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. [4] It is interesting that the reverse direction

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

is actually provable - for the details we need a better understanding of $||-||$. However, it should be intuitively clear that from a hidden function we can generate hidden information but not vice versa.

4.6 Induction is recursion

In section 3.7 we analysed primitive recursion for simple types. That is we explained that for example to define a function from the natural numbers to another type let's say $f : \mathbb{N} \to A$, we need to say what the function is doing for 0 hence we need an $z : A$ and we need to say what the function is doing for the successor assuming we already know the result for the previous number, that menas we need a function $s : A \to A$ and we can define f by primitive recursion as:

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

We also discovered that we can turn this principle into a higher order function which I called iter which for any type A has the type

$$\mathrm{iter}_A : A \to (A \to A) \to \mathbb{N} \to A$$

actually now that we know about universes we can get rid of the ad hoc treatment of the type and just say

$$\mathrm{iter} : \Pi_{A:\mathbf{Type}} A \to (A \to A) \to \mathbb{N} \to A$$

which is defined as

$$\begin{aligned} \mathrm{iter}_A\,a\,f\,0 &:\equiv a \\ \mathrm{iter}_A\,a\,f\,(\mathrm{suc}\,n) &:\equiv f\,(\mathrm{iter}\,a\,f\,n) \end{aligned}$$

[4]It is actually equivalent if we drop the restriction that R is propositional.

We can define f just using iter:

$f :\equiv \mathrm{iter}\, z\, s$

Now let's switch tack and say we want to define a dependent function $f : \Pi n : \mathbb{N}.A\, n$ where $A : \mathbb{N} \to \mathbf{Type}$ is a dependent type. As before we need to say what f does for 0 hence we need $z : A\, 0$ and we need to say how to compute $f\,(\mathrm{suc}\, n)$ from $f\, n$ and that means we need $s : \Pi n : \mathbb{N}.A\, n \to A\,(\mathrm{suc}\, n)$. Given these ingredients we can define f by dependent primitive recursion:

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

Indeed, this is almost the same as before: the only difference is that we need to communicate the natural number which we use to s, which if you remember makes it more like prec. I often hide the first argument to s in which case the definition looks exactly the same.

We have already seen some examples. In the beginning of the chapter we defined zeros : $\Pi n : \mathbb{N}.\mathrm{Vec}\,\mathbb{N}\, n$ using dependent primitive recursion:

$$\begin{aligned} \mathrm{zeros}\, 0 &:\equiv [] \\ \mathrm{zeros}\,(1+n) &:\equiv 0 :: (\mathrm{zeros}\, n) \end{aligned}$$

In this case $z \equiv []$ and $s\, n\, x :\equiv 0 ::_n x$. Another example using propositions as types I need to define a dependent function to show $\Pi n : \mathbb{N}.n = n + 0$. We defined $f : \Pi n : \mathbb{N}.n = n + 0$ as:

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

So here $z \equiv \mathrm{refl}\, 0$ and $s\, n\, x :\equiv \mathrm{resp}\,\mathrm{suc}\, x$.

And again as before we can define one define one higher order function which implements dependent primitive recursion: this is called elim which is short for eliminator:

$\mathrm{elim} : \Pi A : \mathbb{N} \to \mathbf{Type}.A\, 0 \to (\Pi n : \mathbb{N}.A\, n \to A\,(\mathrm{suc}\, n)) \to \Pi n : \mathbb{N}.A\, n$

which is defined in a way very similar to iter:

$$\begin{aligned} \mathrm{elim}\, A\, z\, s\, 0 &:\equiv z \\ \mathrm{elim}\, A\, z\, s\,(\mathrm{suc}\, n) &:\equiv s\, n\,(\mathrm{elim}\, A\, z\, s\, n) \end{aligned}$$

We can put elim to work to derive the previous examples, in the case of zeros : $\Pi n : \mathbb{N}.\mathrm{Vec}\,\mathbb{N}\, n$ we use

$\mathrm{zeros} :\equiv \mathrm{elim}\,(\mathrm{Vec}\,\mathbb{N})\,[]\,(\lambda n, x.0 ::_n x)$

and in the case of $f : \Pi n : \mathbb{N}.n = n + 0$:

$f :\equiv \mathrm{elim}\,(\lambda n.n = n + 0)\,(\mathrm{refl}\, 0)\,(\lambda n, x.\mathrm{resp}\,\mathrm{suc}\, x)$

It is worthwhile to look at the type of elim and notice that it exactly corresponds to the principle of induction for natural numbers using the propositions as types view. Or using our refined version we could say that induction is the special case of dependent primitive recursion in the case of a propositional family $A : \mathbb{N} \to \mathbf{Prop}$, which is what we used in the 2nd case.

While this is not a hard theorem it is an important observation that the proposition as types views shows us that induction is just a special case of dependent primitive recursion. This hopefully correspond to our naive understanding of induction, which is true because we can use recursion to repeat the inductive step as many times as needed before resorting to the base case.

As we were able to extend primitive recursion to other datatypes we can do the same in the for the other datatypes we looked at. To define a function out of a datatype we only need to say what it is doing for the constructors. We can condense this into an eliminator which is a higher order function implementing this reduction.

What is the point of an eliminator? Using eliminators we can reduce the definability of a function to a formal criterion, the same way as we reduced primitive recursion to just using one constant, iter. In practice we don't want to use eliminators all the time but come up with nice and readable definitions which can be reduced to eliminators. But if in doubt we better provide a translation into the use of basic eliminators to make sure that we are not cheating.

As an example let's derive the eliminator for lists, which with our propositions as types glasses will correspond to an induction principle for lists. On the other hand it should generalize the fold we have already seen. But let's just go through the basic motions. To define a dependent function $f : \Pi x : \text{List}\,A.B\,x$ where $B : \text{List}\,A \to \mathbf{Type}$ we need to say what f is doing for the constructors of List A that is we have to complete the lines

$$\begin{aligned} f\,[] &:\equiv ?_0 \\ f\,(a :: l) &:\equiv ?_1 \end{aligned}$$

Clearly $?_0 : B[]$ so lets just assume we have $?_0 \equiv n : B\,[]$. To complete $?_1 : B\,(a :: l)$ we can use the result of the recursive call $f\,l : B\,l$ and we can also use $a : A$, that is we need a function $c : \Pi a : A, \Pi l : \text{List}\,A.B\,l \to B\,(a :: l)$ and we set $?_1 \equiv c\,a\,l\,(f\,l)$. To put it all together we arrive at the following definition of f

$$\begin{aligned} f\,[] &:\equiv n \\ f\,(a :: l) &:\equiv c\,a\,l\,(f\,l) \end{aligned}$$

given

$$\begin{aligned} &n : B\,[] \\ &c : \Pi a : A, \Pi l : \text{List}\,A.B\,l \to B\,(a :: l) \end{aligned}$$

Now to condense everything into an eliminator we make all the parameters explicit which includes the type parameters:

$$\begin{aligned}\mathrm{elim}^{\mathrm{List}} : \quad & \Pi_{A:\mathbf{Type}}\Pi B : \mathrm{List}\,A \to \mathbf{Type}.B\,[\,]\\ & \to (\Pi a : A, l : \mathrm{List}\,A.B\,l \to B\,(a :: l))\\ & \to \Pi l : \mathrm{List}\,A.B\,l\end{aligned}$$

$$\begin{aligned}\mathrm{elim}^{\mathrm{List}}\,B\,n\,c\,[\,] &:\equiv n\\ \mathrm{elim}^{\mathrm{List}}\,B\,n\,c\,(a :: l) &:\equiv c\,a\,l\,(\mathrm{elim}^{\mathrm{List}}\,B\,n\,c\,l)\end{aligned}$$

There is one more example I would like to cover and that is to show an eleminator for an inductively defined family like $\mathrm{Fin} : \mathbb{N} \to \mathbf{Type}$. To remind us: Fin is inductively generated by:

$$\begin{aligned}0 &: \Pi_{n:\mathbb{N}}\mathrm{Fin}\,1+n\\ \mathrm{suc} &: \Pi_{n:\mathbb{N}}\mathrm{Fin}\,n \to \mathrm{Fin}\,(1+n)\end{aligned}$$

Now how to define a function out of Fin. Such a function will need to inputs: the index $n : \mathbb{N}$ and the element $i : \mathrm{Fin} n$. That is we want to define a function $f : \Pi_{n:\mathbb{N}}\Pi i : \mathrm{Fin}\,n.A\,n\,i$ where $A : \Pi_{n:\mathbb{N}}\mathrm{Fin}\,i \to \mathbf{Type}$. The idea is the same as before: we have to say what f is returning for 0_n and we have to say what it returns for $\mathrm{suc}_n\,i$ assuming we know the result of $f_n\,i$. That is we have to complete

$$\begin{aligned}f_{1+,n}\,0_n &:\equiv ?_0\\ f_{1+n}(\mathrm{suc}_n\,i) &:\equiv ?_1\end{aligned}$$

Before we fill in the ?s - do you note something? The hidden parameter, that is the index $n : \mathbb{N}$ is always $1+n$! That is because the constructors of Fin always produce elements in $\mathrm{Fin}\,(1+n)$ and never any in $\mathrm{Fin}\,0$ - which is only right because $\mathrm{Fin}\,0$ is intentionally left empty. And indeed we are analyzing the constructors of Fin not the indizes.

Now back to the ?s. $?_0 : A_{1+n}\,0_n$ hence we need $z : \Pi_{n:\mathbb{N}}A_{1+n}\,0_n$ and $?_1 : A_{n+1}\,(\mathrm{suc}_n\,i)$ but here we can use the recursive result of $f_n\,i : A_n\,i$. Hence we stipulate a function $s : \Pi_{n:\mathbb{N}}A_n\,i \to A_{1+n}\,(\mathrm{suc}_n\,i)$ and we can fill in

$$\begin{aligned}f_{1+,n}\,0_n &:\equiv z\\ f_{1+n}(\mathrm{suc}_n\,i) &:\equiv s_n\,(f\,n\,i)\end{aligned}$$

Now we package this all into one complicated looking eliminator:

$$\begin{aligned}\mathrm{elim}^{\mathrm{Fin}} : \Pi A : (\Pi_{n:\mathbb{N}}\mathrm{Fin}\,n \to \mathbf{Type}).&(\Pi n : \mathbb{N}.A_{n+1}\,0_n)\\ &\to (\Pi_{n:\mathbb{N}}\Pi i : \mathrm{Fin}\,n.A_n\,i \to A_{1+n}\,(\mathrm{suc}_n\,i))\\ &\to \Pi_{n:\mathbb{N}}\Pi i : \mathrm{Fin}\,n \to A_n\,i\end{aligned}$$

Let us use the eliminator to fullfill an earlier promise, namely to give a justification why

$$\mathrm{nth} : \Pi_{n:\mathbb{N}}\mathrm{Vec}\,A\,n \to \mathrm{Fin}\,n \to A$$

$$\begin{aligned}\mathrm{nth}\,(a :: v)\,0 &:\equiv a\\ \mathrm{nth}\,(a :: v)\,(1+n) &:\equiv \mathrm{nth}\,v\,n\end{aligned}$$

is a reasonable definition, even though it doesn't seem to cover the index zero. To translate this we need head and tail for vectors which are inverting the cons operation for vectors. They are defined as follows:

$$\begin{aligned}&\mathrm{hd} : \Pi_{n:\mathbb{N}}\mathrm{Vec}\,A\,(1+n) \to A\\ &\mathrm{hd}\,(a :: v) :\equiv a\\ &\mathrm{tl} : \Pi_{n:\mathbb{N}}\mathrm{Vec}\,A\,(1+n) \to \mathrm{Vec}\,A\,n\\ &\mathrm{tl}\,(a :: v) :\equiv v\end{aligned}$$

I leave it as an exercise to derive the eliminator for Vec and then show that hd and tl can be defined using it. To make it work we need to commute the arguments, because we will need to make sure that the index of the vector is determined by the index of the element of Fin. That is we define

$$\mathrm{nth}' : \Pi_{n:\mathbb{N}}\mathrm{Fin}\,n \to \mathrm{Vec}\,A\,n \to A$$

This way we can use $\mathrm{Vec}\,A\,n \to A$ depending on n as the motive:

$$\mathrm{nth}' :\equiv \mathrm{elim}^{\mathrm{Fin}}\,(\lambda n.\mathrm{Vec}\,A\,n \to A)\,(\lambda n.\mathrm{hd}_n)\,?_0\,?_1$$

The type of $?_0$ is $\Pi n : \mathbb{N}.\mathrm{Vec}\,A\,(1+n) \to A$, the 1+ here has the origin in the 1+ in the type of 0_n. This fits perfectly with the type of hd which is what we want to do here, hence $?_0 \equiv \lambda n.\mathrm{hd}_n$. In the other case we have

$$?_1 : \Pi_{n:\mathbb{N}}\Pi i : \mathrm{Fin}\,n.(\mathrm{Vec}\,A\,n \to A) \to (\mathrm{Vec}\,A\,(1+n)) \to A)$$

which perfectly fits with calling the recursively computed result with the tail of the input vector:

$$?_1 \equiv \lambda n, i, h.\lambda v.h\,(tl_n v)$$

We notice that we aren't actually using the i parameter to the motive, which means that we don't use the eliminator in its full generality here. Indeed, the result type only depnds on the index not on the element of Fin itself. Putting everything together we can define nth also inlining the parameter swap:

$$\begin{aligned}\mathrm{nth}\,v\,i \quad &:\equiv \mathrm{elim}^{\mathrm{Fin}}\,(\lambda n.\mathrm{Vec}\,A\,n \to A)\\ &\quad (\lambda n.\mathrm{hd}_n)\\ &\quad (\lambda n, i, h.\lambda v.h\,(\mathrm{tl}_n v))\\ &\quad i\,v\end{aligned}$$

OK, this is maybe more principled then the first version of nth but hardly readable. This is a common tension: if we try to justify a construction from first principles we loose readability but on the other hand it is clear that we are not cheating. To navigate in this space we usually present the readable version but when in doubt make sure that it is actually derivable using the spartan world of eliminators. Also implementations of Type Theory should work like this and should translate elegant presentations into type-theoretic assembly language. Alas very few do and it is often considered easier to directly interpret a fairly high level language. This also leads to unintended consequences in form of unsoundness (we can prove False) which diminish the trustworthiness of the system. We will get back to this in the last chapter.

Also you may remark that while I have presented the idea how to derive eliminators from the constructor type, this can hardly be called systematic. What exactly are the restrictions on constructor types (see our discussion in 3.7)? And shouldn't we present the derivation of the eliminator in a more systematic way

There are a number of possible answers here. Both the restrictions and the derivation of the eliminators can be presented in the form of schematic syntax rules. They are precise but do get quite complicated and it is hard to design them so that we call cover all the cases. And if the instances of the eliminators look complicated they are nothing compared to the rules. While there is a point to represent a system based on some precise rules, these rules are hardly readable and they don't give us much insight.

More insight can be obtained by using the language of category theory. As we have seen in the section on simple types data types can be described by functors and the actual type is the initial algebra of a functor. Actually not completely, because we couldn't really capture the η-rules for more interesting datatypes like the natural numbers or lists. This is in a way fixed once we add the dependent eliminator because we can prove the η-rules using the equality type which I will introduce in full gory detail in the next section. What about dependent types like Fin? Here we have to change the category we live in and move from the category of types and functions, to the category of families of types and families of functions. That is types are replaced by families $A : \mathbb{N} \to \mathbf{Type}$ and given $A, B : \mathbb{N} \to \mathbf{Type}$ a family of functions is given by $f : \Pi n : \mathbb{N}.A\,n \to B\,n$. Now Fin can be also specified by a functor that is on type families:

$$F : (\mathbb{N} \to \mathbf{Type}) \to (\mathbb{N} \to \mathbf{Type})$$
$$F\,A :\equiv \lambda n.(\Sigma m : \mathbb{N}.m = 1 + n) + (\Sigma m : \mathbb{N}.m = 1 + n \times A\,m)$$

I leave it as an exercise (which does involve a better understanding of the yet unspecified equality type) to show that this operation comes with a well behaved map operation on family of functions. And indeed, Fin arises as an initial algebra of this functor with the same asymmetry between definitional β-rules and provable η-rules.

The fact that datatypes and even dependent datatypes can be understood as initial algebras is certainly a nice fit between category theory and Type Theory. However, category theory doesn't answer the question what datatypes are acceptable. Already earlier, in section 3.7 I argued that some functors do not correspond to reasonable datatypes, e.g. the functor given by $F : \mathbf{Type} \to$

Type with $F\,X :\equiv (X \to 2) \to 2$ seems problematic. We want to restrict ourselves to strictly positive datatypes but what does this mean exactly?

The answer is that we only want to consider functors which correspond to *containers*, where a container is given by a type of shapes S : **Type** and a family of positions $P : \mathbb{N} \to$ **Type** which gives rise to a functor [5]

$$\begin{aligned} &T : \textbf{Type} \to \textbf{Type} \\ &T\,X :\equiv \Sigma s : S.P\,s \to X \end{aligned}$$

The intuition is that an instance of a container $T\,A$ is given by a choice of shape $s : S$, and an assignment of a *payload*, that is an element of A to each position, i.e. a function $P\,s \to A$. An intuitive example is given by the container representation of the list functor: the shape of a list is its length hence $S :\equiv \mathbb{N}$ and the positions is the finite set with n elements because this is how much payload a list of length n can take, hence $P :\equiv \mathrm{Fin} : S \to$ **Type**. Indeed, we can show that any sytactically strict positive datatype can be represented as a container.

Ever container gives rise to a datatype, that is the type of trees whose nodes are labelled by elements of S and the subtrees of a node labelled by $s : S$ is indexed by $P\,s$. This datatype is called a W-type and given a container that is S : **Type** and $P : S \to$ **Type** and now W : **Type** is generated by the constructor

$$\mathrm{node} : \Pi s : S \to (P\,s \to W) \to W$$

As we have already observed the natural numbers are generated by the functor $T\,X = 1 + X$ which is a container given by $S = 2$ and $P\,false :\equiv 0$ and $P\,true :\equiv 1$. The trees generated by $W\,S\,P$ correspond to the natural numbers where $\mathrm{node\,false} : (0 \to W) \to W$ represents the number 0 all we need to do is to add the function $\mathrm{efq} : 0 \to W$ (which does need no definition) as a parameter $0 \equiv \mathrm{node\,false\,efq}$. Successors are represented using nodes of the other shape $\mathrm{node\,true} : (1 \to W) \to W$ hence we define $\mathrm{suc}\,n :\equiv \mathrm{noe\,true}\,(\lambda x.n)$.

We can do this with other datatypes we have encountered so far, for example lists over A : **Type** are generated by the W-type specified by

$$\begin{aligned} S &:\equiv 1 + A \\ P\,(\mathrm{inl}\,()) &:\equiv 0 \\ P\,(\mathrm{inr}\,a) &:\equiv 1 \end{aligned}$$

To better understand the general idea let's anaylze the expression trees which were given by

$$\begin{aligned} \mathrm{const} &: \mathbb{N} \to \mathrm{Expr} \\ \mathrm{plusOp} &: \mathrm{Expr} \to \mathrm{Expr} \to \mathrm{Expr} \\ \mathrm{timesOp} &: \mathrm{Expr} \to \mathrm{Expr} \to \mathrm{Expr} \end{aligned}$$

[5] As an exercise try to derive the map part of this functor.

Here we hace 3 constructors the first one parametrised by $\mathbb{N}$. Hence we choose

$S :\equiv \mathbb{N} + 2$

How many recursive occurences of Expr can we count for each of the three constructors: none for the first and 2 for the 2nd and the 3rd. Hence we define P as follows:

$$P\,(\mathrm{inl}\,n) :\equiv 0$$
$$P\,(\mathrm{inr}\,b) :\equiv 2$$

We can also represent InfTree which was given by

$$\mathrm{leaf} : \mathrm{InfTree}$$
$$\mathrm{node}(\mathbb{N} \to \mathrm{InfTree}) \to \mathrm{InfTree}$$

here we have a situation where P can be infinite. That is we define

$$S :\equiv 2$$
$$P\,\mathrm{false} :\equiv 0$$
$$P\,\mathrm{true} :\equiv \mathbb{N}$$

What about dependent datatypes like Fin and Vec? With some more work and excessive use of equality they can be done as well. What about coinductive types like streams? It turns out that they can be represented as well by viewing same as a limit of approximations of trees of finite depth. There are some issues here which have to do with the nature of equality which I will cover in the next section.

Ok, we can represent datatypes using W-types. What is the point? The point is that we now only need to specify one datatype namely the W-type. To be precise W is a parametrised type because S and P are really parameters hence

$$W : \Pi S : \mathbf{Type}.(S \to \mathbf{Type}) \to \mathbf{Type}$$

it comes with an eliminator (to make it readable lets fix S and P again) that is $W \equiv W\,S\,P$.

$$\begin{aligned}\mathrm{elim}^W &: \Pi M : W \to \mathbf{Type}.\\ &\quad (\Pi s : S.\Pi f : P\,s \to W.(\Pi p : P\,s.A\,(f\,p) \to W\,(\mathrm{node}\,s\,f)))\\ &\quad \to \Pi w : W.A\,s\end{aligned}$$

$$\mathrm{elim}^W\,M\,m\,(\mathrm{node}\,s\,f) :\equiv m\,s\,f\,(\lambda p.\mathrm{elim}^W\,M\,m\,(f\,p))$$

Ok, this is ugly enough but since we can reduce all other datatypes to this one, this is the only eliminator we have to write down to specify what we mean by dependent primitive recursion for any datatype.

Ok, this is not completely true: we also need to specify eliminators some more basic types: the finite types $0, 1, 2$, Σ-types and the mysterious equality type which I have saved for the next section. This way Type Theory can be turned into a closed system like set theory and everything we need can be derived which maybe quite hard work. Hopefully we can rely on the help of computers here. [6]

4.7 The mystery of equality

I am keeping the equality type to the end because there are some issues with it which is a good preparation for the next chapter. The basic idea is quite easy: for any type $A : \mathbf{Type}$ equality is a dependent type $- =_A - : A \to A \to \mathbf{Type}$ whose only constructor is reflexivity:

$\mathrm{refl} : \Pi_{a:A} A \to A \to \mathbf{Type}$

After having seen many examples of dependent primitive recursion it shouldn't be hard to derive the eliminator for this type. Even though the word recursion is certainly misapplied here because there is no recursion going on. Let's go through the motions: How do we define a function out of an equality type? As before for Fin we also have to abstract over the indices, that is the question is how do we define a function

$f : \Pi_{x,y:A} \Pi p : x = y.M\,x\,y\,p$

where $M : \Pi_{x,y:A} x = y \to \mathbf{Type}$. Since the only constructor is equality we only need to prescribe what is the result for

$f_{x\,x}\,(\mathrm{refl}\,x \equiv ?$

and ? has the type $M\,x\,x\,(\mathrm{refl}\,x)$. That is to define f we need

$m : \Pi x : A.M\,x\,x\,(\mathrm{refl}\,x)$

and given m we define

$f_{x\,x}\,(\mathrm{refl}\,x = m\,x$

We can condense this into one of the horrible eliminators

$$\begin{aligned}\mathrm{elim}^{=} : \Pi M : (\Pi_{x,y:A} x = y \to \mathbf{Type}).\\ (\Pi_{x:A} M_{x\,x}\,\mathrm{refl}_x) \qquad\qquad \to \Pi_{x,y:A}\Pi p : x = y.M\,x\,y\,p\end{aligned}$$

$\mathrm{elim}^{=}\,M\,m\,\mathrm{refl}_x :\equiv m_x$

[6] Just in case my clever colleagues read this and shake their heads: This is not completely true if we want to get the same definitional equalities as we get from the naive definition. I will discuss this particular can of worms in the final chapter.