Introduction to Homotopy Type Theory

Lecture notes for a course at the Types Summerschool 2017

Thorsten Altenkirch

July 14, 2017

1 What is this course about?

The word *type theory* has at least two meanings:

- The theory of types in programming language
- Martin-Löf's Type Theory as a constructive foundation of Mathematics

We will be mainly concerned with the latter (which is emphasised by capitalising it), even though there are interactions with the design of programming languages as well.

Type Theory is the base of a number of computer systems used as the base of interactive proof systems and very advanced (functional) programming language. Here is an incomplete list:

NuPRL is maybe the oldest implementation of Type Theory, which was developed at Cornell. It is based on a different flavour of type theory which was called *Extensional Type Theory* but which is now referred to as *Computational Type Theory*.

Coq is maybe now the system most used in formal Mathematics and has been used for some impressive developments, including a formal proof of the Four Colour Theorem and the verification of an optimising C compiler.

Agda is a sort of a twitter it can be used as a interactive proof assistant or as a dependently typed programming language.

Idris goes further on the programming language road by addressing more pragmatic concerns when using Type Theory for programming.

Lean is developed at Microsoft Research with strong support for automatic reasoning.

Cubical Cubical is a very new system and more a proof of concept but it is the only one (so far) that actually implements Homotopy Type Theory.

One way to introduce Type Theory is to pick one system (I usually pick Agda) and then learn Type Theory by doing it. While this is a good way to approach this subject, and do I recommend to play with a system I want to concentrate more on the conceptual issues and then I find that having to explain the intricacies of a particular system can be a bit of a distraction. Hence this course will be a paper based introduction to Type Theory.

This course can be viewed as a taster of the book on Homotopy Type Theory [2] which was the output of a special year at the Institute for Advanced Study in Princeton. However, a few things have happened since the book was written (e.g. the construction of cubical) and I will mention them where appropriate.

2 Type Theory vs Set Theory

I view Type Theory in the first place as a intuitive foundation of Mathematics. This is similar to how most Mathematicians use Set Theory: the have an intuitive idea what sets are but they don't usually refer back to the axioms of Set Theory. This is sometimes called *naive Set Theory* [1] and similar what I am doing here can be called *naive Type Theory*.

In Set Theory we write $3 \in \mathbb{N}$ to express that 3 is an element of the set of natural numbers. In Type Theory we write $3 : \mathbb{N}$ to express that 3 is an element of the type of natural numbers. While this looks superficially similar, there are important differences:

- While $3 \in \mathbb{N}$ is a proposition, $3 : \mathbb{N}$ is a *judgement*, that is a piece of static information.
- In Type Theory every object and every expression has a (unique) type which is statically determined. [2]
- Hence it doesn't make any sense to use $a : A$ as a proposition.
- This is similar to the distinction between statically and dynamically typed programming languages. While in dynamically typed languages there are runtime functions to check the type of an object this doesn't make sense in statically typed languages.
- In Set Theory we define $P \subseteq Q$ as $\forall x.x \in P \to x \in Q$. We can't do this in Type Theory because $x \in P$ is not a proposition.
- Also set theoretic operations like $\cup$ or $\cap$ are not operations on types. However, they can be defined as operations on predicates, aka subsets, of a given type. $\subseteq$ can be defined as a predicate on such subsets.
- Type Theory is extensional in the sense that we can't talk about details of encodings.

[1] This is also the title of a well kown book by Halmos [1].

[2] We are not considering subtyping here, which can be understood as a notational device allowing the omission of implicit coercions.

- This is different in Set Theory where we can ask wether $\mathbb{N} \cap \text{Bool} = \emptyset$? Or wether $2 \in 3$? The answer to these questions depends on the choice of representation of these objects and sets.

Apart from the judgement $a : A$ there is also the judgement $a \equiv_A b$ which means that $a, b : A$ are *definitionally* equal. We write definitions using $:\equiv$, e.g. we can define $n : \mathbb{N}$ as 3 by writing $n :\equiv 3$. As for $a : A$ definitional equality is a static property which can be determined statically and hence which doesn't make sense as a proposition. We will later introduce $a =_A b$, *propositional equality* which can be used in propositions.

While Type Theory is in some sense more restrictive than Set Theory, this does pay off. Because we cannot talk about intensional aspects, i.e. implementation details, we can identify objects which have the same extensional behaviour. This is reflected in the *univalence axiom*, which identifies extensionally equivalent types (such as unary and binary natural numbers).

Another important difference between Set Theory and Type Theory is the way propositions are treated: Set Theory is formulated using predicate logic which relies on the notion of *truth*. Type Theory is self-contained and doesn't refer to truth but to evidence. Using the propositions-as-types translation (also called the Curry-Howard equivalence) we can assign to any proposition P the type of its evidence $[\![P]\!]$ using the following table:

$$\begin{aligned}
[\![P \implies Q]\!] &\equiv [\![P]\!] \to [\![Q]\!] \\
[\![P \wedge Q]\!] &\equiv [\![P]\!] \times [\![Q]\!] \\
[\![\text{True}]\!] &\equiv 1 \\
[\![P \vee Q]\!] &\equiv [\![P]\!] + [\![Q]\!] \\
[\![\text{False}]\!] &\equiv 0 \\
[\![\forall x : A.P]\!] &\equiv \Pi x : A.[\![P]\!] \\
[\![\exists x : A.P]\!] &\equiv \Sigma x : A.[\![P]\!]
\end{aligned}$$

0 is the empty type, 1 is the type with exactly one element and $+$ is the sum or disjoint union of types. $\to$ (function type) and $\times$ should be familiar but we will revisit all of them from a type theoretic perspective. Π and Σ are less familiar in Set Theory and we will have a look at them later.

We are using a typed predicate logic, in the lines for universal and existential quantification A refers to a type. Other connectives are defined: $\neg P$ is defined as $P \implies \text{False}$. Logical equivalence $P \Leftrightarrow Q$ is defined as $(P \implies Q) \wedge (Q \implies P)$. Careful, equivalence such as $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$ or $\neg(\forall x : A.P) \Leftrightarrow \exists x : A.\neg P$ do not hold in Type Theory, hence we cannot define $\vee$ or $\exists$ via $\neg$ and $\wedge$ and $\forall$.

Later we will see a refinement of the proposition as types translation, which changes the translation of $P \vee Q$ and $\exists x : A.P$.

3 Non-dependent types

3.1 Universes

To get started we have to say what a type is. We could achieve this by introducing another judgement but instead I am going to use universes. A universe is a type of types. For example to say that $\mathbb{N}$ is a type, I write $\mathbb{N} : \mathbf{Type}$ where **Type** is a universe.

But what is the type of **Type**? Do we have $\mathbf{Type} : \mathbf{Type}$? It is well known that this doesn't work in Set Theory due to Russell's paradox (the set of all sets which does not contain itself). However, in Type Theory $a : A$ is not a proposition, hence it is not immediately clear wether the paradox still works.

However, it is possible to encode Russell's paradox in a Type Theory with $\mathbf{Type} : \mathbf{Type}$ by using trees which can branch over any type. In this theory we can construct a tree of all trees which don't have themself as immediate subtree. This tree is a subtree of itself iff it isn't which enables us to derive a contradiction.

To avoid Russell's paradox we introduce a hierarchy of universes

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

and we decree that any type $A : \mathbf{Type}_i$ can be lifted to a type $A^+ : \mathbf{Type}_{i+1}$. Being explicit about universe levels can be quite annoying hence we are going to ignore them most of the time but try to make sure that we don't use universes in a cyclic way. That is we write **Type** as a metavariable for $\mathbf{Type}_i$ and assume that all the levels are the same unless stated explicitly.

3.2 Functions

While in Set Theory functions are a derived concept (a subset of the cartesian product with certain properties), in Type Theory functions are a primitive concept. The basic idea is the same as in functional programming: basically a function is a black box and you can feed it elements of its domain and out come elements of its codomain. Hence given $A, B : \mathbf{Type}$ we introduce the type of functions $A \to B : \mathbf{Type}$. The central operation is application, that is given a function $f : A \to B$ and an argument $a : A$ we can construct $f\,a : B$. That makes sense we are just feeding some input to our box to get some output on the other end. In the propositions as types view this corresponds to *modens ponens*. When we define a function we need to say what the function computes for an arbitrary input, that is we give a defining equation $f(x) :\equiv \dots$.

As an example we define $f : \mathbb{N} \to \mathbb{N}$ as $f(x) :\equiv x + 3$. Having defined f we can apply it, e.g. $f(2) : \mathbb{N}$ and we can evaluate this application by replacing all occurrences of the parameter x in the body of the function $x + 3$ by the actual argument 2 hence $f(2) \equiv 2 + 3$ and if we are lucky to know how to calculate $2 + 3$ we can conclude $f(2) \equiv 5$.

One word about syntax: in functional programming and in Type Theory we try to save brackets and write $f\,2$ for the application and also in the definition

we write $f\,x :\equiv x + 3$.

The explicit definition of a function requires a name but we should be able to define a function without having to give it a name - this is the justification for the λ-notation. We write $\lambda x.x + 3 : \mathbb{N} \to \mathbb{N}$ avoiding to have to name the function. We can apply this $(\lambda x.x+3)(2)$ and the equality $(\lambda x.x+3)(2) \equiv 2+3$ is called β-reduction. The explicit definition $f\,x \equiv x+3$ can now be understood as a shorthand for $f \equiv \lambda x.x + 3$.

In Type Theory every function has exactly one argument. To represent functions with several arguments we use *currying*, that is we use a function that returns a function. So for example the addition function $g :\equiv \lambda x.\lambda y.x + y$ has type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$, that is if we apply it to one argument $g\,3 : \mathbb{N} \to \mathbb{N}$ it returns the function that add 3 namely $\lambda y.3 + y$. We can continue and supply a further argument $(g\,3)\,2 : \mathbb{N}$ which reduces

$$\begin{aligned}(g\,3)\,2 &\equiv (\lambda y.3 + y)\,2\\ &\equiv 3 + 2\end{aligned}$$

To avoid the proliferation of brackets we decree that application is left associative hence we can write $g\,3\,2$ and that $\to$ is right associative hence we can write $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ for the type of g.

When calculating with variables we have to be a bit careful. Assume we have a variable $y : \mathbb{N}$ hanging around, now what is $g\,y$? If we naively replace x by y we obtain $\lambda y.y + y$, that is the variable y got *captured*. This is not the intended behaviour and to avoid capture we have to rename the bound variable that is $\lambda x.\lambda y.x + y \equiv \lambda x.\lambda z.x + z$ - this equality is called α-congruence. After having done this we can β-reduce. Here is the whole story

$$\begin{aligned}g\,y &\equiv (\lambda x.\lambda y.x + y)\,y\\ &\equiv (\lambda x.\lambda z.x + z)\,y\\ &\equiv \lambda z.y + z\end{aligned}$$

Clearly the choice of z here is arbitrary, but any other choice (apart from y) would have yielded the same result upto α-congruence.

3.3 Products and sums

Given A, B : **Type** we can form their product $A \times B$: **Type** and their sum $A + B$: **Type**. The elements of a product are tuples, that is $(a, b) : A \times B$ if $a : A$ and $b : B$. The elements of a sum are injections that is left $a : A + B$ if $a : A$ and right $b : A + B$, if $b : B$.

To define a function from a product or a sum it is sufficient to say what the functions returns for the constructors, that is for tuples in the case of a product or the injections in the case of a sum.

As an example we derive the tautology

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

using the propositions as types translation. We assume that $P, Q, R : \mathbf{Type}$, we have to construct an element of the following type

$$\begin{aligned}&((P \times (Q + R) \to (P \times Q) + (P \times R))\\ \times&((P \times Q) + (P \times R) \to P \times (Q + R))\end{aligned}$$

We define:

$$\begin{aligned}&f : P \times (Q + R) \to (P \times Q) + (P \times R)\\ &f\,(p, \text{left}\,q) :\equiv \text{left}\,(p, q)\\ &f\,(p, \text{right}\,r) :\equiv \text{right}\,(p, r)\end{aligned}$$

$$\begin{aligned}&g : (P \times Q) + (P \times R) \to P \times (Q + R)\\ &g\,(\text{left}\,(p, q)) :\equiv (p, \text{left}\,q)\\ &g\,(\text{right}\,(p, r))) :\equiv (p, \text{right}\,r)\end{aligned}$$

Now the tuple (f, g) is an element of the type above.

In this case the two functions are actually inverses, but this is not necessary to prove the logical equivalence.

Exercise 1 *Using the propositions as types translation, try* [3] *to prove the following tautologies:*

1. $(P \wedge Q \implies R) \Leftrightarrow (P \implies Q \implies R)$
2. $((P \vee Q) \implies R) \Leftrightarrow (P \implies R) \wedge (Q \implies R)$
3. $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$
4. $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$
5. $\neg(P \Leftrightarrow \neg P)$

where $P, Q, R : \mathbf{Type}$ *are propositions represented as types.*

Exercise 2 *While the principle of excluded middle* $P \vee \neg P$ *(*tertium non datur*) is not provable, prove its double negation using the propositions as types translation:*

$$\neg\neg(P \vee \neg P)$$

If for a particular proposition P *we can establish* $P \vee \neg P$ *then we can also derive the principle of indirect proof (*reduction ad absurdo*) for the same proposition* $\neg\neg P \implies P$*. Hence show:*

$$(P \vee \neg P) \implies (\neg\neg P \implies P)$$

However, the converse does not hold (what would be a counterexample?). However, use the two tautologies to show that the two principles are equivalent.

[3] I didn't say they are all tautologies!

Functions out of products and sums can be reduced to using a fixed set of combinators called non-dependent eliminators or *recursors* (even though there is no recursion going on). For the following we assume as given $A, B, C : \mathbf{Type}$.

$$\mathrm{R}_C^{A\times B} : (A \to B \to C) \to A \times B \to C$$
$$\mathrm{R}_C^{A\times B}\, f\, (a, b) :\equiv f\, a\, b$$

$$\mathrm{R}_C^{A+B} : (A \to C) \to (B \to C) \to A + B \to C$$
$$\mathrm{R}_C^{A+B}\, f\, g\, (\mathrm{left}\, a) :\equiv f\, a$$
$$\mathrm{R}_C^{A+B}\, f\, g\, (\mathrm{right}\, b) :\equiv g\, b$$

The recursor $\mathrm{R}^\times$ for products maps a curried function $f : A \to B \to C$ into its uncurried form, taking tuples as arguments. The recursor R^+ basically implements the *case* function performing case analysis over elements of $A + B$.

Exercise 3 *Show that using the recursor* $\mathrm{R}^\times$ *we can define the projections:*

$$\mathrm{fst} : A \times B \to A$$
$$\mathrm{fst}\, (a, b) :\equiv a$$
$$\mathrm{snd} : A \times B \to B$$
$$\mathrm{snd}\, (a, b) :\equiv b$$

Vice versa: can the recursor be defined using only the projections?

We also have the case of an empty product 1, called the unit type and the empty sum 0, the empty type. There is only one element of the unit type: $() : 1$ and none in the empty type. We introduce the corresponding recursors:

$$\mathrm{R}^1 : C \to (1 \to C)$$
$$\mathrm{R}^1\, c\, () :\equiv c$$

$$\mathrm{R}^0 : 0 \to C$$

The recursor for 1 is pretty useless, it just defines a constant function. The recursor for the empty type implements the logical principle *eq falso quod libet*, from false follows everything. There is no defining equation because it will never be applied to an actual element.

Exercise 4 *Construct solutions to exercises 1 and 2 using only the eliminators.*

The use of arithmetical symbols for operators on types is justified because they act like the corresponding operations on finite types. Let us identify the

number n with the type of elements $0_n, 1_n, \ldots (n-1)_n : \overline{n}$, then we observe that it is indeed the case that:

$$\begin{aligned} \overline{0} &= 0 \\ \overline{m+n} &= \overline{m} + \overline{n} \\ \overline{1} &= 1 \\ \overline{m \times n} &= \overline{m} \times \overline{n} \end{aligned}$$

Read $=$ here as *has the same number of elements*. This use of equality will be justified later when we introduce the univalence principle.

The arithmetic interpretation of types also extends to the function type, which corresponds to exponentiation. Indeed, in Mathematics the function type $A \to B$ is often written as B^A. And indeed we have:

$$\overline{m^n} = \overline{n} \to \overline{m}$$

4 Dependent types

By a dependent type we mean a type indexed by elements of another type. For example the types of n-tuples A^n : **Type** their elements are $(a_0, a_1, \ldots a_{n-1}) : A^n$ where $a_i : A$, or the finite type $\overline{n}$: **Type**. Indeed, tuples are also indexed by A : **Type**. We can use functions into **Type** to represent these dependent types:

$$\begin{aligned} &\mathrm{Vec} : \mathbf{Type} \to \mathbb{N} \to \mathbf{Type} \\ &\mathrm{Vec}\, A\, n :\equiv A^n \end{aligned}$$

$$\begin{aligned} &\mathrm{Fin} : \mathbb{N} \to \mathbf{Type} \\ &\mathrm{Fin}\, n :\equiv \overline{n} \end{aligned}$$

In the propositions as types view dependent types are used to represent predicates, e.g. $\mathrm{Prime} : \mathbb{N} \to \mathbf{Type}$ assigns to any natural number $n : \mathbb{N}$ the type of evidence $\mathrm{Prime}\, n$: **Type** that n is a prime number. This does not need to be inhabited, e.g $\mathrm{Prime}\, 4$ is empty. Using currying we can use this also to represent relations, e.g. $\leq : \mathbb{N} \to \mathbb{N} \to \mathbf{Type}$, $m \leq n$: **Type** is the type of evidence that m is less or equal to n.

4.1 Π-types and Σ-types

Π-types generalize function types to allow the codomain of a function to depend on the domain. For example consider the function zeroes that assigns to any natural number $n : \mathbb{N}$ a vector of n zeroes

$$\underbrace{(0, 0, \ldots, 0)}_{n} : \mathbb{N}^n$$

We use Π to write such a type:

$$\begin{aligned}&\text{zeroes} : \Pi n : \mathbb{N}.\mathbb{N}^n\\&\text{zeroes}\, n :\equiv \underbrace{(0, 0, \ldots, 0)}_{n}\end{aligned}$$

The non-dependent function type can now be understood as a special case of Π-types, $A \to B \equiv \Pi - : A.B$.

We are using the same syntax to write dependent functions as for non-dependent functions. That is we can use λ-abstraction to introduce anonymous dependent functions that is we can write

$$\lambda n. \underbrace{(0, 0, \ldots, 0)}_{n} : \Pi n : \mathbb{N}.\mathbb{N}^n$$

Given $A : \mathbf{Type}$ and $B : A \to \mathbf{Type}$ and a dependent function $f : \Pi x : A.B\, x$ we can apply it to $a : A$ and obtain an element of the codomain at the specific instance, that is $f\, a : B\, a$.

Currying also works for dependent functions, but it is worthwhile to track the dependencies. In the case of a function with two arguments, the 2nd argument can depend on the first and the codomain can depend on bith of them. That is we have $A : \mathbf{Type}, B : A \to \mathbf{Type}$ for the inputs and $C : \Pi x : A.B\, x \to \mathbf{Type}$. Now a dependent function with two inputs woud have the type $\Pi x : A, \Pi y : B\, x.C\, x\, y$. Unlike in the case of non-dependent functions we cannot switch the order of arguments of there is a proper dependency.

In the same vain, Σ-types generalize product types to the case when the 2nd component depends on the first. For example we can represent tuples of arbitrary size as a pair of a natural number $n : \mathbb{N}$ and a vector of this size A^n as an element of $\Sigma n : \mathbb{N}.A^n$. So for example $(3, (1, 2, 3)) : \Sigma n : \mathbb{N}.\mathbb{N}^n$ because $(1, 2, 3) : \mathbb{N}^3$. Indeed, this type is very useful so we give it a name:

$$\begin{aligned}&\text{List} : \mathbf{Type} \to \mathbf{Type}\\&\text{List}\, A :\equiv \Sigma n : \mathbb{N}.A^n\end{aligned}$$

In general again assuming $A : \mathbf{Type}$ and $B : A \to \mathbf{Type}$ we can make dependent tuples, given $a : A$ and $b : B\, a$ we can form $(a, b) : \Sigma x : A.B, x$.

In the propositions as types translation we use Π-types to represent evidence for universal quantification. For example a proof of $\forall x : \mathbb{N}.1 + x = x + 1$ is a dependent function of type $f : \Pi x : \mathbb{N}.1 + x = x + 1$, such that applying it as in $f\, 3$ is evidence that $1 + 3 = 3 + 1$.

Similar, we use Σ-types to represent evidence for existential quantification where the first component is the instance for which the property is supposed to hold and the second component a proof that it holds for this particular instance. For example the statement $\exists n : \mathbb{N}.\text{Prime}\, n$ is translated to $\Sigma n : \mathbb{N}.\text{Prime}\, n$ and a proof of this is $(3, p)$ where $p : \text{Prime}\, 3$.

As for Π-types the non-dependent products arise as a special case of Σ-types: $A \times B :\equiv \Sigma - : A.B$.

To avoid clutter we sometimes want to omit arguments to a Π-type when it is derivable from later arguments or the first component of a Σ-type. In this case we write the argument in subscript as in $\Pi_{x:A} B\,x$ or $\Sigma_{x:A} B\,x$. For example if we define $\mathrm{List}\,A :\equiv \Sigma_{n:\mathbb{N}}.A^n$ we can omit the length and just write $(1,2,3) : \mathrm{List}\,\mathbb{N}$.

Exercise 5 *Using the propositions as types translation for predicate logic try to derive the following tautologies:*

1. $(\forall x : A.P\,x \wedge Q\,x) \Leftrightarrow (\forall x : A.P\,x) \wedge (\forall x : A.Q\,x)$
2. $(\exists x : A.P\,x \vee Q\,x) \Leftrightarrow (\exists x : A.P\,x) \vee (\exists x : A.Q\,x)$
3. $(\exists x : A.P\,x) \implies R \Leftrightarrow \forall x : A.P\,x \implies R$
4. $\neg\exists x : A.P\,x \Leftrightarrow \forall x : A.\neg P\,x$
5. $\neg\forall x : A.P\,x \Leftrightarrow \exists x : A.\neg P\,x$

where $A, B : \mathbf{Type}$ *and* $P, Q : A \to \mathbf{Type}$, $R : \mathbf{Type}$, *represent predicates and a proposition.*

We have seen that Σ-types are related to products but they are also related to sums. Indeed we can derive + from Σ using as the first component an element of $\mathrm{Bool} = \overline{2}$ and the second component is either the first or the 2nd component of the sum (assuming $A, B : \mathbf{Type}$:

$$\begin{aligned} &A + B : \mathbf{Type} \\ &A + B :\equiv \Sigma x : \mathrm{Bool}.\mathrm{if}\,x\,\mathrm{then}\,A\,\mathrm{else}\,B \end{aligned}$$

In the same way we can also derive × from Π by using dependent functions over the booleans which returns either one or the other component of the product.

$$\begin{aligned} &A \times B : \mathbf{Type} \\ &A \times B :\equiv \Pi x : \mathrm{Bool}.\mathrm{if}\,x\,\mathrm{then}\,A\,\mathrm{else}\,B \end{aligned}$$

It is interesting to note that × can be viewed in two different ways: either as a non-dependent Σ-type or as a dependent function-type over the booleans.

Exercise 6 *Show that injections, pairing, non-dependent eliminators can be derived for these encodings of sums and products.*

Finally, we notice that the arithmetic interpretation of types extends to Σ and Π giving a good justification for the choice of their names, let $m : \mathbb{N}$ and $f : \overline{m} \to \mathbb{N}$:

$$\begin{aligned} \overline{\Sigma x : m.f\,x} &= \Sigma_{x=0}^{x<\overline{m}} \overline{f\,x} \\ \overline{\Pi x : m.f\,x} &= \Pi_{x=0}^{x<\overline{m}} \overline{f\,x} \end{aligned}$$

Once we have introduced dependent types we need to revisit the eliminators or recursors. The recursors provided a general way to define non-dependent functions out of a type but now we want to consider dependent functions. E.g. in the case of the sum $A + B$ we assume that we have a dependent type $C : A + B \to \mathbf{Type}$ and we want to construct a function $h : \Pi x : A + B.C\,x$. In the non-dependent case we needed functions $f : A \to C$ and $g : B \to C$ to cover the cases. Now we need dependent functions: $f : \Pi x : A.C\,(\mathrm{left}\,a)$ and $\Pi y : B.C\,(\mathrm{right}\,x)$. Now we can define

$$h\,(\mathrm{left}\,a) :\equiv f\,a$$
$$h\,(\mathrm{right}\,b) :\equiv g\,b$$

We can distill this process into one dependent eliminator: [4]

$$E_C^{A+B} : (\Pi x : A.C\,(\mathrm{left}\,a)) \to (\Pi y : B.C\,(\mathrm{right}\,x) \to \Pi z : A + B.C\,z$$
$$E_C^{A+B}\,f\,g\,(\mathrm{left}\,a) :\equiv f\,a$$
$$E_C^{A+B}\,f\,g\,(\mathrm{right}\,b) :\equiv g\,b$$

Exercise 7 *Derive dependent eliminators for products ($E^{A\times B}$), the unit type (E^1) and the empty type E^0. Is the eliminator for the unit type still redundant (as the recursor)? Can you derive the eliminator for* 0 *from the non-dependent recursor R^0?*

We can also define a dependent recursor or eliminator for Σ-types which allows us to define any dependent function out of a Σ-type. This eliminator is not just parametrized by a type but by a family $C : \Sigma x : A.B\,x \to \mathbf{Type}$:

$$\mathrm{E}^{\Sigma} : (\Pi x : A, \Pi y : B\,x.C\,(x, y)) \to \Pi p : \Sigma x : A.B\,x.C\,p$$
$$\mathrm{E}^{\Sigma}\,f\,(a, b) :\equiv f\,a\,b$$

Exercise 8 *As in exercise 3 we can define projections out of a Σ-type, let A :* **Type** *and $B : A \to$* **Type***:*

$$\mathrm{fst} : \Sigma x : A.B\,x \to A$$
$$\mathrm{fst}\,(a, b) :\equiv a$$
$$\mathrm{snd} : \Pi p : \Sigma x : A.B\,x.B\,(\mathrm{fst}\,p)$$
$$\mathrm{snd}\,(a, b) :\equiv b$$

Note that the type of the 2nd projections is a dependent function type using the first projection.

Derive the projections using only the eliminator E^{Σ}. Vice versa, can you derive the eliminator from the projections without making further assumptions?

[4] Using dependent types we could have turned A, B, C into actual parameters but makes the type almost unreadable.

4.2 The equality type

Given $a, b : A$ the equality type $a =_A b : \mathbf{Type}$ is generated from one constructor $\mathrm{refl} : \Pi_{x:A} x =_A x$. That is we are saying that two things which are identical are equal and this is the only way to construct an equality. Using this idea we can establish some basic properties of equality, namely that it is an equivalence relation, that is a relation that is reflexive, symmetric and transitive. Moreover, it is also a congruence, it is preserved by all functions. Since we already have reflexivity from the definition, lets look at symmetry first. We can define symmetry by just saying how it acts on reflexivity:

$$\begin{aligned}&\mathrm{sym} : \Pi_{x,y:A} x = y \to y = x\\&\mathrm{sym}_{a\,a}\mathrm{refl}_a :\equiv \mathrm{refl}_a\end{aligned}$$

The main idea here is that once we apply sym to refl we also know that that the two elements $x, y : A$ must be identical and hence we can prove the result using refl again. Note that pattern matching with dependent types behaves very different from pattern matching for simple types: In the simple typed case we could look at each argument in isolation there was no interaction. With dependent types it is very different, matching one input may constrain what the others can be. Somebody could ask what happens if I partially apply sym to two different $a, b : A$ what is $\mathrm{sym}_{a\,b}$, since we have never defined the function for this case? However, we know that there is no element of $a =_A b$ in this case and hence there is no need to define the function in this case for the same reason as there was no need to define $R^0 : C \to C$.

Exercise 9 *Provide proofs of*

$$\begin{aligned}&\mathrm{trans} : \Pi_{x,y,z:A} x = y \to y = z \to x = z\\&\mathrm{resp} : \Pi f : A \to B.\Pi_{m,n:\mathbb{N}} m = n \to f\,m = f\,n\end{aligned}$$

using the same idea.

For equality there is a recursor and an eliminator, and for the examples above we only need the recursor because we have no dependency on the actual proofs of equality. However, there is some dependency because equality itself is a dependent type. We assume a family that depends on the indices of equality but not on the equality proofs themselves: $C : A \to A \to \mathbf{Type}$ then the recursor is:

$$\begin{aligned}&\mathrm{R}^{=} : (\Pi x : A.C\,x\,x) \to \Pi_{x,y:A} x = y \to C\,x\,y\\&\mathrm{R}^{=}\,f\,(\mathrm{refl}_a) :\equiv f\,a\end{aligned}$$

Exercise 10 *Derive* $\mathrm{sym}, \mathrm{trans}, \mathrm{resp}$ *using the recursor* $\mathrm{R}^{=}$.

What would be a statement that actually depends on equality proofs? It seems that equality is rather trivial since there s at most one proof of it and we should be able to prove this. This is called uniqueness of equality proofs

and states that any two proofs of equality are equal and it has an easy direct definition exploiting exactly the fact that the only proof of equality is reflexivity:

$$\begin{aligned}&\mathrm{uep} : \Pi_{x,y:A}\Pi p, q : x = y\\&\mathrm{uep}\,\mathrm{refl}_a\,\mathrm{refl}_a :\equiv \mathrm{refl}_{\mathrm{refl}_a}\end{aligned}$$

Now we should be able to perform the usual exercise and reduce the direct definition of uep to one using only the eliminator. The first step is clear, by eliminating one argument we can reduce the problem to:

$$\Pi_{x:A}\Pi q : x = x.\mathrm{refl}_x = q$$

but now we are stuck. We cannot apply the eliminator because we need a family were both indices are arbitrary. Indeed, Hofmann and Streicher were able to show that uep is unprovable from the eliminator. In the next section we will discuss reasons why this is actually not a bad thing. However, at least based on our current understanding of equality it seems that this is an unwanted incompleteness. One which can actually fixed by introducing a special eliminator which works exactly in the case when we want to prove something about equality proofs where both indices are equal. That is we assume as given a family $C : \Pi_{x:A} x = x \to \mathbf{Type}$ and introduce

$$\begin{aligned}&K : (\Pi_{x:A} C\,\mathrm{refl}_x) \to \Pi_{x:A}\Pi p : x = x.C\,p\\&K\,f\,\mathrm{refl}_x :\equiv f_x\end{aligned}$$

This eliminator is called K because K is the next letter after J.

Exercise 11 *Derive uep using only* $\mathrm{E}^{=}$ *and* K*.*

Exercise 12 *Instead of viewing equality as a relation generated by* refl*, we can also fix one index* $a : A$ *and now define the predicate of being equal to* a*:* $a = - : A \to \mathbf{Type}$*. This predicate is generated by* $\mathrm{refl}_a : a = a$ *so no change here. However, the eliminator looks different. Let's fix* $C : \Pi x : A.a = x \to \mathbf{Type}$

$$\begin{aligned}&J'_a : (C\,\mathrm{refl}_a) \to \Pi_{x:A}\Pi p : a = x.C\,p\\&J'_a\,f\,(\mathrm{refl}_a) :\equiv f\end{aligned}$$

Show that $\mathrm{E}^{=}$ *and* J' *are interderivable, that is both views of equality are equivalent.*

4.3 Induction and recursion

Following Peano the natural numbers are introduced by saying that 0 is a natural number $(0 : \mathbb{N})$, and if n is a natural number $(n : \mathbb{N})$ then $\mathrm{suc}\,n$ is a natural number $(\mathrm{suc}\,n)$, which is equivalent to saying $\mathrm{suc} : \mathbb{N} \to \mathbb{N}$. When defining a function out of the natural numbers, we allow ourselves to recursively use

the function value on n to compute it for $\mathrm{suc}\, n$. An example is the doubling function:

$$\begin{aligned}&\mathrm{double} : \mathbb{N} \to \mathbb{N}\\&\mathrm{double}\, 0 :\equiv 0\\&\mathrm{double}\, (\mathrm{suc}\, n) :\equiv \mathrm{suc}\, (\mathrm{suc}\, (\mathrm{double}\, n))\end{aligned}$$

We can distill this idea into a non-dependent eliminator which is now rightfully called the recursor:

$$\begin{aligned}&\mathrm{R}^{\mathbb{N}} : C \to (C \to C) \to \mathbb{N} \to C\\&\mathrm{R}^{\mathbb{N}}\, z\, s\, 0 :\equiv z\\&\mathrm{R}^{\mathbb{N}}\, z\, s\, (\mathrm{suc}\, n) :\equiv s\, (\mathrm{R}\, z\, s\, n)\end{aligned}$$

Exercise 13 *We define addition recursively:*

$$\begin{aligned}&+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\\&0 + n :\equiv n\\&(\mathrm{suc}\, m) + n :\equiv \mathrm{suc}\, (m + n)\end{aligned}$$

Define addition using only the recursor $\mathrm{R}^{\mathbb{N}}$*.*

Exercise 14 *Not all recursive functions exactly fit into this scheme. For example consider the function that halves a number forgetting the remainder:*

$$\begin{aligned}&\mathrm{half} : \mathbb{N} \to \mathbb{N}\\&\mathrm{half}\, 0 :\equiv 0\\&\mathrm{half}\, (\mathrm{suc}\, 0) :\equiv 0\\&\mathrm{half}\, (\mathrm{suc}\, (\mathrm{suc}\, n)) :\equiv \mathrm{suc}\, (\mathrm{half}\, n)\end{aligned}$$

Try to derive half *only using the recursor* $\mathrm{R}^{\mathbb{N}}$*.*

When we want to prove a statement about natural numbers we have to construct a dependent function. An example is the proof that half is the left inverse of double: $\forall n : \mathbb{N}.\mathrm{half}\, (\mathrm{double}\, n) = n$. I haven't introduced equality yet but we only need two ingredients to carry out this construction, given A, B : **Type**:

$$\begin{aligned}&\mathrm{refl} : \Pi x : A.x = x\\&\mathrm{resp} : \Pi f : A \to B.\Pi_{m,n:\mathbb{N}} m = n \to f\, m = f\, n\end{aligned}$$

Using those we can define a dependent function verifying the statement:

$$\begin{aligned}&h : \Pi n : \mathbb{N}.\mathrm{half}\, (\mathrm{double}\, n) = n\\&h\, 0 :\equiv \mathrm{refl}\, 0\\&h\, (\mathrm{suc}\, n) :\equiv \mathrm{resp}\, \mathrm{suc}\, (h\, n)\end{aligned}$$

As for Σ-types we can derive dependent functions out of the natural numbers using a dependent recursor or eliminator. Assume that we have a dependent type $C : \mathbb{N} \to \mathbf{Type}$:

$$\begin{aligned}
&\mathrm{E}^{\mathbb{N}} : C\,0 \to (\Pi n : \mathbb{N}.C\,n \to C\,(\mathrm{suc}\,n)) \to \Pi n : \mathbb{N}.C\,n\\
&\mathrm{E}^{\mathbb{N}}\,z\,s\,0 :\equiv z\\
&\mathrm{E}^{\mathbb{N}}\,z\,s\,(\mathrm{suc}\,n) :\equiv s\,(\mathrm{E}\,z\,s\,n)
\end{aligned}$$

Exercise 15 *Derive h using only the dependent eliminator $\mathrm{E}^{\mathbb{N}}$.*

The type of $\mathrm{E}^{\mathbb{N}}$ precisely corresponds to the principle of induction - indeed from the propositions as types point of view induction is just dependent recursion.

Exercise 16 *Show that the natural numbers with $+$ and 0 form a commutative monoid:*

1. $\forall x : \mathbb{N}.0 + x = x$
2. $\forall x : \mathbb{N}.x + 0 = x$
3. $\forall x, y, z : \mathbb{N}.x + (y + z) = (x + y) + z$
4. $\forall x, y : \mathbb{N}.x + y = y + x$

Not all dependent functions out of the natural numbers arise from the propositions as types translation. An example is the function $\mathrm{zeroes} : \Pi n : \mathbb{N}.\mathbb{N}^n$ which we only introduced informally. We can make this precise by inductively defining tuples:

$$\begin{aligned}
&\mathrm{nil} : A^0\\
&\mathrm{cons} : \Pi_{n:\mathbb{N}} A^n \to A^{\mathrm{suc}\,n}
\end{aligned}$$

Using this we can define zeroes by recursion

$$\begin{aligned}
&\mathrm{zeroes}\,0 :\equiv \mathrm{nil}\\
&\mathrm{zeroes}\,(\mathrm{suc}\,n) :\equiv \mathrm{cons}\,0\,(\mathrm{zeroes}\,n)
\end{aligned}$$

This can be easily translated into an application of the eliminator

$$\mathrm{zeroes} :\equiv \mathrm{E}^{\mathbb{N}}\,\mathrm{nil}\,(\mathrm{cons}\,0)$$

Exercise 17 *We can also define the finite types in an inductive way, overloading 0 and* suc*:*

$$\begin{aligned}
&0 : \Pi_{n:\mathbb{N}} \overline{\mathrm{suc}\,n}\\
&\mathrm{suc} : \Pi_{n:\mathbb{N}} \overline{n} \to \overline{\mathrm{suc}\,n}
\end{aligned}$$

Using this and the inductive definition of A^n derive a general projection operator

$$\mathrm{nth} : \Pi_{n:\mathbb{N}} A^n \to \overline{n} \to A$$

that extracts an arbitrary component of a tuple.

Exercise 18 *Suggest definitions of eliminators for tuples and finite types. Can you derive all the examples using them?*

4.4 The equality type

What would be a statement that actually depends on equality proofs? It seems that equality is rather trivial since there s at most one proof of it and we should be able to prove this. This is called uniqueness of equality proofs and states that any two proofs of equality are equal and it has an easy direct definition exploiting exactly the fact that the only proof of equality is reflexivity:

$$\begin{aligned}&\mathrm{uep} : \Pi_{x,y:A}\Pi p, q : x = y\\ &\mathrm{uep}\,\mathrm{refl}_a\,\mathrm{refl}_a :\equiv \mathrm{refl}_{\mathrm{refl}_a}\end{aligned}$$

We now define the dependent eliminator for equality which $\mathrm{E}^{=}$ which is also called J but we stick to our terminology. This time we use a family that does indeed depend on the equality proof $C : \Pi_{x,y:A} x = y \to \mathbf{Type}$

$$\begin{aligned}&\mathrm{E}^{=} : (\Pi_{x:A} C\,(\mathrm{refl}_x)) \to \Pi_{x,y:A}\Pi p : x = y.C\,p\\ &\mathrm{E}^{=}\,f\,\mathrm{refl}_x :\equiv f_x\end{aligned}$$

Now we should be able to perform the usual exercise and reduce the direct definition of uep to one using only the eliminator. The first step is clear, by eliminating one argument we can reduce the problem to:

$$\Pi_{x:A}\Pi q : x = x.\mathrm{refl}_x = q$$

but now we are stuck. We cannot apply the eliminator because we need a family were both indices are arbitrary. Indeed, Hofmann and Streicher were able to show that uep is unprovable from the eliminator. In the next section we will discuss reasons why this is actually not a bad thing. However, at least based on our current understanding of equality it seems that this is an unwanted incompleteness. One which can actually fixed by introducing a special eliminator which works exactly in the case when we want to prove something about equality proofs where both indices are equal. That is we assume as given a family $C : \Pi_{x:A} x = x \to \mathbf{Type}$ and introduce

$$\begin{aligned}&K : (\Pi_{x:A} C\,\mathrm{refl}_x) \to \Pi_{x:A}\Pi p : x = x.C\,p\\ &K\,f\,\mathrm{refl}_x :\equiv f_x\end{aligned}$$

This eliminator is called K because K is the next letter after J.

Exercise 19 *Derive uep using only* $\mathrm{E}^{=}$ *and* K*.*

Exercise 20 *Instead of viewing equality as a relation generated by* refl*, we can also fix one index* $a : A$ *and now define the predicate of being equal to* a*:* $a = - : A \to \mathbf{Type}$*. This predicate is generated by* $\mathrm{refl}_a : a = a$ *so no change here. However, the eliminator looks different. Let's fix* $C : \Pi x : A.a = x \to \mathbf{Type}$

$$\begin{aligned}&J'_a : (C\,\mathrm{refl}_a) \to \Pi_{x:A}\Pi p : a = x.C\,p\\ &J'_a\,f\,(\mathrm{refl}_a) :\equiv f\end{aligned}$$

Show that $\mathrm{E}^{=}$ *and* J' *are interderivable, that is both views of equality are equivalent.*

5 Homotopy Type Theory

5.1 Proof relevant equality

If we are only use $\mathrm{E}^{=}$ aka J we cannot in general prove that there is only one proof of equality, but what can we prove? It turns out that we can indeed verify some equalities:

$$\begin{aligned}
\mathrm{trans}\, p\, \mathrm{refl} &= p \\
\mathrm{trans}\, \mathrm{refl}\, p &= p \\
\mathrm{trans}\, (\mathrm{trans}\, p\, q)\, r &= \mathrm{trans}\, p\, (\mathrm{trans}\, q\, r) \\
\mathrm{trans}\, p\, (\mathrm{sym}\, p) &= \mathrm{refl} \\
\mathrm{trans}\, (\mathrm{sym}\, p)\, p &= \mathrm{refl}
\end{aligned}$$

where $p : a =_A b$, $q : b =_A c$ and $r : c =_A d$.

It is easy to verify these equalities using only J because all the quantifications are over arbitrary $p : x = y$ and there is no repetition of variables.

Exercise 21 *Explicitly construct the proofs using J.*

A structure with these properties is called a groupoid. A groupoid is a category where every morphism is an isomorphism. Here the objects are the elements of the type A, given $a, b : A$ the homset (actually a type) is $a =_A b$, composition is trans, identity is refl and sym assigns to every morphism its inverse.

We can go further an observe that resp also satisfies some useful equalities:

$$\begin{aligned}
\mathrm{resp}\, f\, \mathrm{refl} &= \mathrm{refl} \\
\mathrm{resp}\, f\, (\mathrm{trans}\, p\, q) &= \mathrm{trans}\, (\mathrm{resp}\, f\, p)\, (\mathrm{resp}\, f\, q)
\end{aligned}$$

where $f : A \to B, p : a =_A b, q : b =_A c$.

In categorical terms this means that f is a functor: its effect on objects $a : A$ is $f\, a : B$ and its effect on morphism $p : a =_A b$ is $\mathrm{resp}\, f\, p : f\, a =_B f\, b$.

Exercise 22 *Why don't we prove that* $\mathrm{resp}\, f$ *also preserves symmetries?*

$$\mathrm{resp}\, f\, (\mathrm{sym} p) = \mathrm{sym}\, (\mathrm{resp}\, f\, p)$$

Indeed, the idea of Streicher's and Hofmann's proof is to turn this around and to show that we can generally interpret types as groupoids where the equality type corresponds to the homset. Interestingly we can also interpret J in this setting but clearly we cannot interpret K because it forces the groupoid to be trivial, i.e. to be an equivalence relation. However, in the moment there is no construction which generates non-trivial groupoids, but this will change once we have the univalence principle or if we introduce Higher Inductive Types (HITs).

However, groupoids are not the whole story. There is no reason to assume that the next level of equalities, i.e. the equality of the equality of equality proofs

is trivial. We need to add some further laws, so called coherence laws, which are well known and we end up with a structure which is called a 2-Groupoid (in particular all the homsets are groupoids). But the story doesn't finish here we can go on forever. The structure we are looking for is called a weak ω-groupoid. Alas, it is not very easy to write down what this is precisely. Luckily, homotopy theoreticians have already looked at this problem and they have a definition: a weak ω-groupoid is a Kan complex, that is a simplicial set with all Kan fillers. This is what Voevodsky has used in his homotopical model of Homotopy Type Theory. However, it was noted that this construction uses classical principles, i.e. the axiom of choice. Thierry Coquand and his team have now formulated a constructive alternative which is based on cubical sets.

5.2 What is a proposition?

Previously, we have identified propositions as types but it is fair enough to observe that types have more structure, they can carry more information by having different inhabitants. This becomes obvious in the next exercise:

Exercise 23 *Given* $A, B : \mathbf{Type}$ *and a relation* $R : A \to B \to \mathbf{Type}$ *the axiom of choice can be stated as follows:*

$$(\forall x : A.\exists y : B.R\,x\,y) \to \exists f : A \to B.\forall x : A.R\,x\,(f\,x)$$

Apply the propositions as types translation and prove the axiom of choice.

This is strange because usually the axiom of choice is an indicator of using some non-constructive principle in Mathematics. But now we can actually prove it in Type Theory? Indeed, the formulation above doesn't really convey the content of the axiom of choice because the existential quantification is translated as a Σ-type and hence makes the choice of the witness explicit. In conventional Mathematics propositions do not carry any information hence the axiom of choice has to build the choice function without any access to choices made when showing the premise.

To remedy this mismatch we are more specific about propositions: we say that a type $P : \mathbf{Type}$ is a proposition if it has at most one inhabitant, that is we define

$$\begin{aligned}&\mathrm{isProp} : \mathbf{Type} \to \mathbf{Type}\\ &\mathrm{isProp}\,A :\equiv \Pi x, y : A.x =_A y\end{aligned}$$

I write $P : \mathbf{Prop}$ for a type that is propositional, i.e. we can prove $\mathrm{isProp}\,P$. That is we are interpreting $\mathbf{Prop}$ as $\Sigma P : \mathbf{Type}.\mathrm{isProp}\,P$ but I am abusing notation in that I omit the first projection (that is an instance of subtyping as an implicit coercion).

Exercise 24 *Show that equality for natural number is a proposition, that is establish:*

$$\forall x, y : \mathbb{N}.\mathrm{isProp}\,(x =_{\mathbb{N}} y)$$

Looking back at the proposition as types translation we would like that for any proposition in predicate logic P we have that $[\![P]\!] : \mathbf{Prop}$. That can be shown to be correct for the so called negative fragment, that is the subset of predicate logic without $\vee$ and $\exists$.

Exercise 25 *Show that if* $P, Q : \mathbf{Prop}$ *and* $R : A \to \mathbf{Prop}$ *where* $A : \mathbf{Type}$ *then*

1. $[\![P \implies Q]\!] : \mathbf{Prop}$
2. $[\![P \wedge Q]\!] : \mathbf{Prop}$
3. $[\![\mathrm{True}]\!] : \mathbf{Prop}$
4. $[\![\mathrm{False}]\!] : \mathbf{Prop}$
5. $[\![\forall x : A.P]\!] : \mathbf{Prop}$

However this fails for disjunction and existential quantification: $[\![\mathrm{True} \vee \mathrm{True}]\!]$ is equivalent to Bool which is certainly not propositional since true $\neq$ false. Also $\exists x : \mathrm{Bool}.\mathrm{True}$ is equivalent to Bool and hence also not propositional.

To fix this we introduce a new operation which assigns to any type a proposition which expresses the fact that the type is inhabited. That is given $A : \mathbf{Type}$ we construct $\|A\| : \mathbf{Prop}$, this is called the *propositional truncation* of A. We can construct elements of $\|A\|$ from elements of A, that is we have a function $\eta : A \to \|A\|$. However, we hide the identity of a, that is we postulate that $\eta\, a = \eta\, b$ for all $a, b : A$. How can we construct a function out of $\|A\|$? It would be unsound if we would allow this function to recover the identity of an element. This can be avoided if the codomain of the function is itself propositional. That is given $P : \mathbf{Prop}$ and $f : A \to P$ we can lift this function to $\hat{f} : \|P\| \to A$ with $\hat{f}(\eta\, a) \equiv f\, a$.

Using $\| - \|$ we can redefine $[\![-]\!]$ such that $[\![P]\!] : \mathbf{Prop}$:

$$\begin{aligned} [\![P \vee Q]\!] &:\equiv \|[\![P]\!] + [\![Q]\!]\| \\ [\![\exists x : A.P]\!] &:\equiv \|\Sigma x : A.[\![P]\!]\| \end{aligned}$$

Now the translation of the axiom of choice

$$(\forall x : A.\exists y : B.R\, x\, y) \to \exists f : A \to B.\forall x : A.R\, x\, (f\, x)$$

which is

$$\Pi x : A.\|\Sigma y : B.R\, x\, y\| \to \|\Sigma f : A \to B.\forall x : A.R\, x\, (f\, x)\|$$

is more suspicious. It basically say if for every $x : A$ there is $y : B$ with a certain property, but I don't tell you which one, then there is a function $f : A \to B$ which assigns to every $x : A$ a $f\, x : B$ with a certain property but I don't tell you which function. This sounds like a lie to me!

Indeed lies can make our system inconsistent or they can lead to classical principles, which can be viewed is a form of lying that is not known to be inconsistent. Indeed, assuming one additional principle, *propositional extensionality* we can derive the principle of excluded middle. This proof is due to Diaconescu.

By propositional extensionality we mean that two propositions which are logically equivalent then they are equal:

$$\text{propExt} : \Pi P, Q : \mathbf{Prop}.(P \Leftrightarrow Q) \to P = Q$$

This is reasonable because all what we matters about a proposition is wether it is inhabited, and hence two proposition which are logically equivalent are actually indistinguishable and hence extensionally equal. Indeed, we will see that propExt is a consequence of the univalence principle.

Theorem 1 (Diaconescu) *Assuming the amended translation of the axiom of choice:*

$$\text{ac} : \Pi x : A.\|\Sigma y : B.R\,x\,y\| \to \|\Sigma f : A \to B.\forall x : A.R\,x\,(f\,x)\|$$

and propExt *we can derive the excluded middle for all propositions:*

$$\forall P : \mathbf{Prop}.P \vee \neg P$$

We are going to instantiate A with the type of inhabited predicates over Bool, that is

$$A :\equiv \Sigma Q : \text{Bool} \to \mathbf{Prop}.\exists b : \text{Bool}.Qb$$

$B \equiv \text{Bool}$ and the relation $R : A \to B \to \mathbf{Prop}$ is defined as follows:

$$R\,(Q, q)\,b :\equiv Q\,b$$

that is an inhabited predicate is related to the boolean it inhabits. Can we now prove the premise of the axiom?

$$\Pi x : A.\|\Sigma y : B.R\,x\,y\|$$

that is after plugging in the definition of A, B, R it becomes

$$\Pi(Q, q) : A.\|\Sigma b : B.Q\,b\|$$

and after some currying

$$\Pi Q : \text{Bool} \to \mathbf{Prop}.\|\Sigma b : \text{Bool}.Qb\| \to \|\Sigma b : B.Q\,b\|$$

it is quite obvious that we can.

Now let's look at the conclusion.

$$\|\Sigma f : A \to B.\Pi x : A.R\,x\,(f\,x)\|$$

Let's for the moment ignore the outermost $\|-\|$ and expand the types inside:

$$\Sigma f : A \to \text{Bool}.$$
$$\Pi(Q, q) : A.Q\,(f\,Q)$$

We consider two special predicates $T, F : \text{Bool} \to \mathbf{Prop}$:

$$T\,b :\equiv b = \text{true} \vee P$$
$$F\,b :\equiv b = \text{false} \vee P$$

Where does the P come from? From the beginning of this section: it is the $P : \mathbf{Prop}$ for which we want to show $P \vee \neg P$.

Now applying the conclusion of the axiom to these predicates we obtain two booleans $f\,T, f\,F : \text{Bool}$ and from the second part we know

$$T\,(f\,T) \equiv (f\,T = \text{true}) \vee P$$
$$F\,(f\,F) \equiv (f\,F = \text{false}) \vee P$$

Now let's analyse all the possibilities: there are four combinations:

1. $f\,T = \text{true} \wedge f\,F = \text{false}$
2. $f\,T = \text{true} \wedge P$
3. $P \wedge f\,F = \text{false}$
4. $P \wedge P$

In 2-4 we know that P holds, the only other alternative in which P is not proven is 1. In this case we can show $\neg P$ that is $P \to 0$. For this purpose assume P, in this case both $T\,b$ and $F\,b$ are provable for any b because P is and this means $T\,b \Leftrightarrow F\,b$ which now using propositional extensionality implies $T\,b = F\,b$. But this means that $T = F$ using functional extensionality. This cannot be since we assumed that $f\,T = \text{true}$ and $f\,F = false$, now $T = F$ would imply true = false which is false that is it implies 0. Hence we have shown $\neg P$ by deriving a contradiction from assuming P.

To summarise we have shown $P \vee \neg P$ because in the case 2-4 we have p and in 1 we have $\neg P$. But hang on what about the $||-||$ we have been ignoring? It doesn't matter since $P \vee \neg P$ is already a proposition and putting $||-||$ around it doesn't change anything.

What has really happened is that from the lie that we can recover information we have just hidden we can extract information as long as we hide the function doing the extraction. And this has nothing to do with Σ-types and existentials but with the behaviour of the hiding operation, or inhabitance $||-||$. Hence we can formulate a simpler version of the axiom of choice in Type Theory:

$$(\Pi x : A.\|B\,x\|) \to \|\Pi x : A.B\,x\|$$

This implies the revised translation of the axiom.

5.3 Dimensions of types

We have classified types as propositions if they have at most one inhabitant. uep says that equality is a proposition. Even of we are not accepting uep, we can use this to classify types: we say that a type is a set if all its equalities are propositions.

$$\begin{aligned}&\mathrm{isSet} : \mathbf{Type} \to \mathbf{Type}\\ &\mathrm{isSet}\, A :\equiv \Pi_{x,y:A}\mathrm{isProp}\,(x =_A y)\end{aligned}$$

As for propositions we abuse notation and write $A : \mathbf{Set}$ if $A : \mathbf{Type}$ and we can show $\mathrm{isSet}\, A$. uep basically says that all types are set. The idea here is that sets are types which are quite ordinary, they reflect our intuition that equality is propositional.

Actually we can show that certain types are sets, e.g. exercise 24 basically asks to prove that $\mathbb{N} : \mathbf{Set}$. However, we can do much better, we can show in general that any type with a decidable equality is a set. This is a theorem due to Michael Hedberg.

Theorem 2 (Hedberg) *Given* $A : \mathbf{Type}$ *such that the equality is decidable*

$$d : \forall x, y : A.x = y \vee x \neq y$$

then we can show

$$\mathrm{isSet}\, A$$

To show Hedberg's theorem we establish a lemma saying that if there is a constant function on equality types then the equality is propositional. That is we assume

$$\begin{aligned}&f : \Pi_{x,y:A} x =_A y \to x =_A y\\ &c : \Pi_{x,y:A}\Pi p, q : x =_A y \to f\, p = f\, q\end{aligned}$$

Now for arbitrary $x, y : A, p : x = y$ we can show that $p = \mathrm{trans}\,(f\, p)\,(\mathrm{sym}\,(f\,\mathrm{refl}))$ because using J this reduces to showing that $\mathrm{refl} = \mathrm{trans}\,(f\,\mathrm{refl})\,(\mathrm{sym}\,(f\,\mathrm{refl}))$ which is one of the groupoid properties. Now given any $p, q : x = y$ we can show

$$\begin{aligned}p &= \mathrm{trans}\,(f\, p)\,(\mathrm{sym}\,(f\,\mathrm{refl}))\\ &= \mathrm{trans}\,(f\, q)\,(\mathrm{sym}\,(f\,\mathrm{refl})) \qquad \text{using } c\\ &= q\end{aligned}$$

And hence $=_A$ is propositional.

To construct the function f from decidability we assume $p : x = y$ and apply $d\, x\, y$. Either we have another proof $q : x = y$ and we return this one, or we have that $x \neq y$ but this contradicts that we already have a proof $p : x = y$ and we can use R^0. However, in either case the output doesn't depend on the actual value of the input and hence we can show that the function is constant.

Indeed, if we additionally assume the principle of functional extensionality

$$\text{funExt} : \forall f, g : A \to B.(\forall x : A.f\,x = g\,x) \to f = g$$

we can strengthen Hedberg's theorem:

Theorem 3 *Given* $A : \mathbf{Type}$ *such that the equality is stable*

$$s : \forall x, y : A.\neg\neg(x = y) \to x = y$$

then we can show

$$\text{isSet}\,A$$

This strengthening shows that function types are sets, e.g. $\mathbb{N} \to \mathbb{N} : \mathbf{Set}$ even though its equality is not decidable but it is stable.

To prove the stronger version we observe that for if equality is stable, we can construct the function $g : \Pi x, y : A.x = y \to x = y$ by composing s with the obvious embedding $x = y \to \neg\neg(x = y)$. Since $\neg\neg(x = y)$ is propositional, f must be constant.

Exercise 26 *Show that equality of* $\mathbb{N} \to \mathbb{N}$ *is stable.*

We can extend the hierarchy we have started to construct with **Prop** and **Set**. For example a type whose equalities are sets is called a groupoid (indeed it corresponds to the notion of groupoid which we have introduced previously).

$$\begin{aligned}&\text{isGroupoid} : \mathbf{Type} \to \mathbf{Type}\\ &\text{isGroupoid}\,A :\equiv \forall x, y : A.\text{isSet}\,(x =_A y)\end{aligned}$$

We can also extend this hierarchy downwards, we can redefine a proposition as a type such that its equalities are types with eactly one element - these types are called *contractible*:

$$\begin{aligned}&\text{isContractible} : \mathbf{Type} \to \mathbf{Type}\\ &\text{isContractible}\,A :\equiv \Sigma a : A.\Pi x : A.x = a\end{aligned}$$

$$\begin{aligned}&\text{isProp} : \mathbf{Type} \to \mathbf{Type}\\ &\text{isProp}\,A :\equiv \Pi x, y : A.\text{isContractible}\,(x = y)\end{aligned}$$

We can now define the hierarchy starting with contractible types. For historic reasons (i.e. to be compatible with notions from homotopy theory) we start counting with -2 and not with 0. I am calling the levels dimensions, they are also called truncation levels.

$$\begin{aligned}&\text{hasDimension} : \mathbb{N}_{-2} \to \mathbf{Type} \to \mathbf{Type}\\ &\text{hasDimension}\,(-2)\,A :\equiv \text{isContractible}\,A\\ &\text{hasDimension}\,(n+1)\,A :\equiv \forall x, y : A.\text{hasDimension}\,(x =_A y)\end{aligned}$$

I am writing $\mathbb{N}_{-2}$ for a version of the natural numbers where I start counting with -2. To summarize the definitions so far:

Dimension	Name
-2	Contractible types
-1	Propositions
0	Sets
1	Groupoids

We also introduce the notation $n-\mathbf{Type}$ for $A : \mathbf{Type}$ such that we can show $\mathrm{hasDimension}\, n\, A$.

To convince ourselves that this really is a hierarchy, that is that every $n-\mathbf{Type}$ is also a $n+1-\mathbf{Type}$ we need to show that the hierarchy actually stops at -2 that is that the equality of a contractible type is again contractible.

To show this assume is given a contractible type $A : \mathbf{Type}$ that is we have $a : A$ and $c : \Pi x : A.a = x$. Now we want to show that for all $x, y : A$ the equality $x =_A y$ is contractible, that is we have an element and all other elements are equal. We define $d : \Pi x, y : A.x =_A y$ as $d\, x\, y :\equiv \mathrm{trans}\,(\mathrm{sym}\,(c\, x))\,(c\, y)$. Now it remains to show that $e : \Pi x, y : A.\Pi p : x =_A y.d\, x\, y = p$. Using J we can reduce this to $d\, x\, x = \mathrm{refl}\, x$ unfolding $d\, x\, x \equiv \mathrm{trans}\,(\mathrm{sym}\,(c\, x))\,(c\, x)$ we see that this is an instance of one of our groupoid laws.

As a corollary we obtain that $\mathrm{hasDimension}\, n\, A$ implies $\mathrm{hasDimension}\,(n+1)\, A$.

5.4 Extensionality and univalence

As I have already mentioned in the introduction: extensionality means that we identify mathematical objects which behave the same even of they are defined differently. An example are the following two functions:

$$f, g : \mathbb{N} \to \mathbb{N}$$
$$f\, x :\equiv x + 1$$
$$g\, x :\equiv 1 + x$$

We can show that $\forall x : \mathbb{N}.f\, x = g\, x$ but can we show that $f = g$? The answer is **no**, because if there were a proof without any assumption it would have to be refl and this would only be possible if $f \equiv g$ but it is clear that they are not definitionally equal. However, we cannot exhibit any property not involving this equality which would differentiate them. Another example are the following two propositions:

$$P, Q : \mathbf{Prop}$$
$$P :\equiv \mathrm{True}$$
$$Q :\equiv \neg\mathrm{False}$$

Again we cannot show that $P \equiv Q$ even though there clearly is no way to differentiate between them.

We have already mentioned the two principles which are missing here':

$$\text{funExt} : \forall f, g : A \to B.(\forall x : A.f\,x = g\,x) \to f = g$$
$$\text{propExt} : \Pi P, Q : \mathbf{Prop}.(P \Leftrightarrow Q) \to P = Q$$

We note that the corresponding principles are true in set theory, which seems to contradict my statement that Type Theory is better for extensional reasoning than set theory. However, this shortcoming can be fixed since all constructions preserve these equalities. This can be made precise by interpreting the constructions in the setoid model, where every type is modelled by a set with an equivalence relation. In this case we can model function types by the set of functions and equality is extensional equality. The same works for **Prop**, a proposition is modelled by the set of propositions identified if they are logically equivalent.

However, the shortcoming of set theory becomes obvious if we ask the next question: when are two sets equal? For example

$$A, B : \mathbf{Set}$$
$$A :\equiv \overline{1} + \overline{2}$$
$$B :\equiv \overline{2} + \overline{1}$$

We have no way to distinguish two sets with 3 elements, hence following the same logic as above they should be equal. However, they are not equal in set theory under the usual encoding of finite sets and + and it would be hard to fix this in general.

When are two sets equal? Given $f : A \to B$ we say that f is an isomorphism if there it has an inverse, that is [5]

$$\begin{aligned}&\text{isIso} : (A \to B) \to \mathbf{Type}\\ &\text{isIso}\, f :\equiv \quad \Sigma g : B \to A\\ &\qquad\qquad\quad \eta : \Pi x : B.f\,(g\,x) = x\\ &\qquad\qquad\quad \epsilon : \Pi x : A.g\,(f\,x) = x\end{aligned}$$

And we define $A \simeq B :\equiv \Sigma f : A \to B.\text{isIso}\, f$. Using isIso we can formulate a new extensionality principle: we want to say that there is an isomorphism between isomorphism of sets and equality of sets. Indeed, we can observe that the is a function from equality of sets to isomorphism because every set is isomorphic to itself using J.

Exercise 27 *Define* $\text{eq2iso} : \Pi_{A,B:\mathbf{Set}} A = B \to A \simeq B$

Using this we can state extensionality for sets

$$\text{extSet} : \text{isIso}\,\text{eq2iso}$$

As a corollary we get that $A \simeq B \to A = B$. Since $\overline{1} + \overline{2} \simeq \overline{1} + \overline{2}$ we can show that $\overline{1} + \overline{2} = \overline{1} + \overline{2}$

[5] I am using here a record like syntax for iterated Σ-types.

Exercise 28 *Show that*

1. $(A + B) \to C = (A \to C) \times (B \to C)$
2. $(A \times B) \to C = A \to B \to C$
3. $1 + \mathbb{N} = \mathbb{N}$
4. $\mathbb{N} \times \mathbb{N} = \mathbb{N}$
5. $\mathrm{List}\,\mathbb{N} = \mathbb{N}$
6. $\mathbb{N} \to \mathbb{N} \neq \mathbb{N}$

assuming $A, B, C : \mathbf{Set}$. *For which of the results do we not need* extSet*?*

Exercise 29 *An alternative to isomorphism is bijection, one way to say that a function is bijective is to say there is a unique element in the domain which is mapped to an element of the codomain. We define* $\exists!$ *(exists unique) :*

$$\exists! x : A.P\,x :\equiv \exists x : A.P\,x \wedge \forall y : A, P\,y \implies x = y$$

and using this we define what is a bijection:

$$\begin{aligned}&\mathrm{isBij} : (A \to B) \to \mathbf{Prop}\\ &\mathrm{isBij}\,f :\equiv \forall y : B.\exists! x : A.f\,x = y\end{aligned}$$

Show that isomorphism and bijection are logically equivalent:

$$\forall\, f : A \to B.\mathrm{isBij}\,f \Leftrightarrow \mathrm{isIso}\,f$$

extSet is incompatible with uniqueness of equality proofs (uep) because there are two elements of $\overline{2} \simeq \overline{2}$, namely identity and negation. If we assume that there is only one proof of equality for $\overline{2} = \overline{2}$ then we also identify this two proofs and hence $0_2 = 1_2$ which is inconsistent.

So far we have only considered sets what about types in general? There is a twist: if equality is not propositional then the η and ϵ components of isIso are not propositional in general. Indeed, assuming extSet for all types is unsound. Instead we need to refine the notion of isomorphism by introducing an extra condition which relates η and ϵ. We call this *equivalence* of types.

$$\begin{aligned}&\mathrm{isEquiv} : (A \to B) \to \mathbf{Type}\\ &\mathrm{isEquiv}\,f :\equiv \quad \Sigma g : B \to A\\ &\qquad\qquad\quad \eta : \Pi x : B.f\,(g\,x) = x\\ &\qquad\qquad\quad \epsilon : \Pi x : A.g\,(f\,x) = x\\ &\qquad\qquad\quad \delta : \Pi x : A.\eta\,(f\,x) = \mathrm{resp}\,f\,(\epsilon\,x)\end{aligned}$$

And we define $A \cong B :\equiv \Sigma f : A \to B.\mathrm{isEquiv}\,f$.

Exercise 30 *Define* eq2equiv : $\Pi_{A,B:\mathbf{Set}} A = B \to A \cong B$

We can now state extensionality for types, which is what is commonly called univalence.

$$\text{univalence} : \text{isEquiv}\,\text{eq2equiv}$$

As before for extSet a consequence is that equivalence of types implies equality $A \cong B \to A = B$ which is what most people remember about univalence. In the special case of sets equivalence and isomorphism agree because the type of δ is an equivalence, that is for sets we have $(A \simeq B) = (A \cong B)$. But more is true, even for types in general equivalence and isomorphism are logically equivalence, that is $(A \simeq B) \Leftrightarrow (A \cong B)$. While this may be surprising at the first glance, it just means that we can define functions in both directions but they are not inverse to each other. Indeed, in general there are more proofs of an isomorphism that of an equivalence, indeed isEquiv f is a proposition, while isIso f in general isn't (but it is if f is a function between sets). However, the logical equivalence of equivalence and isomorphism means that to establish an equivalence all we need to do is to construct an isomorphism. In particular all the equalities of exercise 28 hold for types in general.

The definition of isEquiv looks strangely assymmetric. Indeed, we could have replaced δ with its symmetric twin:

$$\delta' : \Pi y : A.\text{resp}\, g\, \eta\, y = \epsilon\, (g\, y)$$

Indeed, we can either use δ or δ' it doesn't make any difference. Why don't we use both? Indeed, this would exactly be the definition of an adjunction between groupoids. However, assuming both messes everything up, now isEquiv f is no longer a proposition and we need to add higher level coherence equations to fix this. Indeed, there is such an infinitary (coinductive) defintion of equivalence. But we don't need to use this, the asymmetric definition (or its mirror image) does the job.

Exercise 31 *We can extend exercise 29 to the case of equivalences by taking the equality proofs into account. That is we say that not only there is a unique inverse but that the pair of inverses and the proof that there are an inverse are unique that is contractible. We define*

$$\begin{aligned}&\text{isEquiv}' : (A \to B) \to \mathbf{Type}\\ &\text{isEquiv}'\, f :\equiv \Pi y : B.\text{isContr}\,(\Sigma x : A.f\, x = y)\end{aligned}$$

We have observed that already extSet implies that there are non-propositional equality, e.g. $\overline{2} = \overline{2}$. In other words the first universe $\mathbf{Type}_0$ is not a set. Nicolai Kraus has generalized this and shown that using univalence we can construct types which are non $n-\mathbf{Type}$ for any $n : \mathbb{N}$. I leave is an exercise to do the first step in this construction:

Exercise 32 *Show that the equalities of* $\Sigma X : \mathbf{Type}_0.X = X : \mathbf{Type}_1$ *are not always sets and hence* $\mathbf{Type}_1$ *is not a groupoid.*

5.5 Higher Inductive Types

In HoTT we view a type together with its equality types with the structure of an ω-groupoid. Hence when we introduce a new type we also have to understand what it's equality types are. So for example in the case of $\to$ and $\times$ we have the following equations:

$$\begin{aligned} f =_{A\to B} g &= \Pi x, y : A.x =_A y \to f\,x = g\,x \\ (a, b) =_{A\times B} (a', b') &= (a =_A a') \times (b =_B b') \end{aligned}$$

When defining types inductively, that is by given the constructors, we can also add constructors for equalities. The simplest example for this is propositional truncation $\|A\|$, which can be inductively defined by the following constructors:

$$\begin{aligned} \eta &: A \to \|A\| \\ \mathrm{irr} &: \Pi x, y : \|A\|.x =_{\|A\|} y \end{aligned}$$

To define a non-dependent function $f : \|A\| \to B$ we need an function $g : A \to B$ but also an equalities $i : \Pi x, y : B.x =_B y$ which correspond to saying that $B : \mathbf{Prop}$. f will satisfy the following equations:

$$\begin{aligned} &f\,(\eta a) :\equiv g\,a \\ &\mathrm{resp}\,f\,(\mathrm{irr}\,x\,y) :\equiv i\,(f\,x)\,(f\,x) \end{aligned}$$

Hence the eliminator does not only determine the behaviour of f on elements but also on equalities. This makes perfect sense because in the view of types as groupoids functions are functors. We can also turn this into a non-dependent eliminator:

$$\begin{aligned} &\mathrm{R}^{\|A\|} : (A \to B) \to (\Pi x, y : B.x = y) \to \|A\| \to B \\ &\mathrm{R}^{\|A\|}\,g\,i\,(\eta a) :\equiv g\,a \\ &\mathrm{resp}\,\mathrm{R}^{\|A\|}\,g\,i\,(\mathrm{irr}\,x\,y) :\equiv i\,(\mathrm{R}^{\|A\|}\,g\,i\,x)\,(\mathrm{R}^{\|A\|}\,g\,i\,y) \end{aligned}$$

We can also derive a dependent eliminator, for which we need a dependent version of resp.

Another class for examples for higher inductive types are set quotients, that is we have a set $A : \mathbf{Set}$ and a relation $R : A \to A \to \mathbf{Prop}$, we define $A/R : \mathbf{Set}$ by the constructors

$$\begin{aligned} &[-] : A \to A/R \\ &[-]^{=} : \Pi x, y : A.Rxy \to [x] =_{A/R} [y] \\ &\mathrm{set} : \Pi_{x,y:A/R}\Pi p, q : x =_{A/R} y.p =_{x=_{A/R}y} q \end{aligned}$$

Note that the last constructor is necessary to make sure that the type we construct is actually a set. We do not require that R is an equivalence relation but obviously the equality on A/R always is.

Exercise 33 *Derive the non-dependent eliminator for set quotients.*

An example is the definition of the integer as a quotient of pairs of integers:

$$\mathrm{eq}_{\mathbb{Z}} : (\mathbb{N} \times \mathbb{N}) \to (\mathbb{N} \times \mathbb{N}) \to \mathbf{Prop}$$
$$\mathrm{eq}_{\mathbb{Z}}\,(a^+, a^-)\,(b^+, b^-) :\equiv a^+ + b^- = b^= + a^- \qquad \mathbb{Z} = (\mathbb{N} \times \mathbb{N})/\mathrm{eq}_{\mathbb{Z}}$$

The idea is that a pair of natural numbers (a^+, a^-) represents its difference $a^+ - a^-$ and we identify pairs which have the same difference.

Exercise 34 *Define addition and multiplication and additive inverses on $\mathbb{Z}$. How would you prove the group laws?*

Higher Inductive Types even for sets are more general than set quotients because we can define elements and equalities at the same time. An example for this are infinite branching trees with permutations. That is we define infinitely branching trees

$$\mathrm{Tree} : \mathbf{Type}$$
$$\mathrm{leaf} : \mathrm{Tree}$$
$$\mathrm{node} : (\mathbb{N} \to \mathrm{Tree}) \to \mathrm{Tree}$$

We inductively define a relation which identifies trees upto permutations of subtrees:

$$- \sim - : \mathrm{Tree} \to \mathrm{Tree} \to \mathbf{Prop}$$
$$\mathrm{perm} : \Pi_{g:\mathbb{N}\to\mathrm{Tree}} \Pi f : \mathbb{N} \to \mathbb{N}.\mathrm{isIso}\, f \to \mathrm{node}\, f \sim \mathrm{node}\,(\lambda n.g\, f\, n)$$
$$\mathrm{respNode} : \Pi_{g,h:\mathbb{N}\to\mathrm{Tree}} (\Pi i : \mathbb{N}.f\ i\ g\ i) \to \mathrm{node}\, f \sim \mathrm{node}\, g$$

Now trees with permutations can be defined as $\mathrm{PTree} :\equiv \mathrm{Tree}/\sim$. However, it turns out that this type is not as useful as we might hope. Let's say we want to lift the node construction to PTree that is we want to define

$$\mathrm{node}' : (\mathbb{N} \to \mathrm{PTree}) \to \mathrm{PTree}$$

such that $\mathrm{node}'(\lambda n.[f\, n]) = [\mathrm{node}\, f]$. However, we have a problem, to define node' we need to commute the quotient definition and function type, that is we need to go from $\mathbb{N} \to \mathrm{Tree}/\mathrm{PermTree}$ to a quotient on $\mathrm{Nat} \to \mathrm{Tree}$. This corresponds to an instance of the axiom of choice, in this case this is called countable choice because the indexing type A is $\mathbb{N}$.

Exercise 35 *Show that we can define* node' *using countable choice.*

However, using HITs we have an alternative: we can define permutable trees by introducing the elements and the equalities at the same time:

$$\mathrm{PTree} : \mathbf{Type}$$
$$\mathrm{leaf} : \mathrm{PTree}$$
$$\mathrm{node} : (\mathbb{N} \to \mathrm{PTree}) \to \mathrm{PTree}$$
$$\mathrm{perm} : \Pi_{g:\mathbb{N}\to\mathrm{Tree}} \Pi f : \mathbb{N} \to \mathbb{N}.\mathrm{isIso}\, f \to \mathrm{node}\, f = \mathrm{node}\,(\lambda n.g\, f\, n)$$

We don't need to add a constructor corresponding to respNode because we know that any function respects equality. We don't ned to derive node$'$ because this is already captured by node itself. In the HoTT book an instance of this idea is used to define the Cauchy Reals. The naive definition identifies converging sequences of rational numbers but we can't show that this definition is Cauchy complete that is that we can identify converging sequences of real numbers. Again this can be fixed by using countable choice or alternatively by a HIT that defines the reals and theor equality at the same time, now Cauchy completeness is simply a constructor.

All the examples of HITs we have seen so far are not really *higher* because they work on the level of sets. The only good examples of true HITs I know come from *synthetic homotopy theory*, here we use HoTT to model constructions in Homotopy theory. In homotopy theory we look at the path spaces of geometric objects: we start with paths between points and then we can have paths between paths which are continuous transformation of paths. The main idea behing synthetic homotopy theory is to identify paths with equality types.

A simple example is the circle. The paths on a circle correspond to the integers, they measure how many times we move around the circle, we have to use integers because we can go forwards and backwards. In HoTT the circle can be defined as the following HIT:

$$\begin{aligned} &S^1 : \mathbf{Type} \\ &\mathrm{base} : S^1 \\ &\mathrm{loop} : \mathrm{base} =_{S^1} \mathrm{base} \end{aligned}$$

Intuitively we imagine that the circle is generated by a point base and by a path loop from the point to itself. S^1 is not a set because there is no reason to assume that loop and refl are equal and also loop and sym loop are not equal. Indeed, we can define a function

$$\begin{aligned} &\mathrm{encode} : \mathbb{Z} \to \mathrm{base} = \mathrm{base} \\ &\mathrm{encode}\, n :\equiv \mathrm{loop}^n \\ &\mathrm{encode}\, 0 :\equiv \mathrm{refl} \\ &\mathrm{encode}\, (-n) :\equiv \mathrm{loop}^{n-1} \end{aligned}$$

To define a function in the other direction is a bit more tricky. We cannot define a function directly from base = base because none of the indices is variable. However, using the result from exercise 20 it is enough if one of the indices is variable. That is we can use base $= x$. However, we also have to generalize the left hand side to depend on x. We define a family $X : S^1 \to \mathbf{Type}$ s.t. X base are the integers. We also need to interpret loop as an equality of types $\mathbb{Z} = \mathbb{Z}$. We don't want to use the trivial equality because the isomorphism should represent the number of times we have used the loop (and its direction). We observe that $\lambda x.x + 1, \lambda x.x - 1 : \mathbb{Z} \to \mathbb{Z}$ are inverses hence using univalence (actually set extensionality is enough) we can derive a non-trivial proof $p : \mathbb{Z} = \mathbb{Z}$. Hence to

summarize we define

$$X : S^1 \to \mathbf{Type}$$
$$X\,\mathrm{base} :\equiv \mathbb{Z}$$
$$\mathrm{resp}\,X\,\mathrm{loop} :\equiv p$$

We define our generalised decode function using equality elimination

$$\mathrm{decode}' : \Pi_{x:S^1}\,\mathrm{base} = x \to X\,x$$
$$\mathrm{decode}'\,\mathrm{refl} :\equiv 0$$

Indeed, $X\,\mathrm{base} \equiv \mathbb{Z}$ and for refl we want to return 0. We obtain now decode by instantiating decode$'$:

$$\mathrm{decode} : \mathrm{base} = \mathrm{base} \to \mathbb{Z}$$
$$\mathrm{decode} :\equiv \mathrm{decode}'_{\mathrm{base}}$$

While it is well known in Homotopy Theory that the group of paths of the circle is the integers the particular proof is due to Dan Licata who formalized in in Agda, a *deformalized* version is described in the HoTT book. Dan also uses dependent elimination to show that encode and decode are indeed inverse to each other. This shows that $(\mathrm{base} =_{S^1} \mathrm{base}) = \mathbb{Z}$. However, it is not always the case that the loop space of a HIT is a set it can be a higher dimensional type again. This is especially true once we move from the circle to the sphere S^2 which can be defined as follows:

$$S^2 : \mathbf{Type}$$
$$\mathrm{base} : S^1$$
$$\mathrm{loop} : \mathrm{refl}_{\mathrm{base}} =_{\mathrm{base}=_{S^2}\mathrm{base}} \mathrm{refl}_{\mathrm{base}}$$

The idea is that surface of the sphere can be generated by a higher path which maps the empty path on the pole onto itself. Now it may appear that the sphere is quite uninteresting because all the paths $\mathrm{base} =_{S_2} \mathrm{base}$ can be transformed into refl. However, we cannot prove that this type is contractible, that is has exactly on element. The reason is that it has non-trivial higher path spaces, and indeed lots of them. From Homotopy theory it is known that all higher path spaces of the sphere are non-trivial hence S^2 is non an $n-\mathbf{Type}$ for any n.

5.6 Example: Definition of the Integers

As a more mundane example we consider the definition of the integers ($\mathbb{Z} :$ **Type**) as a HIT. We have the following element constructors

$$0 : \mathbb{Z}$$
$$\mathrm{suc} : \mathbb{Z} \to \mathbb{Z}$$
$$\mathrm{pred} : \mathbb{Z} \to \mathbb{Z}$$

To get the type we want we need to add some constructors for equalities, namely:

$$\begin{aligned}\mathrm{sucpred} &: \Pi i : \mathbb{Z}.\mathrm{suc}\,(\mathrm{pred}\,i) =_{\mathbb{Z}} i\\ \mathrm{predsuc} &: \Pi i : \mathbb{Z}.\mathrm{pred}\,(\mathrm{suc}\,i) =_{\mathbb{Z}} i\end{aligned}$$

However we are not done yet! The type we have constructed so far is certainly not a set. For example there are two ways to prove

$$\mathrm{suc}\,(\mathrm{pred}\,(\mathrm{suc}\,0)) = \mathrm{suc}\,0$$

namely

$$\begin{aligned}&\mathrm{sucpred}\,(\mathrm{suc}\,0)\\ &\mathrm{resp}\,\mathrm{suc}\,(\mathrm{predsuc}\,0)\end{aligned}$$

We certainly want the integers to be a set, in particular they should have a decidable equality. One way to achieve this would be to equate all proofs by adding another higher constructor:

$$\mathrm{isSet} : \Pi i, j : \mathbb{Z}.\Pi p, q : i =_{\mathbb{Z}} j \to p =_{i=_{\mathbb{Z}}j} q$$

We call such a first order HIT a Quotient Inductive Type (QIT).

As an example let us define addition for integers. I will do this in a pattern matching style. It is certainly a good idea to derive the eliminator and check that the definitions can be reduced to this but just writing down the eliminator is quite a mouthful.

To define $\_ + \_ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ we start with the element constructors:

$$\begin{aligned}0 + i &:\equiv i\\ \mathrm{suc}\,j + i &:\equiv \mathrm{suc}\,(i + j)\\ \mathrm{pred}\,j + i &:\equiv \mathrm{pred}\,(i + j)\end{aligned}$$

To handle the equational constructor we overload functions so that we can also apply them to equalities. In this case we consider the function $\_ + i$ for some $i : \mathbb{Z}$, now given an equality $p : j =_{\mathbb{Z}} k$ we write $p + i : j + i = k + i$ as a shorthand for $\mathrm{resp}\,(\_ + i)\,p$. Using this notation we can give definitions for the equational constructors as well:

$$\begin{aligned}\mathrm{sucpred}\,j + i &:\equiv \mathrm{sucpred}\,(j + i)\\ \mathrm{predsuc}\,j + i &:\equiv \mathrm{predsuc}\,(j + i)\end{aligned}$$

We also have to deal with the isSet constructor and indeed we can extend the overload the notation also to deal with hight proofs. However, all what we need to notice that the constructor forces us to show that the codomain of the operation is a set, which it is since it is the integers themselves.

Exercise 36 *Define the additive inverse $-\_ : \mathbb{Z} \to \mathbb{Z}$ in the same style.*

Now let's prove something about addition, for example that it is associative.

$$\text{assoc} : \Pi_{i,j,k:\int}.(i + j) + k =_{\mathbb{Z}} i + (j + k)$$

We are in for a pleasant surprise: since the codomain of associativity is a proposition we don't have to do anything for the equational constructors because the equations they induce hold trivially. Hence it is enough to only consider element constructors:

$$\begin{aligned} \text{assoc}_0 &:\equiv \text{refl} \\ \text{assoc}_{\text{suc}\, i} &:\equiv \text{suc}\,(\text{assoc}_i) \\ \text{assoc}_{\text{pred}\, i} &:\equiv \text{pred}\,(\text{assoc}_i) \end{aligned}$$

Note that in the definition above we again lift operations to equalities.

Exercise 37 *Show that the additive inverse behaves properly, that is*

$$\Pi_{i:\mathbb{Z}}(-i) + i = 0$$

To show that it is also the 2nd equation hold you need to prove the lemmas you need to prove commutativity. Hence as an extended exercise prove that the integers form a commutative group.

To show that the equality of integers is decidable we need to define a normalisation function. A reasonable notion of normal form is signed numbers, that is we define $\mathbb{Z}^{\text{n}f} : \mathbf{Type}$ by the following constructors:

$$\begin{aligned} 0 &: \mathbb{Z}^{\text{n}f} \\ +\_ &: \mathbb{N} \to \mathbb{Z}^{\text{n}f} \\ -\_ &: \mathbb{N} \to \mathbb{Z}^{\text{n}f} \end{aligned}$$

$\mathbb{Z}^{\text{n}f}$ is a set because all inductive definitions which don't refer to higher types are automatically sets. We can easily embed the normal forms into the integers, by defining the functions $\text{pos}, \text{neg} : \mathbb{N} \to \mathbb{Z}^{\text{n}f}$ which repeat the predecessor n-times

$$\begin{aligned} \text{pos}\, 0 &:\equiv 0 \\ \text{pos}\,(\text{suc}\, n) &:\equiv \text{suc}\,(\text{pos}\, n) \\ \text{neg}\, 0 &:\equiv 0 \\ \text{neg}\,(\text{suc}\, n) &:\equiv \text{pred}\,(\text{neg}\, n) \end{aligned}$$

Putting everything together we define $\text{emb} : \mathbb{Z}^{\text{n}f} \to \mathbb{Z}$ by

$$\begin{aligned} \text{emb}\, 0 &:\equiv 0 \\ \text{emb}\,(+i) &:\equiv \text{pos}\,(\text{suc}i) \\ \text{emb}\,(-i) &:\equiv \text{neg}\,(\text{suc}i) \end{aligned}$$

Exercise 38 *Define a function* $\mathrm{nf} : \mathbb{Z} \to \mathbb{Z}^{\mathrm{n}f}$ *that is inverse to* emb*, i.e. establish the following propositions:*

$$\Pi_{i:\mathbb{Z}} \mathrm{emb}\,(\mathrm{nf}\,i) = i$$
$$\Pi_{i:\mathbb{Z}^{\mathrm{n}f}} \mathrm{nf}\,(\mathrm{emb}\,i) = i$$

You may notice that using univalence we have established that $\mathbb{Z} = \mathbb{Z}^{\mathrm{n}f}$, hence what is the point of defining the integers as a HIT? It turns out that while the two objects are extensionally equivalent they are intensionally different and the HIT definition is much easier to work with.

Let's go back to the definition of the integers. When we added the constructor isSet we forced $\mathbb{Z}$ to be a set but by using a steam hammer. This is bad consequences when we want to define functions from the integers into a type which is not a set like S^1 earlier. The only way to do this would be going via the normal form because we can eliminate into any type form them. However, this spoils the advantages we just claimed form using the HIT. Luckily, there is another way which basically means we solve a coherence problem. Instead of isSet we add the following constructor:

$$\mathrm{coh} : \mathrm{sucpred}\,(\mathrm{suc}\,i) = \mathrm{resp}\,\mathrm{suc}\,(\mathrm{predsuc}\,i)$$

This is enough to turn $\mathbb{Z}$ into a set. Indeed, the constructors pred, predsuc, sucpred together just state that suc is an equivalence [6]. We can see this by adopting the normalisation proof to this setting. This can be substantially simplified by observing that the constructors that state that suc is an equivalence form a proposition.

We end with an open problem which I have learned from Paolo Capriotti. When moving from natural numbers to lists we are defining the free monoid over an arbitrary set of generators. We can do the same with the integers and define the free group $\mathrm{FG}\,A$ over a type A by labelling suc and pred in the definition with elements of A. So in the way we have a cons and an anti-cons and equations that they cancel each other. As before we are saying that $\mathrm{suc}\,a$ is an equivalence. Now the question is given $A : \mathbf{Set}$ is it the case that $\mathrm{FG}\,A$ is a set? If A has a decidable equality we can still define a normalisation function and establish the property this way. However, we don't know how to do this in the general case. Maybe it isn't even true?

References

[1] Paul Richard Halmos. *Naive set theory.* Springer Science & Business Media, 1998.

[2] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* First edition, 2013.

[6] This observation and much more is due to Paolo Capriotti