Just using were elementary reasoning we can show that

$\mathrm{el}\, R\, R \leftrightarrow (\mathrm{el}\, R\, R) \to 0$

And it follows from propositional logic that $\neg(P \Leftrightarrow \neg P)$ from simple reasoning in propositional logic. But this means that the empty type in inhabited.

Where did we actually use **Type** : **Type**? We didn't explicitly but implicitly when we introduced a constructor makeTree whose argument is a type again.

So, if **Type** : **Type** doesn't work what is now the type of **Type**? To avoid the problem we introduce not one namely infinitely many universes.

$\mathbf{Type}_0 : \mathbf{Type}_1 : \mathbf{Type}_2 : \ldots$

Hence the answer depends on the level, if we ask about the first universe $\mathbf{Type}_0$ then its type is $\mathbf{Type}_1$ and what is the type of $\mathbf{Type}_1$? Right is is $\mathbf{Type}_2$. And so on.

The numbers $0, 1, 2, \ldots$ which as the index of types are not our natural numbers (even though the look very much like them) because otherwise we would have to make our mind up was is the type of a function that assigns to a number a type and what is the type of this function?

All our usual garden variety types like $\mathbb{N}, \mathrm{Bool}, \mathrm{List}\,\mathbb{N}, \mathbb{N} \to \mathrm{Bool}, \mathrm{InfTree}, \ldots$ are elements of $\mathbf{Type}_0$, but they are also elements of all higher universes - this is called cummulativity. The thing that is new in $\mathbf{Type}_1$ is $\mathbf{Type}_0$ itself. We say that $\mathbf{Type}_0$ is larger then all the types in $\mathbf{Type}_0$ because if there would be a type in $\mathbf{Type}_0$ which is equivalent to $\mathbf{Type}_0$ we can derive the paradox. There are more new types in $\mathbf{Type}_1$ for example $\mathrm{List}\,\mathbf{Type}_0$ or $\mathbf{Type}_0 \to \mathbb{N}$. This story continues: the new types in $\mathbf{Type}_2$ are $\mathbf{Type}_1$ and all the types which can be built from it.

While this solves the problem with **Type** : **Type** it does introduce a lot of bureaucracy: for example we have many copies of List namely $\mathrm{List}_0 : \mathbf{Type}_0 \to \mathbf{Type}_0$ and $\mathrm{List}_1 : \mathbf{Type}_1 \to \mathbf{Type}_1$ etc. This is not covered by cummulativity which only tells us that $\mathrm{List}_0\,\mathbb{N} : \mathbf{Type}_1$. What is the best way to deal with this problem is a research question: while there are a number of candidates and implementations there is no general agreement what is the best way. On paper we can be lazy and adopt the convention that we just pretend that we had **Type** : **Type** but then check that there is a consistent assignment of indices to each occurence of **Type**. This is close to what the Coq system is doing but there are cases where the inference algorithm is too weak.

4.4 Propositions as Types: Predicate logic

I am now going to deliver on the promise to extend the propositions as types explanation to predicate logic using dependent types. Actually it is rather straightforward and I couldn't avoid giving away the secret in the last section - maybe you noticed. We translate $\forall$ with Π-types. That is for example the statement [3] that

$\forall x : \mathbb{N}.x + 0 = x$

is translated into a type

$\Pi x : \mathbb{N}.x + 0 = x$

That is a reason is a function that assigns to any natural number $n : \mathbb{N}$ a reason that $x + 0 = x$. I haven't yet explained what equality is as a type but I will do this soon for the moment I will appeal on some basic intuition about equality. How would an element of this type look like? I am defining $f : \Pi x : \mathbb{N}.x + 0 = x$ using primitive recursion:

$$\begin{aligned} f\,0 &:\equiv \mathrm{refl}\,0 \\ f\,(\mathrm{suc}\,n) &:\equiv \mathrm{resp}\,\mathrm{suc}\,(f\,n) \end{aligned}$$

Some explanation is required I am using $\mathrm{refl} : \Pi n : \mathbb{N}.n = n$ as the proof of reflexivity and

$\mathrm{resp} : \Pi_{A,B:\mathbf{Type}}\Pi f : A \to B.\Pi_{a,a':A}a = a' \to f\,a \to f\,a'$

as a proof that any function respects equality. That is in particular

$\mathrm{resp}\,\mathrm{suc} : \Pi m, n : \mathbb{N}.m \equiv n \to \mathrm{suc}\,m = \mathrm{suc}\,n$

What does this function actually do? What is for example $f\,5$? Now that is easy $f\,5 \equiv \mathrm{refl}\,5$ since $5 + 0 \equiv 5$ and hence $\mathrm{refl}\,5 : 5 + 0 \equiv 5$. So f is just a constant function returning refl! Could we not have just written

$f\,n :\equiv \mathrm{refl}\,n$?

No, because $n + 0$ is not by definition the same as n, and hence this definition of the function doesn't obviously typecheck. The more complicated version above does, but this requires some thought: the result of $\mathrm{resp}\,\mathrm{suc}\,(f\,n)$ has the type $\mathrm{suc}\,(n + 0) = \mathrm{suc}\,n$ but we needed an element of $(\mathrm{suc}\,n + 0 = \mathrm{suc}\,n)$ In this case however, we can exploit the definition of $+$ which tells us that $(\mathrm{suc}\,n) + m \equiv \mathrm{suc}\,(n + m)$. The definition of f uses primitive recursion for dependent types which generalized iter, but really this is a proof by induction. We will look into this relationship soon.

Similarly we translate $\exists$ with Σ-type. For example we translate

$\exists x : \mathbb{N}.x = x + x$

into

$\Sigma x : \mathbb{N}.x = x + x$

[3] To be consistent I am using here predicate logic with types whereas previously we assumed that there always is only one type we talk about.

That is a reason for an existential statement is a pair: the first component is the actual element we use (sometimes called the witness) and the 2nd component is the reason that this element satisfies the predicate. In this case it is easy to write down a proof:

$(0, \mathrm{refl}\, 0) : \Sigma x : \mathbb{N}.x = x + x$

Here I am using that $\mathrm{refl}\, 0 : 0 = 0 + 0$ since $0 + n \equiv n$ because that is the way we have defined $+$.

What is a predicate? Predicates are properties, since we now work with types we can only define properties of elements of a given type. For example Even is the property of a natural number to be even. So given a natural number $n : \mathbb{N}$ we can construct $Even\, n$ which is a proposition, that is using propositions as types, a type. Hence even is a function from natural numbers to types:

$\mathrm{Even} : \mathbb{N} \to \mathbf{Type}$

That s using the terminology from the previous sections: Even is a dependent type. Indeed, we need depndent types to extend propositions as types to predicate logic, we use dependent types to interpret predicates. This also works for relations using currying for any given type $A : \mathbf{Type}$ we have $- =_A - : A \to A \to \mathbf{Type}$ the equality relation. Or more generally $- = - : \Pi_{A:\mathbf{Type}} A \to A \to \mathbf{Type}$ since equality works polymorphically for any type.

Using Σ-types we can make sense of the comprehension notation of set theory. If we have a type $A : \mathbf{Type}$ and a predicate P over A that is $P : A \to \mathbf{Type}$ we can represent $\{x : A \mid P\, x\}$ as $\Sigma x : A.P\, x$ so an element is an element of A together with a proof that it satsifies the predicate. For example

$\{x : \mathbb{N} \mid \mathrm{Even}\, x\} \equiv \Sigma x : \mathbb{N}.\mathrm{Even}\, x$

represents the type of even numbers as pairs of a number and a proof that this number is even. However, there can be a mismatch here: if we have two ways to prove that for example 2 is even then there are two different versions of 2 in $\{x : \mathbb{N} \mid \mathrm{Even}\, x\}$. This is undesirable and we should demand that there is at most one proof that a certain number is even - in this particular instance this is not difficult to achieve.

Earlier I promised that we can make sense of $\cap$ and $\cap$ and other operations from set theory. As explained earlier they are not operation on types but they can be understood as operations on predicates. That is given a type $A : \mathbf{Type}$ this is often called a universe in set theory but should not be confused with type theoretic universes. Now given two predicates $P, Q : A \to \mathbf{Type}$ we can define new predicates:

$P \cap Q, P \cap Q : A \to \mathbf{Type}$
$P \cap Q :\equiv \lambda x.P\, x \wedge Q\, x$
$P \cup Q :\equiv \lambda x.P\, x \vee Q\, x$

We can also make sense of $P \subseteq Q$ which is a proposition interpreted as a type:

$$P \subseteq Q :\equiv \Pi x : A.P\,x \to Q\,x$$

This interpretation of the operations covers most of matehmatical practice where $\cup, \cap, \subseteq$ are applied to subsets, i.e. predicates, of a given set.

We can use the interpretation of $\forall$ and $\exists$ to verify tautologies of predicate logic. Maybe you remember the slightly surprising tautology

$$(\forall x.\Phi(x) \Rightarrow p) \Leftrightarrow (\exists x.\Phi(x) \Rightarrow p)$$

from section 2.4 which I illustrated using student performance and lecturer happiness. Instead of this sketch of a logical justification we can now construct functions which show us that the reasons can be translated both ways. Let's fix a type A : **Type** the statement becomes:

$$(\forall x : A.Q\,x \Rightarrow P) \Leftrightarrow (\exists x.Q\,x \Rightarrow P)$$

where Q is a predicate over A that is $Q : A \to \textbf{Type}$ and P is a proposition which we represent as a type $P : \textbf{Type}$. Propositions as types here means we replace the symbols:

$$(\Pi x : A.Q\,x \Rightarrow P) \Leftrightarrow ((\Sigma x : A.Q\,x) \Rightarrow P)$$

that means we need to construct 2 functions:

$$f : (\Pi x : A.Q\,x \Rightarrow P) \to ((\Sigma x : A.Q\,x) \Rightarrow P)$$
$$f\,h :\equiv \lambda y.h\,(\pi_0\,y, \pi_1\,y)$$

and for the other direction:

$$g : ((\Sigma x : A.Q\,x) \Rightarrow P) \to (\Pi x : A.Q\,x \Rightarrow P)$$
$$g\,k :\equiv \lambda x, q.k\,(x, q)$$

If you have a dejavu when looking at this definitions you are right. In section 3.2 we defined the functions curry and uncurry:

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

and their definitions exactly match g and f. This should be no surprise: all we have done is to replace the product $\times$ by a Σ-type and the function $\to$ by a Π-type. That is we replaced the non-dependent version of an operation on types by the dependent one. And now we can use this to justify a tautology from predicate logic.

We can use propositions as types to extend simple Type Theory from the previous chapter by reasoning using predicate logic. Indeed, this is the way the Coq system is often used in practice. It seems to me that it is a shame using dependent types only in this fashion but indeed a much tighter integration is often possible and desirable. As an example let's verify that the equality of natural numbers is decidable. As a first step we could implement a decision function $\mathrm{eq} : \mathbb{N} \to \mathbb{N} \to \mathrm{Bool}$ which can be defined as follows:

$$\begin{aligned} \mathrm{eq}\,0\,0 &:\equiv \mathrm{true} \\ \mathrm{eq}\,0\,(\mathrm{suc}\,n) &:\equiv \mathrm{false} \\ \mathrm{eq}\,(\mathrm{suc}\,m)\,0 &:\equiv \mathrm{false} \\ \mathrm{eq}\,(\mathrm{suc}\,m)\,(\mathrm{suc}\,n) &:\equiv \mathrm{eq}\,m\,n \end{aligned}$$

The strategy is that we look at all combinations of successor and zero that are possible: if both are 0 then there are equal; if one is zero and the other is a successor then they are obviously not equal and if both are successor we recursively compare the predeccessors. While this is obviously correct we should be paranoid and demand a proof anyway, which could be given by proving

$$\forall m, n : \mathbb{N}.\mathrm{eq}\,m\,n \Leftrightarrow m = n$$

using the propositions as types translation. However, there is a nicer way using dependent types directly. The type of eq above wasn't very informative: it only told us that we have a function from two natural numbers to bool. Using dependent types we can be more explicit and replace bool by a more informative type. That is for any natural numbers $m, n : \mathbb{N}$ we want to prove $(m = n) \vee \neg(m = n)$. Indeed, this is an instance of the law of excluded middle, which doesn't hold in general but it does in this case, witnessing decidability. That is replacing $\vee$ with $+$ and $\neg$ with $\to 0$ we get

$$\mathrm{eq} : \Pi m, n : \mathbb{N}.(m = n) + (m = n) \to 0$$

If we can prove an instance of the law of the exclude middle we say that the coresponding proposition has beed decided, that is we have established wether it is true or false. If we have a predicate we say that this predicate is decidable if we can decide all its instances, and similar for relations. To avoid writing long and unreadable formulas below it is useful to define

$$\begin{aligned} &\mathrm{Dec} : \mathbf{Type} \to \mathbf{Type} \\ &\mathrm{Dec}\,P :\equiv P + (P \to 0) \end{aligned}$$

where we read Dec as decided. Then the type of eq becomes

$$\mathrm{eq} : \Pi m, n : \mathbb{N}.\mathrm{Dec}\,(m = n)$$

Now let me sketch the definition of the depdnently typed eq only leaving out the bits where I need to use equality reasoning which I haven't yet introduced.

The general structure resembles the previous definition with Bool:

$$\begin{aligned}
\mathrm{eq}\,0\,0 &:\equiv \mathrm{inl}\,(\mathrm{refl}\,0)\\
\mathrm{eq}\,0\,(\mathrm{suc}\,n) &:\equiv \mathrm{inr}\,\mathrm{noConf}\,n\\
\mathrm{eq}\,(\mathrm{suc}\,m)\,0 &:\equiv \mathrm{inr}\,(\lambda p.\mathrm{noConf}\,m\,(\mathrm{sym}\,p))\\
\mathrm{eq}\,(\mathrm{suc}\,m)\,(\mathrm{suc}\,n) &: \mathrm{eqAux}\,(\mathrm{eq}\,m\,n)
\end{aligned}$$

where we use

$$\begin{aligned}
\mathrm{noConf} &: \Pi n:\mathbb{N}.(0 = \mathrm{suc}\,n) \to 0\\
\mathrm{sym} &: \Pi_{A:Type}\Pi_{a,b:A} a = b \to b = a\\
\mathrm{eqAux} &: \Pi_{m,n:\mathbb{N}}\mathrm{Dec}\,(m = n) \to \mathrm{Dec}\,(\mathrm{suc}\,m)\,(\mathrm{suc}\,n)
\end{aligned}$$

noConf stands for *no confusion* and corresponds to the principle in Peano Arithmetic that 0 is not equal to any successor. In the second case I need the same inequality but the other way around which can be derived using a proof sym that equality is symmetric. Here is the definition of eqAux using yet another auxilliary function:

$$\begin{aligned}
\mathrm{eqAux}\,(\mathrm{inl}\,p) &:\equiv \mathrm{inl}\,(\mathrm{resp}\,\mathrm{suc}\,p)\\
\mathrm{eqAux}\,(\mathrm{inr}\,h) &:\equiv \mathrm{inr}\,(\lambda q.h\,(\mathrm{injSuc}\,q)
\end{aligned}$$

What is happening? eqAux preserves the injection becuase clearly the equality of m and n is the same as the equality of $\mathrm{suc}\,m$ and suc, n which is why we didn't need to do anything in the boolean version. However, we have to massage the reason: if $m = n$ then $\mathrm{suc}\,m = \mathrm{suc}\,n$ we have seen this before we can prove it using resp suc. In the onther direction we have to show that $m \neq n$ implies $\mathrm{suc}\,m \neq \mathrm{suc}\,n$. That means from assuming $h : m = n \to 0$ we have to derive $\mathrm{suc} m = \mathrm{suc} n \to 0$, that is given $p : \mathrm{suc}\,m = \mathrm{suc}\,n$ we need to derive a hypothetical element of the empty type. Are only chance is to use h and to close the gap we need

$$\mathrm{injSuc} : \Pi_{m,n:\mathbb{N}}\mathrm{suc}\,m = \mathrm{suc}\,n \to m = n$$

injSuc thans for injectivity of successor and is another principle from Peano Arithmetic if the successors of two numbers are equal then the numbers are equal. Unlike resp this doesn't hold for all functions but it does hold for successor.

I am not sure I have convinced you that this is a nicer way to define this function. The nice thing is that this version of the function gives us both the computation and the correctness proofs. As Conor McBride expressed it: *It says on the tin what is inside the tin.* That is the type tells us everything we want to know about the function: it decides equality. Certainly this refined version of the function is technically more work but then I didn't actually spell out the details of the correctness proof of the boolean version of eq.