Why not W?

Jasper Hugunin
Carnegie Mellon University, Pittsburgh PA, USA
jasper@hugunin.net

Abstract

In an extensional setting, `W` types are sufficient to construct a broad class of inductive types, but in intensional type theory the standard construction of even the natural numbers does not satisfy the required induction principle. In this paper, we show how to refine the standard construction of inductive types such that the induction principle is provable and computes as expected in intensional type theory without using function extensionality.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases dependent types, intensional type theory, inductive types, W types

Supplementary Material Tag `v0.1` of `https://github.com/jashug/WhyNotW`

Acknowledgements I want to thank Jon Sterling for his helpful feedback on a draft of this paper.

1 Introduction

In intensional type theory with only type formers `0`, `1`, `2`, Σ, Π, `W`, `Id` and `U`, can the natural numbers be constructed?

The `W` type [9] captures the essence of induction, and in extensional type theory it is straightforward to construct familiar inductive types out of it, including the natural numbers [4]. Taking the elements of the two-element type `2` to be $\hat{0}$ and $\hat{S}$, we define

$$\tilde{\mathbb{N}} = \mathtt{W}_{b:\mathtt{2}}(\mathtt{case}\, b\, \mathtt{of}\, \{\hat{\mathtt{0}} \mapsto \mathtt{0}, \hat{\mathtt{S}} \mapsto \mathtt{1}\}). \tag{1}$$

(the tilde distinguishes the standard construction from our refined construction of the natural numbers in Section 2)

However, as is well known [4, 8, 10, 11], in intensional type theory we cannot prove the induction principle for $\tilde{\mathbb{N}}$ without some form of function extensionality. The obstacle is in the $\hat{0}$ case, where we end up needing to prove $P\ f$ for an arbitrary $f : \mathtt{0} \to \tilde{\mathbb{N}}$, when we only know $P\ (x \mapsto \mathtt{case}\, x\, \mathtt{of}\, \{\})$.

Can this obstacle be avoided? The answer turns out to be yes; in this paper, we show that refining the standard construction allows the natural numbers and many other inductive types to be constructed from `W` in intensional type theory. [1]

Type-theoretic notations and assumptions

We work in a standard intensional type theory with dependent function types $\Pi_{a:A}B[a]$ (also written $\forall_{a:A}B[a]$, $(a : A) \to B[a]$, non-dependent version $A \to B$, constructed as $(x \mapsto y[x])$ or $(\lambda x.\ y[x])$), dependent pair types $\Sigma_{a:A}B[a]$ (also written $(a : A) \times B[a]$, non-dependent version $A \times B$, constructed as (x, y), destructed as $\mathtt{fst}\, p$, $\mathtt{snd}\, p$), finite types `0`, `1` (with inhabitant $\star$), `2` (with inhabitants `ff` and `tt`, aliased to $\hat{0}$ and $\hat{S}$ when we are talking about constructing the natural numbers), `W` types $\mathtt{W}_{a:A}\, B[a]$ (constructor $\mathtt{sup}\, a\, f$ for $a : A$ and $f : B[a] \to \mathtt{W}_a\, B[a]$), identity types $\mathtt{Id}_A\, x\, y$ (constructor `refl`, destruction of $e : \mathtt{Id}\, x\, y$ keeps

[1] These results have been formalized in Coq 8.12 [13]: see the link to supplementary material in the top matter of this article.

x fixed and generalizes over y and e), and a universe $\mathtt{U}$. We define the coproduct $A + B$ as $\sum_{b:2} \mathtt{case}\, b\ \mathtt{of}\ \{\mathtt{ff} \mapsto A, \mathtt{tt} \mapsto B\}$, and notate the injections as $\mathtt{inl}$ and $\mathtt{inr}$.

Function extensionality is the principle that $\forall_x\, \mathtt{Id}\,(f\ x)\ (g\ x)$ implies $\mathtt{Id}\, f\ g$, and uniqueness of identity proofs is the principle that $\mathtt{Id}_{\mathtt{Id}\, x\ y}\, p\ q$ is always inhabited. We do *not* assume either of these principles.

We require strict β-rules for all type formers, and strict η for Σ (that $p = (\mathtt{fst}\, p, \mathtt{snd}\, p)$) and Π (that $f = (x \mapsto f x)$). For convenience we will also assume strict η for $\mathtt{1}$ (that $u = \star$).

2 Constructing $\mathbb{N}$ (for real this time)

We run into problems in the $\hat{\mathtt{0}}$ case because we don't know that $f = (x \mapsto \mathtt{case}\, x\ \mathtt{of}\ \{\})$ for an arbitrary $f : \mathtt{0} \to \tilde{\mathbb{N}}$. To solve those problems, we will assume them away. To construct $\mathbb{N}$, we will first define a predicate $\mathtt{canonical} : \tilde{\mathbb{N}} \to U$ such that $\mathtt{canonical}(\mathtt{sup}\, \hat{\mathtt{0}}\, f)$ implies $\mathtt{Id}\,(x \mapsto \mathtt{case}\, x\ \mathtt{of}\ \{\})\ f$. We then let $\mathbb{N} = \Sigma_{x:\tilde{\mathbb{N}}}\, \mathtt{canonical}\, x$ be the canonical elements of $\tilde{\mathbb{N}}$ (with $\tilde{\mathbb{N}}$ defined by Equation (1)). This predicate will be defined by induction on $\mathtt{W}$, so we can start out with

$$\mathtt{canonical}(\mathtt{sup}\, x\, f) = ? : \mathtt{U} \qquad (x : \mathtt{2},\ f : \cdots \to \tilde{\mathbb{N}}, \text{ may use } \mathtt{canonical}(f\ i) : \mathtt{U}).$$

The obvious next thing to do is to split by cases on $x : \mathtt{2}$:

$$\mathtt{canonical}(\mathtt{sup}\, \hat{\mathtt{0}}\, f) = ? : \mathtt{U} \qquad (f : \mathtt{0} \to \tilde{\mathbb{N}}, \text{ may use } \mathtt{canonical}(f\ i)),$$
$$\mathtt{canonical}(\mathtt{sup}\, \hat{\mathtt{S}}\, f) = ? : \mathtt{U} \qquad (f : \mathtt{1} \to \tilde{\mathbb{N}}, \text{ may use } \mathtt{canonical}(f\ i)).$$

We need canonical terms to be *hereditarily* canonical, that is, we want to include the condition that all sub-terms are canonical. For the $\hat{\mathtt{S}}$ case, thanks to the strict η rules for $\mathtt{1}$ and Π, the types $\mathtt{canonical}(f\ \star)$ and $(i : \mathtt{1}) \to \mathtt{canonical}(f\ i)$ are equivalent; we can use either one. This will be the only condition we need for the $\hat{\mathtt{S}}$ case, so we can complete this part of the definition:

$$\mathtt{canonical}(\mathtt{sup}\, \hat{\mathtt{S}}\, f) = \mathtt{canonical}(f\ \star).$$

The $\hat{\mathtt{0}}$ case is the interesting one. The blind translation of "every sub-term is canonical" is $(i : \mathtt{0}) \to \mathtt{canonical}(f\ i)$, but this leads to the same problem as before: without function extensionality we can't work with functions out of $\mathtt{0}$. Luckily, we have escaped the rigid constraints of the $\mathtt{W}$ type former, and have the freedom to translate the recursive condition as simply $\mathtt{1}$. No sub-terms of zero, no conditions necessary!

$$\mathtt{canonical}(\mathtt{sup}\, \hat{\mathtt{0}}\, f) = ? : \mathtt{U} \qquad (f : \mathtt{0} \to \tilde{\mathbb{N}})$$

That is all well and good, but we can't forget why we are here in the first place: we need $\mathtt{Id}\,(x \mapsto \mathtt{case}\, x\ \mathtt{of}\ \{\})\ f$. Luckily, there is a hole just waiting to be filled:

$$\mathtt{canonical}(\mathtt{sup}\, \hat{\mathtt{0}}\, f) = \mathtt{Id}\,(x \mapsto \mathtt{case}\, x\ \mathtt{of}\ \{\})\ f.$$

Induction

Now we are ready for the finale: induction for $\mathbb{N}$ with the right computational behavior.

Assume we are given a type $P[n]$ which depends on $n : \mathbb{N}$, along with terms $\mathrm{ISO} : P[\mathrm{O}]$ and $\mathrm{ISS} : \forall_{n:\mathbb{N}} P[n] \to P[\mathrm{S}\ \ n]$. Our mission is to define a term $\mathrm{rec}\mathbb{N} : \forall_{n:\mathbb{N}} P[n]$. Happily, the proof goes through if we simply follow our nose.

$$
\begin{aligned}
&\tilde{\mathbb{N}} = \mathtt{W}_{b:\mathtt{2}}(\mathtt{case}\, b \,\mathtt{of}\ \{\hat{\mathtt{0}} \mapsto \mathtt{0}, \hat{\mathtt{S}} \mapsto \mathtt{1}\}) : \mathtt{U}, \\
&\mathtt{canonical} : \tilde{\mathbb{N}} \to \mathtt{U}, \\
&\mathtt{canonical}(\mathtt{sup}\, \hat{\mathtt{0}}\, f) = \mathtt{Id}\,(x \mapsto \mathtt{case}\, x \,\mathtt{of}\ \{\})\ f, && (2)\\
&\mathtt{canonical}(\mathtt{sup}\, \hat{\mathtt{S}}\, f) = \mathtt{canonical}(f\ \star), \\
&\mathbb{N} = \Sigma_{x:\tilde{\mathbb{N}}}\, \mathtt{canonical}\, x : \mathtt{U}, && (3)\\
&\mathrm{O} = (\mathtt{sup}\, \hat{\mathtt{0}} (x \mapsto \mathtt{case}\, x \,\mathtt{of}\ \{\}), \mathtt{refl}) : \mathbb{N}, && (4)\\
&\mathrm{S} = n \mapsto (\mathtt{sup}\, \hat{\mathtt{S}} (\star \mapsto \mathtt{fst}\, n), \mathtt{snd}\, n) : \mathbb{N} \to \mathbb{N}. && (5)
\end{aligned}
$$

Figure 1 The complete definition of $\mathbb{N}$.

We begin by performing induction on $\mathtt{fst}\, n : \tilde{\mathbb{N}}$, and then case on $\hat{\mathtt{0}}$ vs $\hat{\mathtt{S}}$, just like the definition of $\mathtt{canonical}$.

$$\mathrm{rec}\mathbb{N}(\mathtt{sup}\, \hat{\mathtt{0}}\, f, y) = ? : P[(\mathtt{sup}\, \hat{\mathtt{0}}\, f, y)] \qquad (f : \mathtt{0} \to \tilde{\mathbb{N}},\ y : \mathtt{Id}\,(x \mapsto \mathtt{case}\, x \,\mathtt{of}\ \{\})\ f),$$

$$\mathrm{rec}\mathbb{N}(\mathtt{sup}\, \hat{\mathtt{S}}\, f, y) = ? : P[(\mathtt{sup}\, \hat{\mathtt{S}}\, f, y)] \qquad (f : \mathtt{1} \to \tilde{\mathbb{N}},\ y : \mathtt{canonical}(f\ \star)).$$

(where we may make recursive calls $\mathrm{rec}\mathbb{N}(f\ i, y')$ for any i and y')

In the $\hat{\mathtt{S}}$ case, $f = (\star \mapsto f\ \star)$ by the η rules for $\mathtt{1}$ and Π, and thus $(\mathtt{sup}\, \hat{\mathtt{S}}\, f, y) = \mathrm{S}\ (f\ \star, y)$. We can thus define

$$\mathrm{rec}\mathbb{N}(\mathtt{sup}\, \hat{\mathtt{S}}\, f, y) = \mathrm{ISS}\ (f\ \star, y)\ (\mathrm{rec}\mathbb{N}(f\ \star, y)).$$

The $\hat{\mathtt{0}}$ case is again the interesting one, but it is only a little tricky. We know $\mathrm{ISO} : P[(\mathtt{sup}\, \hat{\mathtt{0}}\, (x \mapsto \mathtt{case}\, x \,\mathtt{of}\ \{\}), \mathtt{refl})]$, and we want $P[(\mathtt{sup}\, \hat{\mathtt{0}}\, f, y)]$. But since we have $y : \mathtt{Id}\,(x \mapsto \mathtt{case}\, x \,\mathtt{of}\ \{\})\ f$, this is a direct application of the eliminator for $\mathtt{Id}$. We thus complete the definition of $\mathrm{rec}\mathbb{N}$ with

$$\mathrm{rec}\mathbb{N}(\mathtt{sup}\, \hat{\mathtt{0}}\, f, y) = \mathtt{case}\, y \,\mathtt{of}\ \{\mathtt{refl} \mapsto \mathrm{ISO}\}.$$

Examining the definitions, we can see that as long as we have strict η for Σ and strict β for $\mathtt{Id}$, $\mathrm{rec}\mathbb{N}\, \mathrm{O} = \mathrm{ISO}$ and $\mathrm{rec}\mathbb{N}(\mathrm{S}\, n) = \mathrm{ISS}\ n\ (\mathrm{rec}\mathbb{N}\ \ n)$. Thus we have indeed defined the natural numbers with the expected induction principle and computational behavior in terms of the $\mathtt{W}$ type.

▶ **Theorem 1.** *The natural numbers can be constructed in intensional type theory with only type formers* $\mathtt{0}$, $\mathtt{1}$, $\mathtt{2}$, Σ, Π, $\mathtt{W}$, $\mathtt{Id}$ *and* $\mathtt{U}$, *such that the induction principle has the expected computational behavior.*

3 The General Case

Above, we have refuted a widely held intuition about the expressiveness of intensional type theory with $\mathtt{W}$ as the only primitive inductive type. Once we know we can construct the natural numbers, that we can construct lots of other inductive types is much less surprising.

Nevertheless, for completeness we define below an internal type of codes for inductive types along with the construction from $\mathtt{W}$ types of the interpretation of those codes. For convenience, in this section we assume that we have not just one universe $\mathtt{U}$ but an infinite cumulative tower of universes $\mathtt{U}_0 : \mathtt{U}_1 : \cdots : \mathtt{U}_i : \mathtt{U}_{i+1} : \ldots$ all closed under $\mathtt{0}$, $\mathtt{1}$, $\mathtt{2}$, Σ, Π, $\mathtt{W}$, and $\mathtt{Id}$ such that $A : \mathtt{U}_i$ implies $A : \mathtt{U}_{i+1}$.

The end result is a universe of inductive types which is self-describing, or "levitating" in the sense of [2].

Given

$$\text{a type } P[n] \text{ depending on } n : \mathbb{N}, \tag{6}$$
$$\mathrm{ISO} : P[\mathrm{O}], \tag{7}$$
$$\mathrm{ISS} : \forall_{n:\mathbb{N}} P[n] \to P[\mathrm{S}\ n], \tag{8}$$

we have

$$\mathrm{rec}\mathbb{N} : \forall_{n:\mathbb{N}} P[n],$$
$$\mathrm{rec}\mathbb{N}(\mathtt{sup}\,\hat{\mathtt{0}}\,f, y) = \mathtt{case}\,y\,\mathtt{of}\ \{\mathtt{refl} \mapsto \mathrm{ISO}\}, \tag{9}$$
$$\mathrm{rec}\mathbb{N}(\mathtt{sup}\,\hat{\mathtt{S}}\,f, y) = \mathrm{ISS}\ (f\ \star, y)\ (\mathrm{rec}\mathbb{N}(f\ \star, y)),$$
$$\mathrm{rec}\mathbb{N}\,\mathrm{O} = \mathrm{ISO}, \tag{10}$$
$$\mathrm{rec}\mathbb{N}(\mathrm{S}\,n) = \mathrm{ISS}\ n\ (\mathrm{rec}\mathbb{N}\ \ n). \tag{11}$$

Figure 2 Induction for $\mathbb{N}$.

3.1 Inductive Codes

We will let $\mathrm{Code}_i : \mathtt{U}_{i+1}$ be the type of codes for inductive types in $\mathtt{U}_i$, and implement it for now as a primitive inductive type. In Section 3.4 we will show how to construct Code itself from $\mathtt{W}$.

To define Code, we adapt the axiomatization of induction-recursion from [5]. Thus Code_i is generated by the constructors

$$\mathrm{nil} : \mathrm{Code}_i, \quad \mathrm{nonind} : (A : \mathtt{U}_i) \to (A \to \mathrm{Code}_i) \to \mathrm{Code}_i, \quad \mathrm{ind} : \mathtt{U}_i \to \mathrm{Code}_i \to \mathrm{Code}_i.$$

Looking at $\mathtt{U}_i$ as the usual category of types and functions, a code $A : \mathrm{Code}_i$ defines an endofunctor $F_A : \mathtt{U}_i \to \mathtt{U}_i$ defined by recursion on A by

$$F_{\mathrm{nil}}\ X = 1, \tag{12}$$
$$F_{\mathrm{nonind}(A,B)}\ X = \Sigma_{a:A} F_{(B\ a)}\ X, \tag{13}$$
$$F_{\mathrm{ind}(\mathrm{Ix},B)}\ X = (\mathrm{Ix} \to X) \times F_B\ X. \tag{14}$$

▶ **Example 2.** We can define a code for the natural numbers as

$$\text{“}\mathbb{N}\text{”} = \mathrm{nonind}(2, b \mapsto \mathtt{case}\,b\,\mathtt{of}\ \{\hat{\mathtt{0}} \mapsto \mathrm{nil}, \hat{\mathtt{S}} \mapsto \mathrm{ind}(1, \mathrm{nil})\}) : \mathrm{Code}_0.$$

Each code also defines a polynomial functor $G_A\ X = \Sigma_{s:S_A}(P_A\ s \to X)$, which is what is used in the standard construction:

$$S_{\mathrm{nil}} = 1 \qquad P_{\mathrm{nil}}\ \star = 0 \tag{15}$$
$$S_{\mathrm{nonind}(A,B)} = \Sigma_{a:A} S_{(B\ a)} \qquad P_{\mathrm{nonind}(A,B)}\ (a,b) = P_{(B\ a)}\ b \tag{16}$$
$$S_{\mathrm{ind}(\mathrm{Ix},B)} = S_B \qquad P_{\mathrm{ind}(\mathrm{Ix},B)}\ b = \mathrm{Ix} + P_B\ b. \tag{17}$$
$$\tilde{\mathrm{El}}\,A = \mathtt{W}_{s:S_A}\ P_A. \tag{18}$$

There is an easy-to-define natural transformation $\epsilon : F \Rightarrow G$, and it even has a left inverse on objects, but without function extensionality ϵ does not have a right inverse (roughly speaking, ϵ is not surjective); there are usually terms $g : G\ X$ not in the image of ϵ. This is exactly the problem we ran into in the case of the natural numbers: the map $(\star \mapsto (x \mapsto \mathtt{case}\,x\,\mathtt{of}\ \{\})) : 1 \to (0 \to X)$ is not surjective.

The last component we need is $\mathrm{All}_A\ s : (Q : P_A\ s \to \mathtt{U}_j) \to \mathtt{U}_j$ (for universe level $j \geq i$), the quantifier "holds at every position" (a refinement of $\forall_p, Q\ p$):

$$\mathrm{All}_{\mathrm{nil}} \star Q = \mathtt{1}, \tag{19}$$

$$\mathrm{All}_{\mathrm{nonind}(A,B)} (a, b)\ Q = \mathrm{All}_{(B\ a)}\, b\, Q, \tag{20}$$

$$\mathrm{All}_{\mathrm{ind}(\mathrm{Ix},B)}\, b\, Q = (\forall_i, Q\ (\mathtt{inl}\, i)) \times \mathrm{All}_B\, b\, (Q \circ \mathtt{inr}). \tag{21}$$

Noting that $\mathtt{snd}(\epsilon\ t) : P\ (\mathtt{fst}(\epsilon\ t)) \to X$ enumerates the sub-terms of $t : F\ X$, $\mathrm{All}(Q \circ \mathtt{snd}(\epsilon\ t))$ lets us lift a predicate $Q : X \to \mathtt{U}_j$ to a predicate over $t : F\ X$.

▶ **Lemma 3.** *There is an equivalence r (à la Voevodsky, a function with contractible fibers)*

$$r : F\ (\Sigma_{x:X} C\ x) \simeq \Sigma_{(t:F\ X)}\, \mathrm{All}(C \circ \mathtt{snd}(\epsilon\ t)). \tag{22}$$

Proof. Follows easily by induction on the code A. ◀

3.2 The General Construction

We are finally ready to define the true construction of inductive types $\mathrm{El} : \mathrm{Code} \to \mathtt{U}_i$. As with natural numbers, we define a "canonicity" predicate on $\tilde{\mathrm{El}}\, A$, which says that "all subterms are canonical, and this node is in the image of ϵ". This translates as:

$$\mathtt{canonical}(\mathtt{sup}\, s\, f) = \mathrm{All}(\mathtt{canonical} \circ f) \times (t : F\ (\tilde{\mathrm{El}}\, A)) \times \mathtt{Id}\, (\epsilon\ t)\ (s, f) : \mathtt{U}_i, \tag{23}$$

and thus we finally have

$$\mathrm{El}\ \ A = \Sigma_{x:\tilde{\mathrm{El}}\, A}\, \mathtt{canonical}\, x. \tag{24}$$

For the constructors, we expect to have $\mathrm{intro} : F\ (\mathrm{El}\, A) \to \mathrm{El}\, A$, which we define by

$$\mathrm{intro}\, x = (\mathtt{sup}\ (\epsilon\ (\mathtt{fst}\ (r\ x))), (\mathtt{snd}(r\ x), \mathtt{fst}\ (r\ x), \mathtt{refl})). \tag{25}$$

using the equivalence r from Lemma 3 to split $x : F\ (\mathrm{El}\, A)$ into $\mathtt{fst}(r\ x) : F\ (\tilde{\mathrm{El}}\, A)$ and $\mathtt{snd}(r\ x) : \mathrm{All}(\mathtt{canonical} \circ \mathtt{snd}(\epsilon\ \mathtt{fst}\ (r\ x)))$.

3.3 General Induction

When we go to define the induction principle for $\mathrm{El}\, A$, we are given $P : \mathrm{El}\, A \to \mathtt{U}_j$ for some $j \geq i$ and the induction step $\mathrm{IS} : \forall_{(x:F\ (\mathrm{El}\, A))}\, \mathrm{All}(P \circ \mathtt{snd}(\epsilon\ x)) \to P\ (\mathrm{intro}\, x)$, and want to define $\mathrm{rec} : \forall_{(x:\mathrm{El}\, A)} P\ x$. The definition proceeds by induction on $\mathtt{fst}\, x$:

$$\mathrm{rec}(\mathtt{sup}\, s f, (h, t, e)) = ? : P\ (\mathtt{sup}\, s f, (h, t, e)) \qquad h : \mathrm{All}(\mathtt{canonical} \circ f) \quad e : \mathtt{Id}\, (\epsilon\ t)\ (s, f),$$

and we have induction hypothesis $H = p \mapsto c \mapsto \mathrm{rec}(f\ p, c) : \Pi_p \Pi_c P\ (f\ p, c)$. Next, we destruct the identity proof e, generalizing over both h and H, leaving us with

$$\mathrm{rec}(\mathtt{sup}(\epsilon\ t), (h, t, \mathtt{refl})) = ? : P\ (\mathtt{sup}(\epsilon\ t), (h, t, \mathtt{refl})),$$

for $t : F\ (\tilde{\mathrm{El}}\, A)$, $h : \mathrm{All}(\mathtt{canonical} \circ \mathtt{snd}(\epsilon\ t))$, and $H : \Pi_p \Pi_c P\ (\mathtt{snd}(\epsilon\ t)\ p, c)$. The last step to bring us in line with the definition of intro is to use the equivalence from Lemma 3 to replace (t, h) with $r\ x$ for some $x : F\ (\mathrm{El}\, A)$, leaving us with

$$\mathrm{rec}(\mathtt{sup}(\epsilon\ (\mathtt{fst}(r\ x))), (\mathtt{snd}(r\ x), \mathtt{fst}(r\ x), \mathtt{refl})) = ? : P\ (\mathrm{intro}\, x)$$

and induction hypothesis $H : \Pi_p \Pi_c P\ (\texttt{snd}(\epsilon\ (\texttt{fst}(r\ x)))\ p, c)$. We can then apply IS, but that leaves us with an obligation to prove $\mathrm{All}(P \circ \texttt{snd}(\epsilon\ x))$. Fortunately, it is easy to show by induction on the code A that our hypothesis H is sufficient to dispatch this obligation.

This completes the definition of the induction principle, and it can be observed on concrete examples like the natural numbers to have the expected computational behavior. We can also prove a propositional equality $\texttt{Id}\,(\mathrm{rec}(\mathrm{intro}\,x))\ (\mathrm{IS}\ x\ (\mathrm{rec} \circ \texttt{snd}(\epsilon\ x)))$ witnessing the expected computation rule, and observe on concrete examples that this witness computes to reflexivity. The details of this construction have all been formalized in Coq.

3.4 Bootstrapping

In Section 3.1 we postulated the type Code_i to be a primitive inductive type, which leads to the question of whether the general construction we have proposed is *really* constructing inductive types out of `W` or whether it is making sneaky use of the inductive structure of Code_i to perform the construction.

As a first observation, $\mathrm{Code}_i : \texttt{U}_{i+1}$ while $\mathrm{El} : \mathrm{Code}_i \to \texttt{U}_i$, thus Code_i can't appear as data in $\mathrm{El}\,A$: it is too big! However, this argument doesn't show that we can completely eliminate Code_i from the construction.

Next, we observe that the inductive type Code_i itself has a code "Code_i" : Code_{i+1}:

$$
\begin{aligned}
\text{“Code}_i\text{”} = \ & \mathrm{nonind}((1 + \texttt{U}_i) + \texttt{U}_i, t \mapsto \texttt{case}\,t\ \texttt{of}\ \{ \\
& \quad \texttt{inl}(\texttt{inl}\,\star) \mapsto \mathrm{nil}, && \text{(case nil)} \\
& \quad \texttt{inl}(\texttt{inr}\,A) \mapsto \mathrm{ind}(A, \mathrm{nil}), && \text{(case nonind)} \\
& \quad \texttt{inr}\,\mathrm{Ix} \mapsto \mathrm{ind}(1, \mathrm{nil}), && \text{(case ind)} \\
& \}).
\end{aligned}
$$

Then we can propose to define $\mathrm{Code}_i = \mathrm{El}$ "Code_i", but this is a circular definition: we define Code_i by using recursion on Code_{i+1}. What we really want, and in some ways should be able to expect, is that El "Code_i" *computes* to a normal form which no longer mentions Code but is expressed purely in terms of `W`. We could then tie the knot by defining Code_i to be what El "Code_i" *will compute to*, once we have defined El.

There is just one problem to resolve, which is that currently, El, which is defined by recursion on codes, gets stuck on $\mathrm{El}(\texttt{case}\,t\ \texttt{of}\ \{\ldots\})$ which is used to branch on constructor tags; we are missing some sort of commuting conversion [7, section 10]. Fortunately, we can do without it, by reifying branching on constructor tags as part of Code. We add another constructor

$$
\mathrm{choice} : \mathrm{Code}_i \to \mathrm{Code}_i \to \mathrm{Code}_i, \qquad F_{\mathrm{choice}(A,B)}\ X = F_A\ X + F_B\ X \tag{26}
$$

which encodes the simple binary sum of functors, specializing the dependent sum of functors $\mathrm{nonind}(2, b \mapsto \texttt{case}\,b\ \texttt{of}\ \{\ldots\})$ (but with all proofs essentially the same). With this in hand, we can define

$$
\begin{aligned}
\text{“Code}_i\text{”} = \ & \mathrm{choice}(\mathrm{choice}(\\
& \quad \mathrm{nil}, && \text{(case nil)} \\
& \quad \mathrm{choice}(\\
& \quad\quad \mathrm{nonind}(\texttt{U}_i, A \mapsto \mathrm{ind}(A, \mathrm{nil})), && \text{(case nonind)} \\
& \quad\quad \mathrm{ind}(1, \mathrm{ind}(1, \mathrm{nil})))), && \text{(case choice)} \\
& \quad \mathrm{nonind}(\texttt{U}_i, \mathrm{Ix} \mapsto \mathrm{ind}(1, \mathrm{nil}))). && \text{(case ind)}
\end{aligned} \tag{27}
$$

With this adjustment, the structure of the code is not hidden inside `case`, and the computation of El "Code$_i$" proceeds to completion without becoming stuck, resulting in a term which does not mention Code at all. From there, we can define El such that El "Code$_i$" $=$ Code$_i$, as in [2] but with no invisible cables, just the `W` type.

▶ **Theorem 4.** *In intensional type theory with type formers* `0`, `1`, `2`, Σ, Π, `W`, `Id` *and an infinite tower of universes* $\mathtt{U}_i$*, there exist terms* $\mathrm{Code}_i : \mathtt{U}_{i+1}$ *such that* Code_i *is a type of codes for inductive types, constructed by* $\mathrm{El} : \mathrm{Code}_i \to \mathtt{U}_i$*, and terms "*$\mathrm{Code}_i$*"*$: \mathrm{Code}_{i+1}$ *such that* El *"*Code_i*"* $= \mathrm{Code}_i$.

4 Discussion

4.1 Composition

Being codes for functors, one may ask if Code$_i$ is closed under composition of functors? As with the codes for inductive-recursive types we have modified, without function extensionality we do not appear to have composition (for similar reasons as considered in [6]). Indeed, experiments suggest that the general construction of a class of inductive types closed under composition of the underlying functors essentially requires function extensionality. Even worse, to get definitional computation rules for the resulting inductive types, all our attempts have required that transporting over $\mathtt{funext}(x \mapsto \mathtt{refl})$ computes to the identity, a property which not even cubical type theory [3] satisfies (it is satisfied, however, by observational type theory [1]). Thus, we do not know how to combine a class of inductive types closed under composition constructed from the `W` type as we have in Section 3 with the the principle of Univalence [12] while maintaining good computational behavior.

We do however wish to emphasize that the construction in Section 3 (which is not closed under composition) is completely compatible with Univalence, and could be implemented in cubical type theory as long as an identity type with strict β rule is used.

4.2 Canonicity

Despite being constructed from W types, our natural numbers enjoy the canonicity property (that for every closed term n of type $\mathbb{N}$, either $n = \mathrm{O}$ or $n = \mathrm{S}\, m$ for some closed $m : \mathbb{N}$), at least as long as `2` and `Id` enjoy canonicity (closed $b : \mathtt{2}$ implies $b = \hat{\mathtt{0}}$ or $b = \hat{\mathtt{S}}$, and closed $e : \mathtt{Id}\, x\, y$ implies $e = \mathtt{refl}$ and $x = y$). The trick is that when we have some representation of zero, it looks like $(\mathtt{sup}\, \hat{\mathtt{0}}\, f, e)$, where e is a closed term of type $\mathtt{Id}\,(x \mapsto \mathtt{case}\, x\, \mathtt{of}\, \{\})\ f$, and thus by canonicity for `Id`, this must be $(\mathtt{sup}\, \hat{\mathtt{0}}(x \mapsto \mathtt{case}\, x\, \mathtt{of}\, \{\}), \mathtt{refl}) = \mathrm{O}$.

However, in a situation like cubical type theory where function extensionality holds, `Id` no longer enjoys canonicity, and neither does our construction of the natural numbers.

4.3 Problems

What are the problems with using this construction as the foundation for inductive types in a proof assistant? While we have shown bare possibility, this is not an obviously superior solution when compared to the inductive schemes present in proof assistants today.

The construction is complex, which has the possibility of confusing unification and other elaboration algorithms. While the reduction behavior simulates the expected such, the reduction engine has to make many steps to simulate one step of a primitive inductive type, which can lead to a large slowdown. As an example, we observed the general construction slow down from seconds to check to half an hour when replacing primitive inductive types the bootstrapped definition of Code. Understanding exactly why this slowdown happens and how to alleviate it is an important question to be answered before attempting to apply this construction in practice.

This construction is also limited in it's expressivity. Nested inductive types such as `Inductive tree := node : list tree → tree` do not appear to be constructible, nor do mutual inductive types landing in a mixture of impredicative and predicative sorts at different levels, and nor do inductive-inductive types.

4.4 Conclusion

We have shown that intensional type theory with `W` and `Id` types is more expressive than was previously believed. It supports not only the natural numbers, but a whole host of inductive types, generated by an internal type of codes, which is itself an inductive type coded for by itself (one universe level up). This brings possibilities for writing generic programs acting on inductive types internally, and perhaps simplifies the general study of extensions of intensional type theory: once you know `W` works, you know lots of inductive types work (with a few side conditions to check).

Thus we return to the titular question: why not use `W` as the foundation of induction in intensional type theory? Equipped with this result, one can no longer say that it is impossible.

References

1 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68. ACM, 2007. doi:10.1145/1292597.1292608.

2 James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14. ACM, 2010. doi:10.1145/1863543.1863547.

3 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8475, doi:10.4230/LIPIcs.TYPES.2015.5.

4 Peter Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, 176(1):329 – 335, 1997. doi:https://doi.org/10.1016/S0304-3975(96)00145-4.

5 Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999. doi:10.1007/3-540-48959-2_11.

6 Neil Ghani, Conor McBride, Fredrik Nordvall Forsberg, and Stephan Spahn. Variations on inductive-recursive definitions. In Kim G. Larsen, Hans L. Bodlaender, and Jean-François Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, August 21-25, 2017 - Aalborg, Denmark*, volume 83 of *LIPIcs*, pages 63:1–63:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.MFCS.2017.63.

7 Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1990. Translated and with appendices by Paul Taylor and Yves Lafont. URL: `http://www.paultaylor.eu/stable/prot.pdf`.

8 Healfdene Goguen and Zhaohui Luo. Inductive data types: Well-ordering types revisited. *Logical Environments*, 1992. URL: `https://www.cs.rhul.ac.uk/home/zhaohui/WTYPES93.pdf`.

9 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by G. Sambin of a series of lectures given in Padua, 1980.

10 Conor McBride. W-types: good news and bad news, Mar 2010. URL: `https://mazzo.li/epilogue/index.html%3Fp=324.html`.

11 Bengt Nordsrtöm, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

12 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: `https://homotopytypetheory.org/book/`.

13 The Coq Development Team. The Coq proof assistant, version 8.12.0, July 2020. `doi:10.5281/zenodo.4021912`.