ and hence 8. We can illustrate this – add outputs another box $\mathsf{add}\ 3$:



which we can then feed with the 2nd argument to get the final output:



There are some syntactic conventions to simplify the use of currying: $\to$ is *right associative*, hence we can write $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ for $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ and application is *left associative* hence we can write $\mathsf{g}\ 3\ 5$ for $(\mathsf{g}\ 3)\ 5$. Moreover we can combine several λ-abstractions and write $\lambda\ \mathsf{x}\ \mathsf{y} \to \mathsf{x} + \mathsf{y}$ for $\lambda\ \mathsf{x} \to (\lambda\ \mathsf{y} \to \mathsf{x} + \mathsf{y})$. Consequently the same convention also applies for the explicit function definition:

$$\begin{aligned}&\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\\ &\mathsf{add}\ \mathsf{x}\ \mathsf{y} = \mathsf{x} + \mathsf{y}\end{aligned}$$

Instead of returning a function we can also have functions that have functions as input (these are called *higher order functions*). An example is:

$$\begin{aligned}&\mathsf{k} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}\\ &\mathsf{k}\ \mathsf{h} = \mathsf{h}\ 2 + \mathsf{h}\ 3\end{aligned}$$

Note that in this case we cannot omit the brackets in the function type. k is a function that gets a function on the natural numbers as input and returns a number. What is $\mathsf{k}\ \mathsf{add2}$? We can reason as follows:

$$\begin{aligned}\mathsf{k}\ \mathsf{add2} &= \mathsf{add2}\ 2 + \mathsf{add2}\ 3\\ &= (2 + 2) + (3 + 2)\\ &= 9\end{aligned}$$

2.2 Identity and composition

Some functions work for any type, we call them *polymorphic*. An example is the identity function $\mathsf{id} : \mathsf{A} \to \mathsf{A}$ which works for every type $\mathsf{A} : \mathsf{Set}$. In Agda we can write :

$$\begin{aligned}&\mathsf{id} : \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{A}\\ &\mathsf{id}\ \mathsf{x} = \mathsf{x}\end{aligned}$$

Writing $\{\mathsf{A} : \mathsf{Set}\} \to \ldots$ indicates that the function works for every Set. Agda will automatically instantiate A that is we can just write $\mathsf{id}\ 3 : \mathbb{N}$ and Agda infers that $\mathsf{A} = \mathbb{N}$ in this case. In some cases it may be necessary to instantiate

the type variable explicitly in this case we can write id $\{\mathbb{N}\}$ or in case there are many parametrs but we only want to instantiate a specific one we write id $\{A = \mathbb{N}\}$.

We will later explain types like $\{A : Set\} \to ...$ which are actually instances of dependent types. In the moment we view polymorphism as a metamathematical notion, and we will use A B C : Set to indicate variable types. In Agda this can be achieved by declaring:

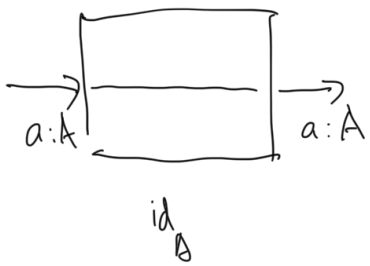
```
variable
  A B C : Set
```

which means that I can write

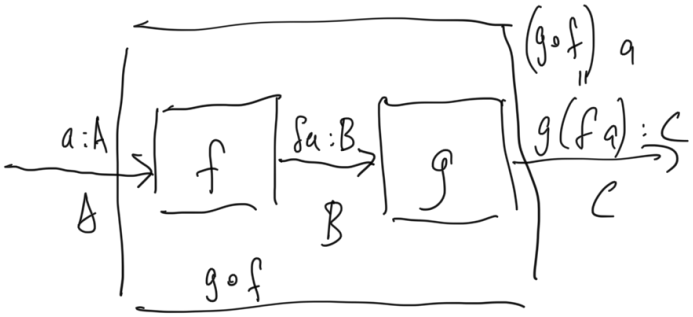
```
id : A → A
id x = x
```

and Agda will automatically translate the type into $\{A : Set\} \to A \to A$.

We can draw a picture for the identity function which just consists of a wire from the input to the output:



Another example is function composition: given $g : A \to B$ and $f : B \to C$ we can construct a new function $f \circ g : A \to C$ which feeds the output of g into the input of f:



Composition can also be defined as a polymorphic function:

```
_∘_ : (B → C) → (A → B) → (A → C)
(g ∘ f) x = g (f x)
```

In Agda we can define infix functions (and indeed more complicated syntactic schemes) by using _ to indicate where the arguments go. The price we have to pay for this flexibility is that we have to separate any syntactical component with spaces, e.g. Agda would read g∘f just as one identifier.

Let's do an example. Earlier we define the function add2 : $\mathbb{N} \to \mathbb{N}$, in the same vain we can define a squaring function:

```
square : ℕ → ℕ
square x = x * x
```

Now we can construct square ∘ add2 : $\mathbb{N} \to \mathbb{N}$, what is (square ∘ add2) 3? Let's calculate:

$$\begin{aligned}(\mathsf{square} \circ \mathsf{add2})\ 3 &= \mathsf{square}\ (\mathsf{add2}\ 3)\\ &= \mathsf{square}\ (3 + 2)\\ &= \mathsf{square}\ 5\\ &= 5 * 5\\ &= 25\end{aligned}$$

This example can also be used to show that composition is not commutative,

what is (add2 ∘ square) 3?

$$\begin{aligned}(\mathsf{add2} \circ \mathsf{square})\ 3 &= \mathsf{add2}\ (\mathsf{square}\ 3)\\ &= \mathsf{add2}\ (3 * 3)\\ &= \mathsf{add2}\ 9\\ &= 2 + 9\\ &= 11\end{aligned}$$

Hence we can see that square ∘ add2 and add2 ∘ square are different functions. In any case it isn't always possible to turn around composition. Assume as given a function IsEven : ℕ → Bool that returns true if the input is even but false otherwise (e.g. isEven 3 = false but isEven 6 = true). Now we can form isEven ∘ square : ℕ → Bool (which actually behaves the same way as isEven) but it doesn't make sense to form square ∘ isEven because the output of isEven is Bool and this doesn't match the input of square which is ℕ.

You may note that there is a strange twist in the order of arguments to _∘_ which is visible in its type (B → C) → (A → B) → (A → C). Why is it not (A → B) → (B → C) → (A → C)? And indeed when we evaluate f ∘ g we first evaluate g and then f. However, the reason is that function application is already written *the wrong way around*, that is when we evaluate square (add2 3) we first evaluate add2 3 and then square 5, which is actually counterintuitive for people who write and read from left to right. Maybe we should write function application postfix 3 add2 instead of add2 3. However, it seems to be too late to change this convention. And since (f ∘ g) x = f (g x) it seems wiser not to change the order when defining composition.

2.3 λ-calculus

I don't want to give a fully formal introduction to λ-calculus at this point. We will do this later when we have developed enough type theory to do this in Agda itself.

As we have already discussed we can view the explicit definition of a function like

```
_∘_ : (B → C) → (A → B) → (A → C)
(f ∘ g) x = f (g x)
```

is a shorthand for an explicit definition using λ-terms:

```
_∘_ : (B → C) → (A → B) → (A → C)
_∘_ = λ f → λ g → λ x → f (g x)
```

And since unfolding definitions is standard practice in Mathematics and elsewhere we concentrate on what happens to λ-terms.

When we form a λ-term we are using variables. Hence beginners often ask *What is the type of a variable?* The answer is that we assume that we are given a type of any variable, this is often called the (typing-) context.

The other cases are application and λ-abstraction:

variable A variable x has the type we have assumed for it in the context.

application Given a term $\mathsf{M} : \mathsf{A} \to \mathsf{B}$ and a term $\mathsf{N} : \mathsf{A}$ we can form a term $\mathsf{M}\,\mathsf{N} : \mathsf{B}$.

abstraction Given a variable x [3] if from assuming $\mathsf{x} : \mathsf{A}$ (here we extend the context) we can show that $\mathsf{M} : \mathsf{B}$ then we can form $\lambda\,\mathsf{x} \to \mathsf{M} : \mathsf{A} \to \mathsf{B}$. I need to add that we will only assume one type for a variable, that is by assuming $\mathsf{x} : \mathsf{A}$ we are ignoring all earlier assumptions regarding x.

This covers the *pure* λ-terms which don't use constructions specific to datatypes like $\mathbb{N}$ or Bool (but we may assume that the types of standard functions like $\_+\_$ or isEven are in the context.

I have already mentioned β-equality. By β-equality we mean that we can reduce an application of a λ-abstraction to an argument by substituting the variable with the argument, that is

$$(\lambda\,\mathsf{x} \to \mathsf{M})\,\mathsf{N} = \mathsf{M}[\mathsf{x}{:=}\mathsf{N}]$$

where $\mathsf{M}[\mathsf{x}{:=}\mathsf{N}]$ is M with all occurrences of x replaced by N.

Hang on, actually it is a bit more complicated. We certainly do not want to replace bound variables, that is $(\lambda\,\mathsf{x} \to (\lambda\,\mathsf{x} \to \mathsf{x}))\,3$ should just by $\lambda\,\mathsf{x} \to \mathsf{x}$ and not $\lambda\,\mathsf{x} \to 3$. We remedy this by saying that $\mathsf{M}[\mathsf{x}{:=}\mathsf{N}]$ is M with all *free* occurrences of x replaced by N. An occurence is free if it is not bound.

We are not done yet! Consider $\lambda\,\mathsf{x} \to (\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y})$ and let's say we have a variable y lying around. What is $(\lambda\,\mathsf{x} \to (\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y}))\,\mathsf{y}$? If we mechanically replace x by y we obtain $\lambda\,\mathsf{y} \to \mathsf{y} + \mathsf{y}$. This is wrong because the y to refers to the bound variable y and not to our top-level y. This is called variable capture and it has to be avoided.

But then what is $(\lambda\,\mathsf{x} \to (\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y}))\,\mathsf{y}$? Here we introduce another equality, α-equivalence which says that bound variables can be consistently replaced, i.e. $\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y} = \lambda\,\mathsf{z} \to \mathsf{x} + \mathsf{z}$. Using this we can avoid capture by replacing the bound variable y avoiding capture, that is $(\lambda\,\mathsf{x} \to (\lambda\,\mathsf{y} \to \mathsf{x} + \mathsf{y}))\,\mathsf{y} = \lambda\,\mathsf{z} \to \mathsf{y} + \mathsf{z}$. Note that any variable but y would work here.

There is yet another equation, η-equality which is motivated by the idea of extensionality, that is that two functions which are equal when applied to the same argument should be considered equal. In pure λ-calculus this is obtained by adding the following equation: assume $\mathsf{M} : \mathsf{A} \to \mathsf{B}$ and given a variable $\mathsf{x} : \mathsf{A}$ which does not appear free in M, then

$$\lambda\,\mathsf{x} \to \mathsf{M}\,\mathsf{x} = \mathsf{M}$$

[3] Note that here x is not the specific variable called x but a metavariable standing for any concrete variable.

Reading it from right to left it means that if you want to show that two functions M and N are equal it is enough to show that M x = N x where x is a variable which does not appear in M or N because

$$\begin{aligned} \mathsf{M} &= \lambda\, \mathsf{x} \to \mathsf{M}\, \mathsf{x} \\ &= \lambda\, \mathsf{x} \to \mathsf{N}\, \mathsf{x} \\ &= \mathsf{N} \end{aligned}$$

As an example we use η-equality to show that f ∘ id = f:

$$\begin{aligned} \mathsf{f} \circ \mathsf{id} &= \lambda\, \mathsf{x} \to \mathsf{f}\, (\mathsf{id}\, \mathsf{x}) \\ &= \lambda\, \mathsf{x} \to \mathsf{f}\, \mathsf{x} \\ &= \mathsf{f} \end{aligned}$$

2.4 Combinatory logic

The polymorphic functions from section 2.2, id and _∘_ are also called combinators. We are going to introduce two combinators in this section, called S and K, which are *functionally complete*. This means that every function which we can write in (pure) λ-calculus can be written just using these two combinators. This applies only to pure functions which do not refer to other datatypes (like ℕ or Bool), which would require additional combinators. We are going to show this by providing a translation from λ-terms into terms only using combinators, which means that we can eliminate all variables. In a way combinators are a form of functional machine code and they have indeed been used for compilation.

We start with K which introduces constant functions:

```
K : A → B → A
K x y = x
```

So for example K 2 : B → ℕ is the function that will always return 2. Note that this function is still polymorphic, it works for any type B.

The other combinator is S which generalizes composition, given types A B C:

```
S : (A → B → C) → (A → B) → (A → C)
S f g x = f x (g x)
```

S is similar to _∘_ but the first functions gets to see not only the result of the 2nd but also the original input. So for example: S (λ x y → x + y) (λ x → 2 * x) is the same as λ x → x + 2 * x. [4]

We can derive id (which is usually written I in combinatory logic) from S and K. The idea is that we can derive it from S by replacing the first argument f with K this obtaining S K = λ g x → K x (g x) = λ g x → x. To get the identity we can substitute g with any function. The simplest choice is K, i.e.

[4] We need to instantiate A,B,C with ℕ which is done automatically by agda.

$\mathsf{id} = \mathsf{S}\,\mathsf{K}\,\mathsf{K}$. When implementing this in Agda we run into a bit of a technical problem, given a type A we would like to define:

```
id : A → A
id = S K K
```

Actually Agda is going to complain because it cannot infer the argument B for the 2nd K – indeed we can use any type here. The simplest choice is A and we need to tell Agda this explictely:

```
id : A → A
id {A} = S K (K {B = A})
```

We will now show that every pure lambda term can be translated into combinatory logic. I will do this informally we will later use Agda itself to give formal proof of this theorem.

Now let's do composition as an example. We can write $\_\circ\_$ as a pure λ-term

```
_∘_ = λ f g x → f (g x)
```

Actually for our purposes it is better to expand the shorthand for repeated λ-abstraction:

```
_∘_ = λ f → λ g → λ x → f (g x)
```

Our strategy is to eliminate the λs from inside out, that is first we translate $\lambda\,\mathsf{x} \to \mathsf{f}\,(\mathsf{g}\,\mathsf{x})$ into combinators. This term will still contain the variables f and g. However, next we abstract g and then f to obtain a term using only combinators that contains no variables.

The terms which appear during the translation are terms which have no λ-abstractions but may contain variables, applications and the combinators S and K. We are going to construct $\lambda\,\mathsf{x} \to \mathsf{M}$ by looking at each of the possible cases:

M=x In this case clearly $\lambda\,\mathsf{x} \to \mathsf{x} = \mathsf{id}$ which we have already obtained as $\mathsf{S}\,\mathsf{K}\,\mathsf{K}$.

M=y If the variable is different from x we have a function $\lambda\,\mathsf{x} \to \mathsf{y}$ which is a constant function, that is $\mathsf{K}\,\mathsf{y}$.

M = $\mathsf{M}_1\,\mathsf{M}_2$ In this case we use the assumption that we know already how to translate $\lambda\,\mathsf{x} \to \mathsf{M}_1$ and $\lambda\,\mathsf{x} \to \mathsf{M}_2$ Now we have

$$\begin{aligned}\lambda\,\mathsf{x} \to \mathsf{M} &= \lambda\,\mathsf{x} \to \mathsf{M}_1\,\mathsf{M}_2 \\ &= \mathsf{S}\,(\lambda\,\mathsf{x} \to \mathsf{M}_1)\,(\lambda\,\mathsf{x} \to \mathsf{M}_2)\end{aligned}$$

which explains the need for S.

M = K Clearly $\lambda\,\mathsf{x} \to \mathsf{K}$ is just a constant function returning K, hence $\mathsf{K}\,\mathsf{K}$.

$\mathsf{M} = \mathsf{S}$ As above $\lambda\,\mathsf{x} \to \mathsf{S}$ is just $\mathsf{K}\,\mathsf{S}$.

Ok, let's apply the translation to our example $\lambda\,\mathsf{f} \to \lambda\,\mathsf{g} \to \lambda\,\mathsf{x} \to \mathsf{f}\,(\mathsf{g}\,\mathsf{x})$. We start with

$$\begin{aligned}
&\lambda\,\mathsf{x} \to \mathsf{f}\,(\mathsf{g}\,\mathsf{x}) \\
&= \mathsf{S}\,(\lambda\,\mathsf{x} \to \mathsf{f})\,(\lambda\,\mathsf{x} \to \mathsf{g}\,\mathsf{x}) \\
&= \mathsf{S}\,(\mathsf{K}\,\mathsf{f})\,(\mathsf{S}\,(\lambda\,\mathsf{x} \to \mathsf{g})\,(\lambda\,\mathsf{x} \to \mathsf{x})) \\
&= \mathsf{S}\,(\mathsf{K}\,\mathsf{f})\,(\mathsf{S}\,(\mathsf{K}\,\mathsf{g})\,\mathsf{id})
\end{aligned}$$

Before proceeding let's introduce some optimisations to avoid truly gigantic terms. So for example we can see that $\lambda\,\mathsf{x} \to \mathsf{g}\,\mathsf{x}$ is just g using η. Another optimisation is that if the variable x does not appear in M then $\lambda\,\mathsf{x} \to \mathsf{M}$ is just $\mathsf{K}\,\mathsf{M}$ and there is no need to go through M. Using these optimisations we translate $\_\circ\_$ in one go

$$\begin{aligned}
&\lambda\,\mathsf{f} \to \lambda\,\mathsf{g} \to \lambda\,\mathsf{x} \to \mathsf{f}\,(\mathsf{g}\,\mathsf{x}) \\
&= \lambda\,\mathsf{f} \to \lambda\,\mathsf{g} \to \mathsf{S}\,(\lambda\,\mathsf{x} \to \mathsf{f})\,(\lambda\,\mathsf{x} \to \mathsf{g}\,\mathsf{x}) \\
&= \lambda\,\mathsf{f} \to \lambda\,\mathsf{g} \to \mathsf{S}\,(\mathsf{K}\,\mathsf{f})\,\mathsf{g} \\
&= \lambda\,\mathsf{f} \to \mathsf{S}\,(\mathsf{K}\,\mathsf{f}) \\
&= \mathsf{S}\,(\lambda\,\mathsf{f} \to \mathsf{S})\,(\lambda\,\mathsf{f} \to \mathsf{K}\,\mathsf{f}) \\
&= \mathsf{S}\,(\mathsf{K}\,\mathsf{S})\,\mathsf{K}
\end{aligned}$$

While this term is comparatively small it is still hard to understand. The point of combinator is rather to show that we can avoid all the complications involving variables and just get away with S and K instead.

2.5 Products and Sums

So far we have only encountered the function type. In this section we will get to know some more basic type formers:

products Written $\mathsf{A} \times \mathsf{B}$, and also called cartesian product. In basic Mathematics we use products to represent coordinate systems, like

$$(1\,,1) : \mathbb{R} \times \mathbb{R}.$$

sums In Agda we write $A \uplus B$, I prefer $A + B$ but this clashes with $+$ on numbers. They are also called disjoint unions and coproducts.

Actually lets start with sums which are less common in conventional Mathematics but more common in Type Theory and functional programming.

2.5.1 Sums

Sums are necessary to represent alternatives. We have one thing or another. To define sums we use the **data** constructor:

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

This says that an element of A ⊎ B is either inj_1 a where a : A or it is inj_2 b where b : B.

A simple example is that if in a form you can either fill in your order number (OrdNum) or your customer reference (CustRef) we can represent this as

$$\mathsf{OrdNum} \uplus \mathsf{CustRef}.$$

Note that even if we use the same number, lets say 1704 as an order number a customer reference we can always sure which is meant because we will use inj_1 1704 if it is an order number and inj_2 1704 if it is a customer reference.

The symbol for sums in Agda is a bit strange: it is a combination of the symbol for set theoretic union $\cup$ and $+$, this is related to the name for sums in set theory *disjoint union*. I prefer to just use $+$ but this clashes with the use for numbers.

How are $\cup$ and $\uplus$ related? Indeed, there is no operation $\cup$ on types. The reason is that $\cup$ in an *intensional* operator whose meaning depends on the choice of representation. For example let $A = \{\mathsf{a}, \mathsf{b}\}$ and $B = \{0, 1, 2\}$ then

$$A \cup B = \{\mathsf{a}, \mathsf{b}, 0, 1, 2\}$$
$$A \uplus B = \{\mathsf{inj}_1\ \mathsf{a}, \mathsf{inj}_1\ \mathsf{b}, \mathsf{inj}_2\ 0, \mathsf{inj}_2\ 1, \mathsf{inj}_2\ 2\}$$

However, we can view A just as the one possible representation of a type with 2 elements. Another choice would be $A' = \{0, 1\}$. What happens now with $\cup$ and $\uplus$?

$$A' \cup B = \{0, 1, 2\}$$
$$A' \uplus B = \{\mathsf{inj}_1\ 0, \mathsf{inj}_1\ 1, \mathsf{inj}_2\ 0, \mathsf{inj}_2\ 1, \mathsf{inj}_2\ 2\}$$

$A' \cup B$ now has only 3 elements, while $A' \uplus B$ again has 5 elements, hence is just another representation of a type with 5 elements. Thus $\uplus$ is insensitive to the representation of a type, this is a property which holds for all operations definable in Type Theory (and hence in Agda).

What can we do with $\uplus$? A useful general combinator is case that performs case analysis. we can define it using *pattern matching*:

```
case : (A → C) → (B → C) → A ⊎ B → C
case f g (inj₁ a) = f a
case f g (inj₂ b) = g b
```

case chooses between two functions, f knows how to handle A and g know how to handle B. Going back to our example with the order form (OrdNum ⊎ CustRef): if we have a function to lookup an order using the order number

```
lookupOrdNum : OrdNum → Order
```

and a function

```
lookupCustRef : CustRef → Order
```

we can use case to combine them

```
case lookupOrdNum lookupCustRef : OrdNum ⊎ CustRef → Order
```

2.5.2 Products

We define products as a special case of a *record type*:

```
record _×_ (A B : Set) : Set where
  field
    proj₁ : A
    proj₂ : B
```

that is we specify the projections, which are functions with the types:

```
proj₁ : A × B → A
proj₂ : A × B → B
```

We can construct elements of A × B by specifying what the result is when applying projections. This is called *copatternmatching*. For example we can use this to define the pairing constructor which in Agda is just the _,_, that is one doesn't have to put brackets around the tuple.

```
_,_ : A → B → A × B
proj₁ (a , b) = a
proj₂ (a , b) = b
```

We can derive the constructor automatically by adding the constructor keyword to the record definition:

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B
```

What can we do with products? We derive a function curry that turns a function from products into a curried function:

```
curry : (A × B → C) → (A → B → C)
curry f = λ a b → f (a , b)
```

We can also do the reverse and translate a curried function back into one on products:

```
uncurry : (A → B → C) → (A × B → C)
uncurry g = λ x → g (proj₁ x) (proj₂ x)
```

Indeed, the two functions are inverse to each other, that is curry (uncurry g) = g and uncurry (curry f) = f. The first equation follows from what we already know:

```
curry (uncurry g)
= λ a b → uncurry g (a , b)                    Unfolding curry
= λ a b → g (proj₁ (a , b)) (proj₂ (a , b))    Unfolding uncurry
= λ a b → g a b                                Evaluating projections
= g                                            Use η twice.
```

For the 2nd equation we need an η=equality for products which says that (proj₁ x , proj₂ x) = x. This is also called *surjective pairing*:

```
uncurry (curry f)
= λ x → curry f (proj₁ x) (proj₂ x)    Unfolding uncurry
= λ x → f (proj₁ x , proj₂ x)          Unfolding curry
= λ x → f x                            Surjective pairing
= f                                    η for functions.
```

This shows that the type of functions over products and curried functions are equivalent. Most functional programmers prefer to work with curried functions.

2.5.3 Strong sums?

We can try to play a similar game for coproducts. First of all using a product we can uncurry case:

```
case-c : (A → C) × (B → C) → A ⊎ B → C
case-c = uncurry case
```

and hence the inverse of case is:

```
uncase : (A ⊎ B → C) → (A → C) × (B → C)
uncase h = (λ a → h (inj₁ a)) , (λ b → h (inj₂ b))
```

we would hope that the functions are inverse to each other, that is that uncase (case-c f) = f and case-c (uncase g) = g. The first one is we can show:

uncase (case-c f)	
= (λ a → case-c f (inj$_1$ a)) , (λ b → case-c f (inj$_2$ b))	Unfold uncase
= (λ a → proj$_1$ f a , λ b → proj$_2$ f b)	Unfold case-c and case.
= (proj$_1$ f , proj$_2$ f)	η twice.
= f	Surjective pairing

However, the 2nd equality doesn't hold in Agda. Actually we can prove it using propositional equality _≡_ we are going to introduce later. The equality _=_ we have been considering here when talking *about* [5] λ-terms is called definitional equality.

Indeed the η-laws for functions and surjective pairing is already extending the notion of *definitional* equality, they are mainly introduced for convenience. Indeed, the corresponding laws for other types, eg. the natural numbers, would destroy decidability and hence not be very useful for a language like Agda. It is possible, even though not entirely straightforward, to add η-laws for _+_ but this is usually avoided, mainly because it would quickly lead to an exponential blow up of cases and hence be computationally too costly.

2.5.4 Finite Types

The operation _×_ we have introduced handles the binary case of products. We can use it several times to handle more components, as in A × B × C. In practice it is better to use records which have more meaningful field names than proj$_1$ etc.

However, one interesting case not covered by binary products is the nullary product or unit type. In Agda this is denoted as ⊤ and can be defined as a record with no fields:

```
record ⊤ : Set where
  constructor tt
```

It has one element tt : ⊤.

We can play the same game with sums and derive the empty sum, that is a sum with no injections:

```
data ⊥ : Set where
```

The empty sum is the empty type, it is a type with no elements. A version of case for the empty type is quite useful:

```
case⊥ : {A : Set} → ⊥ → A
case⊥ ()
```

[5] Within Type Theory we cannot talk about definitional equality _=_ is is part of the language itself.

We don't need any cases for $\bot$ Agda marks this by () which basically means this case analysis has been intentionally left empty. We observe that we get a function from the empty type into any type. This is ok, because we will never be able to run this function since there isn't an element in the empty type as the name suggests.

We can use the unit type $\top$ and sums $\uplus$ to construct some finite types. For example $\mathsf{Two} : \mathsf{Set}$ is a type with 2 elements:

```
Two : Set
Two = ⊤ ⊎ ⊤
```

Here are the elements:

```
zero₂ : Two
zero₂ = inj₁ tt
one₂ : Two
one₂ = inj₂ tt
```

We can add one more element to Two to construct Three a type with 3 elements:

```
Three : Set
Three = ⊤ ⊎ Two
zero₃ : Three
zero₃ = inj₁ tt
one₃ : Three
one₃ = inj₂ zero₂
two₃ : Three
two₃ = inj₂ one₂
```

However, as already in the case of products in practice it is usually preferable to use **data** to define labelled sums: [6]

```
data Two : Set where
  zero₂ one₂ : Two
data Three : Set where
  zero₃ one₃ two₃ : Three
```

How many elements are in $\mathsf{Two} \uplus \mathsf{Three}$? An example is $\mathsf{inj}_1\ \mathsf{one}_2$. Ok there is the choice between inj_1 and inj_2, if we choose inj_1 we then have a choice between 2 elements and if we choose inj_2 we have a choice of 3 elements, and hence we can choose out of $2 + 3 = 5$ elements.

$\mathsf{inj}_1\ \mathsf{zero}_2$	$\mathsf{inj}_1\ \mathsf{one}_2$	$\mathsf{inj}_2\ \mathsf{zero}_3$	$\mathsf{inj}_2\ \mathsf{one}_3$	$\mathsf{inj}_2\ \mathsf{two}_3$

In general if A has n elements and B has m elements then $\mathsf{A} \uplus \mathsf{B}$ has $\mathsf{m} + \mathsf{n}$ elements.

[6] Note that we can combine constructors with the same type in one line.

How many elements are in Two × Three? A typical example is one_2 , two_3. Here we construct a pair and we have 2 independent choices, for the first components we have 2 options and for the second we have 3 and since these choices are independent we have 2 × 3 = 6 elements.

zero_2 , zero_3	zero_2 , one_3	zero_2 , two_3
one_2 , zero_3	one_2 , one_3	one_2 , two_3

In general if A has n elements and B has m elements then A × B are m × n elements.

What about the function type? How many elements are in Two → Three? Here is a typical example:

```
foo : Two → Three
foo zero₂ = two₃
foo one₂  = one₃
```

We can define any such function by pattern matching: we have 3 choices in each case hence $3 \times 3 = 3^2 = 9$ elements. Now unlike products and sums function types are not symmetric. How many elements are in Three → Two? Here again a typical example:

```
bar : Three → Two
bar zero₃ = zero₂
bar one₃  = one₂
bar two₃  = zero₂
```

There are 3 cases because there are 3 elements in three and in each case we have a choice of 2, hence we have $2 \times 2 \times 2 = 2^3 = 8$ elements. From these examples we can see that in general if A has n elements and B has m elements then A → B has m^n elements.

Indeed, in Mathematics the function type is often written as an exponential, that is A → B is written as B^A.

Previously, we have shown that types A × B → C and A → B → C are equivalent (via curry and uncurry). Writing this in exponential notation we find an old friend from high school:

$$C^{A \times B} = (C^B)^A$$

Similarly, with strong coproducts we would have that (A → C) × (B → C) and A ⊎ B → C are equivalent (via case-c and uncase). If we write + for ⊎ we arrive at another well known equation:

$$C^A \times C^B = C^{A+B}$$

2.6 History

λ-calculus was invented by Alonzo Church who used it in papers about formal logic in the 1930ies, e.g. [**?**]. The story goes that the use of λ arose by accident:

Church used a graphical notation where he connected the binding and the usage of a variable. I imagine he wrote something like this:

But the typesetter (this was not only before latex but before computers) went back and said, *Sorry Professor, I can't do that. But this looks a bit like a* λ*. Is it ok if I use just this?*

I don't know wether this is actually true but it is a nice story. Church realised that one of the problem of his calculus was that he could define two many weird functions, including the so called *fixpoint combinator* which allows to computer the fixpoint of any function by general recursion. But what is the fix point of negation? To avoid this sort of problems Church also invented the typed λ-calculus. [7] Church had two primitive types, ι (the greek letter *iota)* standing for individuals and o (the greek letter *omikron* but it is the same as an o) standing for propositions, motivated by his intended application to formal logic. In my presentation I am not considering any fixed set of base types but a view them as type variables.

The untyped λ-calculus is still useful as a universal programming language which is an alternative to Turing machines. Indeed, when Alan Turing visited Church they realised that Turing machines and λ-calculus compute the same set of computable functions. The generalisation of this observation to all computational mechanism is known as the Church-Turing thesis.

In many papers about typed λ-calculus one starts with the untyped λ-calculus and then introduces a type system on top, as a way of assigning types to untyped terms. I do not follow this view which I think is too syntactic and consider typed terms as primitive. We call this *intrinsic typing*. Yes, it is true if we have to implement λ-terms we will at some point think about untyped terms but we also need to think about strings, characters bits etc and will not talk about these implementation details.

Combinatory logic was actually invented before λ-calculus by Moses Schön-

[7] A reference here is a a paper from 1940: [**?**]

finkel in the 1920ies [8]. Again his interest was in formal logic and he realised that combinators are a nice way to avoid the complexities of variables. This was also the motivation of Haskell Curry who reinvented combinatory logic more or less in the form we know it now. [9] Some people say that the idea of *currying* a function with several arguments which is named after Curry, should have been called *schönfinkeln* but this never caught on - I wonder why? A good overview over the history of λ-calculus and combinatory logic is [**?**].

To understand my presentation of products and sums better (also called coproducts) one needs to know about category theory. Category theory is basically an abstract form of algebra which was introduced by Saunders MacLane and others. [10] Their motivation came mainly from pure Mathematics, algebraic topology, I believe. However, later it was realised that category theory is also extremely useful for theoretical computer science. One important observation was Joachim Lambek's observation that a fundamental notion in category theory, *cartesian closed categories*, corresponds exactly to typed λ-calculus. In the categorical account products are already built-in and function types are characterised by the curry-uncurry-equivalence. One of the beautiful features of category theory is the presence of a mirror (*duality*) and it turns out that the mirror image (the dual) of products are sums or coproducts as we have introduced which are also extremely useful.

At this point you may wonder what the mirror image of functions is? This is a longer story but the short answer is that there isn't one. If you assume that there is one, the theory collapses: all λ-terms are equal.

λ-calculus is important for programming, to be precise in *functional programming*. Indeed the 2nd oldest programming language, LISP (The oldest is FORTRAN) was developed by John McCarthy in the late 1950ies and was inspired by the (untyped) λ-calculus [**?**]. Later types were introduced, famously by Robin Milner in the form of ML which stands for *Metalanguage* as a component of the Edinburgh LCF project (*Logic for computable functions*) [**?**]. Nowadays languages like Haskell (named after Haskell curry) [11] and CAML are modern instances of typed functional programming and Agda's syntax and mechanisms borrow a lot especially from Haskell.

2.7 Exercises

1. Given

$$\begin{aligned} &\mathsf{add3} : \mathbb{N} \to \mathbb{N} \\ &\mathsf{add3}\,\mathsf{x} = \mathsf{x} + 3 \end{aligned}$$

   And for any type A:

[8] If you can read german, you can check out [**?**].
[9] An early reference is See [**?**]
[10] MacLanes book *Categories for the working Mathematician* is still worth a look. [**?**]
[11] A good introduction to Haskell is [**?**]

```
tw : (A → A) → A → A
tw f n = f (f n)
```

Consider tw tw add3 1.

(a) What is its type? Justify your answer.

(b) What is its value? Show every step of the evaluation.

2. Derive a term in combinatory logic (that is only using S and K) for tw using

```
tw : (A → A) → A → A
tw = λ f → λ n → f (f n)
```

3. In section 2.2 we defined identity and composition:

```
id : A → A
id x = x
_∘_ : (B → C) → (A → B) → (A → C)
(f ∘ g) x = f (g x)
```

Using β and η show that the following equations [12] hold definitionally:

$$\mathsf{f} \circ \mathsf{id} = \mathsf{f}$$
$$\mathsf{id} \circ \mathsf{f} = \mathsf{f}$$
$$\mathsf{f} \circ (\mathsf{g} \circ \mathsf{h}) = (\mathsf{f} \circ \mathsf{g}) \circ \mathsf{h}$$

4. Define an operation with the following type:

```
A× : (B → C) → A × B → A × C
```

and show that the following equations hold using β, η and surjective pairing:

$$\mathsf{A}\times\ \mathsf{id} = \mathsf{id}$$
$$\mathsf{A}\times\ (\mathsf{f} \circ \mathsf{g}) = (A \times f) \circ (A \times g)$$

This shows that the operation A × _ on types is (definitionally) a functor.

5. Find elements of the following types using only pure λ-terms and the combinators for sums and products (proj_1 , proj_2 , _,_, inj_1 , inj_2, case).

```
x₀ : (A → B) → (B → C) → (A → C)
x₁ : (A → B) → ((A → C) → C) → ((B → C) → C)
x₂ : A × (B ⊎ C) → (A × B) ⊎ (A × C)
x₃ : (A × B) ⊎ (A × C) → A × (B ⊎ C)
```

[12] These are the equations of a category. We are establishing that types and functions from a category, definitionally.