Did you notice that $\times$ actually appears twice: we can either view $A \times B$ as a non-dependent Σ-type ($\Sigma x : B.B$) or as a Bool-indexed Π-type. This dual nature of products indeed leads to slightly different approaches to products: either based on tupling or based on projections.

To summarize these observations we can say that we can classify the operators in 2 different ways: either along dependent / non-dependent or along binary or iterated:

non-dep.	dep.
$\times$	Σ
$\to$	Π

binary	iterated
$+$	Σ
$\times$	Π

This can lead to some linguistic confusions: what is a dependent product type? Hence I prefer just to say Π-type and Σ-type.

4.3 One Universe is not enough

So far we have avoided to consider types as explicit objects but instead they were part of our prose. Remember, I was saying: *given a type A we define the type of lists* List A. But what is List if not a function on types, that is List : **Type** $\to$ **Type**. Here we introduce **Type** as a type of types and this sort of thing is called a *universe* in Type Theory. So for example $\mathbb{N}$: **Type** but also $3 : \mathbb{N}$ hence $\mathbb{N}$ can appear on either side of the colon. A universe is a type whose elements are types again. This also works for Fin : $\mathbb{N} \to$ **Type** and Vec : $\mathbb{N} \to$ **Type** $\to$ **Type**. We call Fin and Vec families of types or dependent types and we should also call List a dependent type but the terminology is a bit vague here and some people would call it a *type indexed type*.

The reason is that functional programming languages support types like List but not *properly dependent types* like Fin or Vec. This has mainly to do with pragmatic reasons, namely that programming language implementers don't want to evaluate programs at compile time but they would have to do check that applying a function $f : \mathrm{Fin}\,7 \to \mathbb{N}$ to $i : \mathrm{Fin}\,(3+4)$ as in $f\,i$ is well typed. In this case this is easy and just involves checking that $7 \equiv 3 + 4$ but there are no limits what functions we could use here. In the case of a language like Haskell this could even involve a function that loops which would send the compiler into a loop. However, implementations of Type Theory like Agda or Coq aren't really so much better, at least from a pragmatic point of view, since we may use the Ackermann function here which may mean that while the function will eventually terminate this may only happen after the heat death of the universe.

The obvious question is what is the type of **Type**? Is it **Type** : **Type**. Hang on wasn't this the sort of thing where Frege got in trouble'? Does Russell's paradox applies here? The type of all type which does not contain itself? Not immediately because : isn't a proposition hence we cannot play this sort of trick here. Maybe this was what Per Martin-Löf was thinking when he wrote his first paper about Type Theory which did include **Type** : **Type**. This time it wasn't an English philosopher who pointed out the problem but a french one: Jean-Yves Girard. But then Per Martin-Löf is from Sweden not Germany.

Girard wasn't interested in dependent types at the time but rather in the type indexed by type variety. Actually he wasn't interested in types for programming but types for logic. His goal was to prove Takeuti's conjecture who was looking for a *syntactic proof of the consistency of 2nd order logic*, which is predicate logic extended by extra quantifiers which allow you to quantify over all propositions. Girard's approach was to introduce a λ-calculus which he called *System F* which corresponds to 2nd order logic in the same way as Gentzen's System T corresponds to Peano Arithmetic. I should add that the power of 2nd order logic and hence System F vastly exceeds Peano's Arithmetic / System T. By the power of a logic we mean the ability of one logic to *eat* another one: we can prove the consistency of Peano's arithmetic in 2nd order logic but not the other way around. Hence Takeuti's conjecture was asking for a relative simple, i.e. syntactic, proof of the consistency of a very powerful system. I believe it wasn't very clear what exactly he meant by this and it is maybe doubtful wether Girard's ingenious proof of the strong normalisation of System F fits this bill. Girard proved that all programs in System F terminate no matter how you evaluate them, this is called strong normalisation.

The main feature of System F is that we can construct Π-types that quantify over all types as in $\Pi X : \mathbf{Type}.X \to X$. This is the type of the *polymorphic identity function*:

$$I \equiv \lambda X.\lambda x : X.x : \Pi X : \mathbf{Type}.X \to X$$

The nice feature of the polymorphic identity is that it works for any type, e.g. $I\,\mathbb{N} : \mathbb{N} \to \mathbb{N}$ is the identity for natural numbers. It even works for its own type as in

$$I\,(\Pi X : \mathbf{Type}.X \to X) : (\Pi X : \mathbf{Type}.X \to X) \to (\Pi X : \mathbf{Type}.X \to X)$$

which means that in particular we can apply it to itself:

$$I\,(\Pi X : \mathbf{Type}.X \to X)\,I : \Pi X : \mathbf{Type}.X \to X$$

I find this rather mind boggling and it also looks dangerous since in the untyped λ-calculus self-apply was the main ingredient of the fixpoint combinator which enables us to run any program including some which don't terminate. Hence it is reassuring to know that strong normalisation holds and that this is indeed impossible.

There are some interesting games one can play with System F: there is a clever way to encode the natural numbers which is called *Church numerals*:

$$\mathbb{N} :\equiv \Pi X.X \to (X \to X) \to X$$

The idea is that a number is represented by a higher order function that repeats another function, e.g. $3\,a\,f \equiv f\,(f\,(f\,a))$. We can represent zero and successor:

$0 : \mathbb{N}$
$0 \equiv \lambda X, a, f.a$
$\mathrm{suc} : \mathbb{N} \to \mathbb{N}$
$\mathrm{suc} \equiv \lambda n, X, a, f.f\,(n\,X\,a\,f)$

and there also clever ways to encode the standard arithmetic functions:

$\mathrm{plus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathrm{plus} : \lambda m, n.\lambda X, a, f.mXf(nXfa)\mathrm{mult} \qquad : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathrm{mult} : \lambda m, n.\lambda X, a, f.mX(nXf)a$
$\mathrm{pow} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathrm{pow} : \lambda m, n.\lambda X, a, f.n(X \to X)(mX)$

Unlike our primitive recursive definitions, plus doesn't use suc, mult doesn't use plus and pow doesn't use mult. Instead we use the power to repeat: plus repeats the function f first n-times and then m-times, mult repeats the repetition of n, m-times and pow? I don't know anybody who understand pow without getting a headache but it is interesting that this is the first function which exploits the *polymorphic* nature of the natural numbers and instantiates the 2nd argument to a type different from X.

The idea of System F was so natural that it got reinvented in Computer Science by John Reynolds who introduced the polymorphic λ-calculus only to find out that Girard had invented basically the same calculus a few years earlier under the name System F. However, it is certainly Reynolds influence which lead to the polymorphic λ-calculus aka System F to be used as the base of type systems for many modern programming languages.

Ok, this was a long diversion. What has all this to do with Martin-Löf's Type Theory using **Type** : **Type**. I am almost there! One generalisation of System F is to allow not only to use **Type** but also functions over types, that is something like

$\Pi F : (\mathbf{Type} \to \mathbf{Type}).\Pi X.F\,(F\,X) \to X$

This extension which allows to quantify over functions over types is called System F^ω and it is still strongly normalizing. But Girard wondered wether this level could be polymorphic again, i.e. using System F instead just the simply typed λ-calculus to define functions over types, such as

$\Pi G : (\Pi X : \mathbf{Type}.X \to X).\Pi X : \mathbf{Type}.GX \to X$

But here Girard hit a brick wall - this system which he called System U is not strongly normalising anymore. Girard's truly ingenious paradox used an encoding of the Burali-Forti Paradox in System U: the collection of all well-orders cannot be well-ordered itself.

Now when Girard saw Martin-Löf's system he quickly realized that this system is powerful enough to encode System U, and hence it couldn't be strongly normalising itself. Even worse it turns out that in a system with **Type** : **Type** every type is inhabited including the empty type. This makes it useless as a logic because now we can prove everything.

To explain why **Type** : **Type** doesn't work I prefer a construction by Thierry Coquand [**?**] who showed that we can encode Russell's paradox in such a theory using trees. This is less general than Girard's paradox because it relies on the availability of trees and it doesn't work for System U (at least I am not aware how it could be adapted to System U). But Girard's paradox is very technical and hard to understand intuitively (at least for me).

Thierry's idea was to use an idea by Peter Aczel and encode the sets of set theory as trees. That is he defined a type Tree : **Type** with a constructor

$\mathrm{mkTree} : \Pi I : \mathbf{Type}(I \to \mathrm{Tree}) \to \mathrm{Tree}$

The idea is that mkTree works like the curly brackets from set theory. That is if we have any collection of trees indexed by any type we can form a new tree. For example we can define the empty tree empty : Tree using the empty type,

$\mathrm{empty} :\equiv \mathrm{mkTree}\, 0\, \mathrm{efq}$

where $\mathrm{efq} : 0 \to \mathrm{Tree}$ is a function that needs no definition. Also given $a, b : \mathrm{Tree}$ we can define a new tree $\mathrm{pair}\, a\, b : \mathrm{Tree}$ representing $\{a, b\}$. This implements the axiom of pairing.

$\mathrm{pair}\, a\, b :\equiv \mathrm{mkTree}\, \mathrm{Bool}\, (\lambda x.\mathrm{if}\, x\, \mathrm{then}\, a\, \mathrm{else}\, b)$

We don't get the axiom of extensionality because $\mathrm{pair}\, a\, b$ and $\mathrm{pair}\, b\, a$ are not equal. However, this doesn't matter for Russell's paradox which also works for trees. We do get unlimited comprehension for trees that is given a property of trees encoded as a family of types $P : \mathrm{Tree} \to \mathbf{Type}$ we can define $\mathrm{compr}\, P :$ Tree representing $\{t \mid P\, t\}$ as

$\mathrm{compr}\, P :\equiv \mathrm{mkTree}\, (\Sigma t : \mathrm{Tree}.P\, t)\pi_0$

That is the index type is a pair of a tree and a proof that it satisfies the property and the function which assigns trees to indices is just the first projection.

To define the property of trees that they are not an element of themselves we have to explain what an element of a tree is, namely a direct subtree. That is we define $\mathrm{el} : \mathrm{Tree} \to \mathrm{Tree} \to \mathbf{Type}$ with $\mathrm{el}\, a\, b$ expressing $a \in b$.

$\mathrm{el}\, a\, (\mathrm{mkTree}\, I\, f) \equiv \Sigma i : I.f\, i = a$

Here we use Σ to encode $\exists$ and we use the equality type which we haven't discussed yet but I hope this is clear anyway. Now we can express the property of a tree t not containing itsef $(\mathrm{el}\, t\, t) \to 0$ and we define Russell's tree $R : \mathrm{Tree}$ as

$R :\equiv (\mathrm{compr}(\lambda t.(\mathrm{el}\, t\, t) \to 0)$

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