I am not saying that $g \circ f$ is the same as $g \circ f$, first of all this doesn't *type-check* because if $g : B \to C$ and $f : A \to C$ with all three types being different then only $g \circ f : A \to C$ makes sense but $g \circ f$ doesn't. For this equality to make sense we need to assume that $f, g : A \to A$ and in this case we can compare $f \circ g$ and $g \circ f$ because both have the type $A \to A$. However, it is easy to find a counterexample: let $f : \mathbb{N} \to \mathbb{N}$ be $f\, x \equiv x + 2$ and $g : \mathbb{N} \to \mathbb{N}$ be $g\, x \equiv 2 \times x$ then $f \circ g \equiv \lambda x.f\,(g\, x) \equiv \lambda x.(2 \times x) + 2$ while $g \circ f \equiv \lambda x.g\,(f\, x) \equiv 2 \times (x + 2)$. These functions are different, since for example $(f \circ g)\, 1 \equiv 4$ while $(g \circ f)\, 1 \equiv 6$.

A structure satisfying the three laws $f \circ \mathrm{id} \equiv f$, $\mathrm{id} \circ f \equiv f$ and $f \circ (g \circ h) \equiv (f \circ g) \circ h$ is called a category. Category theory grew out of very abstract algebra and was established since the 1940ies by Saunders MacLane and others. There is a very good fit between category theory and Type Theory but they have different purposes. Type theory is based on some intuitive insight what laws should hold, while category theory has some very effective mechanisms to classify laws and support abstract reasoning.

3.2 Products without multiplication

Products in simple type theory are very easy, given $a : A$ and $b : B$ we can form $(a, b) : A \times B$. How do we construct a function out of a product, that is $f : A \times B \to C$? We just need to say what it does on tuples, e.g. we can define the alternative version of

$\mathrm{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

by defining

$\mathrm{plus}\,(x, y) :\equiv x + y$

This induces a definitional equality $\mathrm{plus}(3, 4) :\equiv 3 + 4$.

We can define two generic functions, which are called *projections* to extract the components of a tuple: $\pi_0 : A \times B \to A$ and $\pi_1 : A \times B \to B$ which are defined:

$\pi_0(x, y) :\equiv x$
$\pi_1(x, y) :\equiv y$

Using only projection there is a different way to define plus:

$\mathrm{plus}\, p :\equiv (\pi_0\, p) + (\pi_1\, p)$

Indeed all functions on $\times$ can be defined using only projections. Hence the defining equations for the projections are the β-equality for products.

Given a tuple $p : A \times B$ with $p :\equiv (a, b)$ what happens if we take it's projections and then put it back together? That is $(\pi_0\, p, \pi_1\, p) : A \times B$. Exactly we end up where we started $(\pi_0\, p, \pi_1\, p) \equiv (a, b)$. Generalising this to any term gives us the η-rule for products:

$(\pi_0\, p, \pi_1\, p) \equiv p$

As for functions you may wonder why we add this law because can't we derive it. Indeed if p is of the form (a, b) (or equal $\equiv$ to it) then it follows from the β-rule. The η rule extends this to expressions which are not of this form, e.g. the identity function on a products $\mathrm{id} : A \times B \to A \times B$ with $\lambda x.x$ is judgementally equal to $\lambda x.(\pi_0\, x, \pi_1\, x)$.

I have now presented two versions of plus, one had the type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and the other $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. In functional programming we prefer the 2nd version, which is called a curried function. Indeed, we can derive some generic operations to curry and uncurry a function and hence to move between the two ways to represent functions with two arguments: $A \times B \to C$ and $A \to B \to C$. We can define two operations which translate between the two representations:

$$\begin{aligned} \mathrm{curry} &: (A \times B \to C) \to (A \to B \to C) \\ \mathrm{uncurry} &: (A \to B \to C) \to (A \times B \to C) \end{aligned}$$

They are not hard to define:

$$\begin{aligned} \mathrm{curry}\, f &:\equiv \lambda x.\lambda y.f\,(x, y) \\ \mathrm{uncurry}\, g &:\equiv \lambda z.g\,(\pi_0\, z))\,(\pi_1\, z) \end{aligned}$$

What happens if we curry and then uncurry? If we just expand the definitions of curry and uncurry in [1]:

$$\mathrm{uncurry}\,(\mathrm{curry}\, f)$$

we get

$$\lambda z.(\lambda x.\lambda y.f\,(x, y))\,(\pi_0\, z)\,(\pi_1\, z)$$

and we can use the β-law for functions to reduce this to

$$\lambda z.f\,(\pi_0\, z, \pi_1\, z)$$

now we have a wonderful opportunity to use the η-law for products to obtain $\lambda z.f\, z$ and by the η-law for functions this is judgementally equal to f - hence we get back to where we started.

Vice versa what happes if we uncurry and then curry? Expanding

$$\mathrm{curry}\,(\mathrm{uncurry}\, g)$$

we obtain

$$\lambda x.\lambda y.\,(\lambda z.g\,(\pi_0\, z))\,(\pi_1\, z))\,(x, y)$$

as before we use the β-rule for functions to reduce this to

$$\lambda x.\lambda y.g\,(\pi_0\,(x, y)))\,(\pi_1\,(x, y)))$$

[1] Actually we apply the β-rule for functions

and now applying the definition of the projections, that is the β-law for products we can simplify this to $\lambda x.\lambda y.g\,x\,y$ and now after applying the η-law for functions twice we are left with g, hence again where we started.

We say that the operations curry and uncurry are inverse to each other. After all this shouldn't come as a surprise because I was saying that the two representations of functions with two arguments are equivalent and this equivalence is witnessed by these two inverse operations between the two representations. In the language of category this equivalence is used to express the relationship between $\times$ and $\to$.

3.3 Coproducts : Products in the mirror

So what is the *mirror image* of products? They are called coproducts and the *co-* is the categorical terminology for mirror. Given types A and B we construct a new type $A + B$ whose elements are $\mathrm{inl}\,a : A + B$ for $a : A$ and $\mathrm{inr}\,b : A + B$ for $b : B$. These operations are called *injections*, inl stands for inject left and inr for inject right. In set-theory this operation is called *disjoint union* and it is defined using the $\cup$ operation, that is

$$x + y = \{0\} \times x \cup \{1\} \times y$$

assuming now that x, y are sets and using the representations of $0, 1, \times$ introduced in the previous section.

$+$ is different from $\cup$ in that it labels the arguments and hence makes them disjoint. To see the difference, consider the type of booleans with $\mathrm{true}, \mathrm{false} : \mathrm{Bool}$ corresponding to the set $\mathrm{Bool} = \{\mathrm{true}, \mathrm{false}\}$. Now in set theory $\mathrm{Bool} \cup \mathrm{Bool} = \mathrm{Bool}$ since multiplicity of elements doesn't matter. On the other hand in Type Theory $\mathrm{Bool} + \mathrm{Bool}$ has four elements, namely

$$\mathrm{inl}\,\mathrm{false}, \mathrm{inl}\,\mathrm{true}, \mathrm{inr}\,\mathrm{false}, \mathrm{inr}\,\mathrm{true} : \mathrm{Bool} + \mathrm{Bool}$$

It should also be no surprise that this type has exactly $4 = 2 + 2$ elements, since the coproduct of two finite types has the sum of numer of elements of the two types. Hence the use of the $+$-symbol, using the same analogy as for $\times$.

$+$ is a sensible operation in Type Theory while $\cup$ isn't. $\cup$ exposes the internal encoding used, e.g consider $\mathbb{N} \cup \mathrm{Bool}$ in set theory: if we represent $\mathrm{Bool} = \{0, 1\}$ then $\mathbb{N} \cup \mathrm{Bool} \subseteq \mathbb{N}$. However, if we use a different encoding, lets say $\mathrm{Bool} = \{(0, 0), (1, 1)\}$ then this is not the case. This is bad because it exposes the internal implementation of Bool, and this hinders abstraction. On the other hand there is not problem with $+$, $\mathbb{N} + \mathrm{Bool}$ is always the same type upto equivalence, independent of the choice of representation of Bool and $\mathbb{N}$. This is the basic intuition behind the univalence principle which will discuss later.

To summarize: $\cup$ is not a type-theoretic operation because it is too intensional. On the other hand $+$ is an extensional operation but it is not defined via $\cup$ as in set theory. Hence I think that the name *disjoint union* is misleading and I prefer the term *coproduct* which expresses the categorical duality to products.

How to define a function out of a coproduct? As an example consider $f : \mathrm{Nat} + \mathrm{Nat} \to \mathrm{Nat}$ which is defined by saying how it behaves for left and how for right injections:

$$f\,(\mathrm{inl}\,n) :\equiv 2 \times n$$
$$f\,(\mathrm{inr}\,n) :\equiv 2 \times n + 1$$

This function maps the left injections to the even numbers and the right injections to the odd numbers. Indeed it is part of an equivalence between $\mathbb{N} + \mathbb{N}$ and $\mathbb{N}$ but we don't yet have the means to express let alone prove it.

We can also devise a generic operations to define functions out of coproducts which we call case because it performs a form of case analysis. Given types A, B, C we define $\mathrm{case}_{A,B,C} : (A \to C) \to (B \to C) \to (A + B \to C)$ by

$$\mathrm{case}\,f\,g\,(\mathrm{inl}\,a) :\equiv f\,a$$
$$\mathrm{case}\,f\,g\,(\mathrm{inr}\,b) :\equiv g\,b$$

As an example we could have defined the function f above using case:

$$f :\equiv \mathrm{case}\,(\lambda n.2 \times n)\,(\lambda n.2 \times n + 1)$$

The defining equations for case are the β-laws for coproducts. What about the η-rule? We may think that

$$\mathrm{case}\,\mathrm{inl}\,\mathrm{inr}\,x \equiv x$$

is enough but it turns out that this isn't strong enough to prove many equalities on coproducts. As a example consider $\mathrm{swap} : A + B \to B + A$:

$$\mathrm{swap}\,(\mathrm{inl}\,a) :\equiv \mathrm{inr}\,a$$
$$\mathrm{swap}\,(\mathrm{inr}\,b) :\equiv \mathrm{inl}\,b$$

Now we would like to show that swapping twice doesn't do anything:

$$\mathrm{swap}\,(\mathrm{swap}\,x) \equiv x$$

However, the β and η-rules for products are not strong enough to do this. It is possible to add additional equations to fix this and we have shown that the resulting theory is decidable, that is can be checked by a computer, but this wasn't so easy. Even worse the algorithm is quite inefficient and not really useful in practice.

In what sense are coproducts the mirror image of products? From the perspective of Category Theory products are characterized by an equivalence of $(C \to A) \times (C \to B)$ and $C \to A \times B$ given by the following functions:

$$\mathrm{pair} : ((C \to A) \times (C \to B)) \to C \to A \times B$$
$$\mathrm{pair}\,f\,g\,c \equiv (f\,c, g\,c)$$
$$\mathrm{unpair} : (C \to A \times B) \to (C \to A) \times (C \to B))$$
$$\mathrm{unpair}\,h :\equiv (\lambda c.\pi_0\,(h\,c), \pi_1\,(h\,c))$$

Just using the β and η-laws for products and functions we can show that these two functions re inverse to each other, i.e. $\mathrm{unpair}\,(\mathrm{pair}\,x) \equiv x$ and $\mathrm{pair}\,(\mathrm{unpair}\,x) \equiv x$).

Now for coproducts we have another equivalence which can be obtained by turning all the arrows around. That is $(A \to C) \times (B \to C)$ and $A + B \to C$ are equivalent. From left to right this is just a curried version of case and other one uncase : $(A + B \to C) \to (A \to C) \times (B \to C)$ which is defined using the injections:

$$\mathrm{uncase}\, f :\equiv (\lambda a.f\,(\mathrm{inl}\, a), f\,(\lambda b.\mathrm{inr}\, b))$$

We can show that $\mathrm{case}\,(\mathrm{uncase}\,x) \equiv x$ but the other equality $\mathrm{uncase}\,(\mathrm{case}\,x) \equiv x$ corresponds exactly to the strong theory which we have discussed previously. However, once we introduce the equality type we can show that this equality holds upto provable equality $\mathrm{case}\,(\mathrm{uncase}\,x) = x$ which is sufficient. Indeed as soon as we move on to natural numbers in the next section the strong theory is undecidable, i.e. it cannot be implemented on a computer.

So, indeed the mirror we are using is a bit broken. If we want to be principled we really should reject all η-laws. The η-laws are not really definitional equalities, they are extensional equalities which we can implement on a computer. The coproducts are a borderline case which is implementable but not practically. Hence should we just get rid of all the η-laws. Maybe but they are actally convenient to have in implementations of Type Theory. We will get back to this question later.

I had briefly discussed the laws of a monoid, maybe you remember that 1 is the neutral element for multiplication and 0 is the neural element for addition. Indeed we have types corresponding to 1 and 0 and they are called unit and empty type. We have an element of unit $() : 1$. This notation should remind you in an empty tuple because 1 is in a way the extreme case of a product. The empty type has, as the name says, no element. To define a function out of the empty type like $f : 0 \to A$ we have to give no defining equation because this function will never be applied to an argument. Hence there is no work involved defining functions out of the empty type.

Getting back to η-laws the η-rule for the empty type has the particular wierd consequence that any two terms are equal as soon as we can construct an element of the empty type. This is maybe ok for simple types but it is impossible to decide (on a computer) wether the empty tye is inhabited as soon as we have got dependent types. Hence while we could possibly have η-laws for binary coproduct, we can't implement them for the 0-ary coproduct that is the empty type.

Using 1 and + we can define Bool $:\equiv 1 + 1$ with false $:\equiv$ inl() and true $:\equiv$ inr().

3.4 Propositions as types: propositional logic

In the introduction I said that in Type Theory we explain the meaning of propositions by assigning to each proposition the type of reasons or proofs that we accept this proposition. This idea is associated with Haskell Curry (after whom the programming language Haskell is named) and William A. Howard and hence is called the Curry-Howard correspondence. We will look at this idea first for propositional logic which is the part of predicate logic which corresponds to simple types. That is what we get looking only at $\wedge$ (and), $\vee$ (or), $\neg$ (negation), $\rightarrow$ (if-then) and so on but ignoring for the moment predicates and relations and $\forall$ (for all) and $\exists$ (exists). Propositional logic is a bit strange while on the one hand it is simpler than predicate logic on the other hand it is a bit useless because we can't say anything interesting in Mathematics just using propositional logic. Instead we talk about arbitrary propositions similar as I talked about arbitrary types in the previous sections. Hence while propositional logic is a bit easier than predicate logic it always is a bit abstract. To overcome this I find it always useful to think about specific examples from everyday life, like *the sun shines* and *we go to the zoo* and so on but you shouldn't think that propositional logic is mainly about everyday reasoning.

What is the type of reasons to believe that *the sun shines* **and** *we go to the zoo*? We better have reasons to believe both. Writing P, Q for arbitrary propositions (but think about the two I just gave) we say that the reasons to believe in $P \wedge Q$ are pairs of reasons that is $P \times Q$, we identifying a proposition with the reasons to believe in it means also that we say $P \wedge Q \equiv P \times Q$. If you say *the sun shines* **or** *we go to the zoo* you either have a reason to believe in one or the other proposition and this corresponds to using coproducts: $P \vee Q \equiv P + Q$. I think most interesting is the translation of $\Rightarrow$: The reason to believe $P \Rightarrow Q$ is a function that transforms reasons for P in reasons for Q. Once I have such a function and you give me a reason for Q I just apply it and have a reason for Q. Hence we identify $P \Rightarrow Q \equiv P \rightarrow Q$.

How do we translate $\neg P$? $\neg P$ means that there is no reason to believe P, hence we can translate $\neg P$ as *If P then False*, that is $\neg P \equiv P \rightarrow \text{false}$. We don't really say false to mean impossible in natural language. Hence I suggest to read $P \rightarrow \text{false}$ as *If P then pigs have wings.* We translate false with the empty type, because there is no reason to believe that pigs have wings: $\text{false} \equiv 0$. Imagine a man asking a woman to marry her and she replies: *If I marry you then pigs have wings.* I would take this as a *no*.

There are many possible choices for true because any type with an element would do. The simplest choice is 1 the type with one element: $\text{true} \equiv 1$. Finally we translate logical equivalence $P \Leftrightarrow Q$ as implications in both directions $P \Leftrightarrow Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

How can we use this translation to see wether a proposition is a tautology? For example $P \wedge (Q \vee R)$ is logically equivalent to $(P \wedge Q) \vee (P \wedge R)$, this is similar to the law of distributivity in arithmetic $x \times (y + z) = x \times y + x \times z)$.

$$
\begin{aligned}
\text{true} &:\equiv 1 \\
P \wedge Q &:\equiv P \times Q \\
\text{false} &:\equiv 0 \\
P \vee Q &:\equiv P + Q \\
P \Rightarrow Q &:\equiv P \to Q \\
\neg P &:\equiv P \Rightarrow \text{false} \\
&\equiv P \to 0 \\
P \Leftrightarrow Q &:\equiv (P \Rightarrow Q) \wedge (Q \Rightarrow Q) \\
&\equiv (P \to Q) \times (Q \to P)
\end{aligned}
$$

Figure 3.1: Propositions as types for propositional logic

We derive both directions separately:

$$
\begin{aligned}
&\text{lr} : P \times (Q + R) \to (P \times Q) + (P \times R) \\
&\text{lr}\,(x, \text{inl}\,y) :\equiv \text{inl}\,(x, y) \\
&\text{lr}\,(x, \text{inr}\,z) :\equiv \text{inr}\,(x, z)
\end{aligned}
$$

$$
\begin{aligned}
&\text{rl} : (P \times Q) + (P \times R) \to P \times (Q + R) \\
&\text{rl}\,(\text{inl}(x, y)) :\equiv (x, \text{inl}\,y) \\
&\text{rl}\,(\text{inr}(x, z)) :\equiv (x, \text{inr}\,y)
\end{aligned}
$$

$$
\begin{aligned}
&\text{distr} : P \times (Q + R) \leftrightarrow (P \times Q) + (P \times R) \\
&\text{distr} \equiv (lr, rl)
\end{aligned}
$$

To avoid repeating formulas in the translation of $\Leftrightarrow$, I write $P \leftrightarrow Q$ for $(P \to Q) \times (Q \to P)$.

The reason to accept that $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$ is a tautology using propositions as types is quite different from the classical explanation which uses a truthtable:

P	Q	R	$P \wedge (Q \vee R)$	$(P \wedge Q) \vee (P \wedge R)$	$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$
false	false	false	false	false	true
false	false	true	false	false	true
false	true	false	false	false	true
false	true	true	false	false	true
true	false	false	false	false	true
true	false	true	true	true	true
true	true	false	true	true	true
true	true	true	true	true	true

It is clear that either side is true if P and either Q or R are true. The truthtable for $\Leftrightarrow$ is very simple, it is true if both inputs agree., false otherwise. Hence we always obtain true, no matter what the inputs are: this is the definition of a tautology.

Here we have two different ways to observe that

$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$$

is a tautology: either by writing a program which shows us that we will believe it no matter what is the input and the other that it is true no matter what is the truth of the inputs.

However, this is not always the case. In section 2.4 I showed how in classical logic connectives can be defined from each other, so for example $P \wedge Q$ can be defined as $\neg(\neg P \vee \neg Q)$. One ingredient was the de Morgan formula $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q)$. However, in propositions as types only one direction is valid, namely $\neg P \vee \neg Q \Rightarrow \neg(P \wedge Q)$ which translates to

$$\mathrm{demorgan} : (P \to 0 + Q \to 0) \to ((P \times Q) \to 0)$$

The idea is to use reasoning by cases on the assumption $P \to 0 + Q \to 0$: in either case we can derive a contradiction if we assume $P \times Q$. As a program this looks like this:

$$\mathrm{demorgan}\,(\mathrm{inl}\,f)\,(x,y) :\equiv f\,x$$
$$\mathrm{demorgan}\,(\mathrm{inr}\,g)\,(x,y) :\equiv g\,y$$

However, there is no program for the other direction $((P \times Q) \to 0) \to (P \to 0 + Q \to 0)$. To define such a function we assume as input $x : (P \times Q) \to 0$ we need to construct an element of $P \to 0 + Q \to 0$, in particular we have to decide wether to use inl or inr. However, there is no information available to make that choice: all we know is that having both P and Q is inconsistent but not that one of them is and then which one?

However, when we draw the truthtable it is clear that $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q)$ is a tautology:

P	Q	$\neg(P \wedge Q)$	$\neg P \vee \neg Q$	$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$
false	false	true	true	true
false	true	true	true	true
true	false	true	true	true
true	true	false	false	true

In this case the propositions as types view and the classical truth based view diverges. It is interesting to note that other de Morgan formula $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$ is valid in propositions as types and in the classical view. The reason is that both formulas $\neg(P \vee Q)$ and $\neg P \wedge \neg Q$ are *negative*, that is they contain no information. hence the problem where we had to make our mind up for the de Morgan rule $\neg(P \wedge Q) \Rightarrow \neg P \vee \neg Q)$ doesn't arise.

The difference between the intuitionistic (just a another word for the propositions as types view) and the classical view can be brought down to a basic principle: the *law of excluded middle* which expresses that every proposition is either true or false, which can be translated to $P \vee \neg P$. The truth table for this one should be obvious:

P	$\neg P$	$P \vee \neg P$
false	true	true
true	false	true

On the other hand how would we prove $P + (P \to 0)$? Again we have to make up our mind wether to use inl or inr, but which one? P may in general be a proposition for which w simply don't know wether it is true or false. But even worse: we would expect a uniform solution which doesn't depend on P and this means we have to make up our mind in advance wether all propositions are true or not.

The principle of the excluded middle describes exactly the difference between classical and intuitionistic logic: once we assume it we can derive all classical tautologies. Maybe the following question comes to your mind: we cannot prove $P \vee \neg P$ maybe we can find a particular proposition Q for which we can prove $\neg(Q \vee \neg Q)$? This turns out is impossible and we can even do better and show that the negation holds for all propositions $\neg\neg(P \vee \neg P)$, that is we can show the that it is for any proposition P it is not the case that the excluded middle doesn't hold. This clearly implies that we cannot have a counterexample because if we had one we can put it together with the proof I am just about to present and derive a contradiction.

To convince you: applying our translation we need to come up with nnem : $((P + P \to 0) \to 0) \to 0$. Let's do this step by step: we assume $f : (P + P \to 0) \to 0$ and we want to derive an element $?_0 : 0$. The only way to construct an element of the empty type is by using our assumption $?_0 \equiv f\,?_1 : 0$ where $?_1 : P + P \to 0$. But now we have to make a choice we have to go left or right. How can we decide? Going left we would be stuck immediately hence let's see what happens if we go right: $?_1 :\equiv \mathrm{inr}\,?_2$ where $?_2 : P \to 0$. Now there is only one way to implement $?_2$ we introduce a function using λ: $?_2 :\equiv \lambda x.?_3$. Now $?_3 : 0$ – are we running in a circle? But note that we have a new assumption $x : P$ which we can use to define $?_3$. Again the only thing we can use is f, that is $?_3 :\equiv f\,?_4$ with $?_4 : P + P \to 0$. We went right last time, this time we go left: $?_4 :\equiv \mathrm{inl}\,?_5$ with $?_5 : P$. But this is easy because we can just set $?_5 :\equiv x$ and voila, we are done. To summarize:

$$\mathrm{nnem}\, f :\equiv f\,(\mathrm{inr}\,(\lambda x.f\,(\mathrm{inl}\, x)))$$

This proof can be illustrated by the following story which is a slight modification on one that I have heard from Peter Selinger. The story shows that double negation is the same as time travel:

> Once upon a time there was a poor king who had no gold. At this time it was unknown wether ipads existed and the King was keen

> to find out. He offered the hand of this daughter to anybody who could answer this question. Then a magician arrived in the court who seemed to have an answer: he promised the king that he would either get him an ipad or a machine that would turn an ipad into gold. In truth the magician didn't have an ipad and he certainly had no gold but he had a time machine. So once the King agreed, the magician said that he had indeed a machine which turns an ipad into gold which was the time machine with his friend sitting inside. Now the King was satisfied and the magician married the daughter. However, te next day an ipad appeared at court. As soon as the greedy King put the ipad into the machine, his friend took it and travelled back in time. Now the magician was able to fullfil his promise by presenting the ipad and he could still marry the daughter.

I leave it to you to figure out the relationship between the fairy tale and the proof of nnem.

There is an alternative to excluded middle when we want to add classical reasoning to propositions as types. This is based on the double negation operator we have just encountered: the principle of indirect proof says that to show P is is enough to show that P cannot be false, that is $\neg\neg P \to P$ Indeed, this is a very common situation in a classical proof where to prove P you assume $\neg P$ and try tp derive a contradiction. $\neg\neg P \to P$ as not provable using propositions as types which reflects the intuition that indirect evidence is not as good as direct evidence. E.g. to know that the key cannot be outside the house is not as good as actally having it. However, we can derive $\neg\neg P \to P$ from assuming excluded middle $h : P + P \to 0$ we assume $f : (P \to 0) \to 0$ now to show $? : P$ we do case analysis over h: either $h \equiv \mathsf{inl}\, x$ with $x : P$ and in this case we are done with $? \equiv x$ or we have $h \equiv \mathsf{inr}\, g$ with $g : P \to 0$ and in this case we can derive $f\, g : 0$ and now using that we can prove everything form false $\mathsf{efq} : 0 \to P$ we have $? \equiv \mathsf{efq}\,(f\, g)$ To summarize we define $\mathsf{em2ip} : (P + P \to 0) \to ((P \to 0) \to 0) \to P$ as follows

$$\mathsf{em2ip}\, f\, h :\equiv \mathsf{case}\,(\lambda x.x)\,(\lambda g.\mathsf{efq}\,(f\, g))$$

This also works the other way around but not if we keep the proposition P the same. To prove $P \vee \neg P$ from indirect proof we use the fact that we have just shown $nnem : \neg\neg(\neg P \vee P)$ for any P and hence we only have to apply indirect proof for $\neg P \vee P$ to derive it. Hence the two principles: excluded middle and indirect proof are equivalent and both of them enable us to prove all classical tautologies. However, we don't really want to assume either of them globally but sometimes we may be able to prove them in specific instances.

3.5 Counting for dummies

The natural numbers are based on the basic intuition of counting. As a computer scientist I prefer to start counting with 0 this is also useful as the answer to the