W-Types and Initial Algebras via Examples

Joseph Hua

March 3, 2022

1 Introduction

W-types offer a compact method of presenting tree-like inductive types. We introduce this via the two standard examples: the naturals and lists. The idea is to break down an inductive type into two pieces of data, its constructors and their arities, so that each way of making something in the inductive type looks like `the_type`^{n} → `the_type`.

Types such as binary sums and products have simple categorical descriptions via their universal properties. The benefit of viewing inductive types as W-types is that it offers a categorical description of the type as an initial algebra of some endofunctor - for example allowing us to generalise the notion of the naturals to having a natural numbers object or a list object in a category (with certain (co)limits). There are also practical applications; W-type constructions should be useful for computing the cardinalities of inductively defined types. In particular I should be able to compute the cardinality of the type of `terms` in a model-theoretic language by constructing a corresponding W-type, and Chris Hughes has used them for computing the cardinalities of polynomial rings.

My work is across 3 files, though only two of them should be considered original work, as `endofunctor.algebra` is mostly a refactoring of `category_theory/monad/algebra` in the library (except where I show that initial algebras are preserved by their endofunctors). The two examples of W-types make up the file `constructions` and everything else (things about polynomial endofunctors for W-types) is in `endofunctor.W`.

2 W-types as inductive types

We adopt the notational convention from mathlib, where we use α to denote the type indexing the constructors, and β to denote the type for the arities. Given such a pair, one gets a W-type by the following definition:

```
inductive W_type {α : Type*} (β : α → Type*)
| mk (a : α) (f : β a → W_type) : W_type
```

This says, to make a term in the W-type, it suffices to give a constructor `a` and a map `f` from the arity to the type. Since the arity will in our cases be small finite types, the map amounts to picking a finite number of terms in the type.

We break down the naturals into its constructors and their arities. The naturals have two constructors `nat.zero` and `nat.succ`, where `nat.zero` has empty arity (nullary) $\mathbb{N}^0 \to \mathbb{N}$ and `nat.succ` has unit arity (unary) $\mathbb{N}^1 \to \mathbb{N}$. This is because to come up with a natural, we either give the constructor `nat.zero` and nothing else, or give `nat.succ` and a natural n.

```
/-- The constructors for the naturals -/
inductive nat_α : Type
| zero : nat_α
| succ : nat_α

/-- The arity of the constructors for the naturals, ‘zero‘ takes no arguments, ‘succ‘ takes one -/
def nat_β : nat_α → Type
| nat_α.zero := empty
| nat_α.succ := unit
```

Now, `W_type nat_β` should look exactly like $\mathbb{N}$. To make `nat.zero` we pick the constructor `nat_α.zero` and provide nothing (in the form of the unique map out of the empty type). To make `nat.succ n` we pick the constructor `nat_α.succ` and provide the previous term from the W-type corresponding to n.

```
/-- The isomorphism from the naturals to its corresponding ‘W_type‘ -/
def nat_to : ℕ → W_type nat_β
| nat.zero := ⟨ nat_α.zero , empty.elim ⟩
| (nat.succ n) := ⟨ nat_α.succ , λ _ , nat_to n ⟩

/-- The isomorphism from the ‘W_type‘ of the naturals to the naturals -/
def to_nat : W_type nat_β → ℕ
| (W_type.mk nat_α.zero f) := 0
| (W_type.mk nat_α.succ f) := (to_nat (f ())).succ
```

It follows that these give an equivalence between the types. Lists are viewed slightly differently to usual with `list γ` having `1 + γ` many constructors instead of just two: `list.nil` has empty arity $(\texttt{list}\ \gamma)^0 \to$ `list γ` and for each $x : \gamma$ `list.cons a` is a constructor with unit arity $(\texttt{list}\ \gamma)^1 \to$ `list γ`. To make a list we either give the constructor `list.nil` and nothing else, or give `list.cons a` with a list `l`.

```
/-- The constructors for lists -/
inductive list_α : Type u
| nil : list_α
| cons : γ → list_α

/-- The arities of each constructor for lists, ‘nil‘ takes no arguments, ‘cons hd‘ takes one -/
def list_β : list_α γ → Type u
| list_α.nil := pempty
| (list_α.cons hd) := punit
```

Then we can see that to construct a list, the empty list corresponds to giving the constructor `list_α.nil` and nothing else, whilst appending a new value at the head of the list corresponds to giving the constructor `list_α.cons hd` and the list corresponding to the previous list.

```
/-- The isomorphism from lists to W_types -/
@[simp] def list_to_W_type : list γ → W_type (list_β γ)
| list.nil := ⟨ list_α.nil, pempty.elim ⟩
| (list.cons hd tl) := ⟨ list_α.cons hd, λ _ , list_to_W_type tl ⟩

/-- The isomorphism from W_types to lists -/
@[simp] def W_type_to_list : W_type (list_β γ) → list γ
| (W_type.mk list_α.nil f) := []
| (W_type.mk (list_α.cons hd) f) := hd :: W_type_to_list (f punit.star)
```

The equivalence between `W_type (list_β γ)` and `list γ` follows from these definitions.

W-types as described above don't cover everything that one can make using `inductive` in lean. Inductive families such as `dvector` which takes a natural to lists of length n in a type α cannot be described exactly. (The following definitions are both from the flypitch project.)

```
inductive dvector (α : Type u) : ℕ → Type u
| nil : dvector 0
| cons : ∀{n} (x : α) (xs : dvector n), dvector (n+1)
```

Another example would be

```
/-- Preterms are used in model theory to define terms in a language -/
inductive preterm : ℕ → Type u
| var : ∀ (k : ℕ), preterm 0
| func : ∀ {l : ℕ} (f : func_symb l), preterm l
| app : ∀ {l : ℕ} (t : preterm (l + 1)) (s : preterm 0), preterm l
```

The above are both problematic because the types of the constructors depend on some $n : \mathbb{N}$, so that they're not of the form `the_type`^{n} → `the_type` like before. There are then *indexed W-types* designed for these purposes, and reducing indexed W-types to standard W-types is an active area of research. See here.

3 Natural numbers objects

The categorical way of viewing the naturals is as a natural number object in the category `Type`. A natural number object is defined to be the initial algebra of the polynomial endofunctor $1 + X$:

$$\begin{aligned} 1 + X : \mathsf{Type} &\to \mathsf{Type} \\ \alpha &\mapsto \mathsf{punit} \sqcup \alpha \end{aligned}$$

Then an algebra of this endofunctor is an object `c : Type` with a 'structure map' `str : 1 + c → c`.

```
/-- An algebra of an endofunctor; `str` stands for "structure morphism" -/
structure algebra (F : C ⥤ C) :=
(A : C)
(str : F.obj A ⟶ A)
```

The naturals are an algebra of $1 + X$, with the map from $1 + \mathbb{N}$ mapping to `nat.zero` on the left and taking `n` to `nat.succ n` on the right. The data of an algebra of $1 + X$ can be represented using the following diagram.

$$\begin{tikzcd} 1 \arrow[r, "z"] & c \arrow[r, "s"] & c \end{tikzcd}$$

The algebras of an endofunctor form a category, with morphisms in between them maps between the underlying objects that commute with the structure map. [1]

```
/-- A morphism between algebras of endofunctor `F` -/
@[ext] structure hom (A₀ A₁ : algebra F) :=
(f : A₀.1 ⟶ A₁.1)
(h' : F.map f ≫ A₁.str = A₀.str ≫ f . obviously)
```

$$\begin{tikzcd} 1 \arrow[r, "\mathsf{zero}"] \arrow[d, equal] & \mathbb{N} \arrow[r, "\mathsf{succ}"] \arrow[d, dashed] & \mathbb{N} \arrow[d, dashed] \\ 1 \arrow[r, "z"] & c \arrow[r, "s"] & c \end{tikzcd}$$

In this project we will *not* define the general notion polynomial endofunctors on a category with pullbacks, but rather treat a polynomial endofunctor as any functor (isomorphic to one) of the form

$$\Sigma_{a:A} X^{Ba} : \mathsf{Type} \to \mathsf{Type}$$

Then for each W-type we construct such a polynomial endofunctor and show that the W-type is the initial algebra of the endofunctor, meaning it is the initial object in the category of algebras.

[1] The `ext` attribute provides the tactic `ext` with a lemma for showing when two such morphisms are equal. Here it reduces equality to equality of the underlying morphisms in the category `C`.

4 Inital algebras as objects preserved by their endofunctors

Before we prove the main results we study an important property of initial algebras. An intuitive view of initial algebras offered by Bartosz Milewski is that they are the (smallest) objects that are preserved when one recursively applies the associated polynomial endofunctor. Specialising to W-types (which we will show to be initial algebras), this is viewing the inductive type as being freely generated by its constructors. For example this says that "the only way to make a natural is by using `zero` or `succ`".

Note that the initial algebra is not in general the only object preserved by the endofunctor. For example $1+X$ preserves the type $\mathbb{R}$ (types in bijection are equivalent), which certainly is not isomorphic to $\mathbb{N}$ as a type, since they do not biject. The proof that initial algebras are preserved comes from considering this diagram:

$$\begin{tikzcd}
FA \arrow[r, dashed, "Fg"] \arrow[d, "\mathsf{str}"] & FFA \arrow[r, "F\mathsf{str}"] \arrow[d, "F\mathsf{str}"] & FA \arrow[d, "\mathsf{str}"] \\
A \arrow[r, dashed, "g"'] & FA \arrow[r, "\mathsf{str}"'] & A
\end{tikzcd}$$

Naturally, it is the structure map that is the isomorphism from the image FA to the initial algebra A. The inverse of the structure map is g, which comes from the universal property of the initial algebra A, noting that FA is another algebra of F. Applying uniqueness from the universal property of A (`is_initial.hom_ext`) we see that the composition along the bottom is the identity. The inverse g is given by `str_inv`

```
variables {A} (h : limits.is_initial A)

/-- The inverse of the structure map of an initial algebra -/
def str_inv : A.1 ⟶ F.obj A.1 := (h.to ⟨ F.obj A.1 , F.map A.str ⟩).1

/- The composition along the bottom row is the identity (as an algebra morphism) -/
lemma left_inv' : (⟨str_inv h ≫ A.str⟩ : A ⟶ A) = 𝟙 A :=
limits.is_initial.hom_ext h _ (𝟙 A)
```

We can forget this into the underlying category `Type`.

```
lemma left_inv : str_inv h ≫ A.str = 𝟙 _ := congr_arg hom.f (left_inv' h)
```

Since the above diagram commutes, we see that the other direction follows

$$g \circ \mathsf{str} = F(\mathsf{g} \circ \mathsf{str}) = F\mathbb{1}$$

```
lemma right_inv : A.str ≫ str_inv h = 𝟙 _ :=
by { rw [str_inv, ← (h.to ⟨ F.obj A.1 , F.map A.str ⟩).h,
  ← F.map_id, ← F.map_comp], congr, exact (left_inv h) }
```

5 W-types as initial algebras of their polynomial endofunctors

Mathlib contains the result that for any W-type `W_type β`,

```
    W_type β ≃ Σ (a : α), β a → W_type β
```

This is the obvious realization that that W-types are closed under the constructor operation: to make something on either side is just to give a pair $a : \alpha$ and $b : \beta a \to$ `W_type` β.

Notice that the right hand side looks like a polynomial endofunctor applied to `W_type` β. Knowing from the previous section that initial algebras are preserved by their functors, we might guess this is the right polynomial endofunctor to consider. Indeed,

```
/-- The polynomial endofunctor associated to a ‘W_type‘ -/
def polynomial_endofunctor : Type (max u₀ u₁) ⥤ Type (max u₀ u₁) :=
{ obj := λ X, Σ (a : α), β a → X,
  map := λ X Y f p, ⟨ p.1 , f ∘ p.2 ⟩ }
```

Functoriality is simply the composition: if I have βa many terms in X and I apply $f : X \to Y$ to each of them, then I obtain βa many terms in Y.

We can then make the W-type into an algebra over this polynomial endofunctor, naturally using the equivalence above for the structure map

```
def as_algebra : algebra (polynomial_endofunctor β) :=
{ A   := W_type β,
  str := W_type.of_sigma }
```

To show that this is the initial algebra we simply need to show that for any other algebra A we have a unique algebra map from this into A. This map should be defined by induction on the W-type: to map `W_type.mk a b` into A we reconstruct it using the structure map of A (and the inductively given image of `b`):

```
variables {β} (A : algebra (polynomial_endofunctor β))

/-- The map in ‘Type‘ from the initial algebra ‘W_type‘ to any other algebra -/
def lift_f : W_type β → A.A
| (W_type.mk a b) := A.str ⟨ a , λ x, lift_f (b x) ⟩

/-- The map in ‘endofunctor.algebra‘ from the initial algebra ‘W_type‘ to any other algebra -/
def lift : as_algebra β ⟶ A := { f := lift_f A }
```

The uniqueness of this map is also proven by induction:

```
lemma lift_uniq (f : as_algebra β ⟶ A) : f = lift A :=
begin
  ext w,
  induction w with a b hw,
  simp only [lift, lift_f],
  convert (congr_fun f.2 ⟨ a , b ⟩).symm,
  funext x,
  exact (hw x).symm,
end
```

Proof. (In words.) We supplied algebra morphisms with the `ext` attribute, so we simply need to compare the underlying functions in `Type`. This tactic also applies functional extensionality with a given term w : `W_type β`, so we are showing that the functions are equal on points. We induct on w so that we can definitionally reduce the inductively defined `lift_f` and use the induction hypothesis. At this point the goal is to show that "down right" in the below diagrams are equal (the diagrams commute by `lift` and `f` being algebra morphisms).

$$\begin{tikzcd}
P\ \texttt{W\_type} \arrow[r, "P\ \texttt{lift}"] \arrow[d, "\texttt{str}"'] & PA \arrow[d, "\texttt{str}"] \\
\texttt{W\_type} \arrow[r, "\texttt{lift}"'] & A
\end{tikzcd}
\qquad
\begin{tikzcd}
P\ \texttt{W\_type} \arrow[r, "Pf"] \arrow[d, "\texttt{str}"'] & PA \arrow[d, "\texttt{str}"] \\
\texttt{W\_type} \arrow[r, "f"'] & A
\end{tikzcd}$$

Hence it suffices that "right down" in the diagrams commute, which follows from the induction hypothesis. □

6 Computing polynomials

We conclude by showing that the polynomial endofunctors associated to ℕ and `list` γ are naturally isomorphic to $1 + X$ and $1 + \gamma X$. In order to do so we make a category of polynomial endofunctors (of this particular form) `W_type.cat`, as a full subcategory of the functor category `Type → Type` (strictly speaking the objects are pairs α, β rather than endofunctors, and their image under talking `polynomial_endofunctor` forms the full subcategory of the functor category).

```
structure cat :=
{α : Type*}
(β : α → Type*)

instance : category cat :=
{ hom := λ W₀ W₁, nat_trans (polynomial_endofunctor W₀.β) (polynomial_endofunctor W₁.β),
  id := λ _, nat_trans.id _,
  comp := λ _ _ _, nat_trans.vcomp }
```

We supply some nice instances to lean so that we can write out basic polynomials

```
/-- The endofunctor ‘Σ (x : γ), X ^ (fin n) ≃ γ X ^ n ‘ -/
@[simp] def monomial (γ : Type u₀) (n : ℕ) : cat := ⟨ λ x : γ, ulift (fin n) ⟩

/-- The identity endofunctor as a W_type, ‘X ≃ punit X ^ 1‘ -/
@[simp] def X : cat := monomial punit 1

/-- The polynomial endofunctor taking anything to ‘pempty‘,
since ‘pempty ≃ pempty X ^ 0‘ -/
@[simps] instance : has_zero cat := { zero := monomial pempty 0 }

/-- The polynomial endofunctor taking anything to ‘punit‘,
since ‘punit ≃ punit X ^ 0‘ -/
@[simps] instance : has_one cat := { one := monomial punit 0 }

/-- The constant functor going to a type ‘γ‘ is a polynomial
‘γ = Σ (a : γ) 1 = Σ (a : γ) X ^ pempty ‘ -/
instance : has_coe (Type u₀) cat.{u₀ u₀} := { coe := λ γ, monomial γ 0 }

/-- The sum of two polynomial endofunctors
‘Σ (a : α₀) X^(β a) + Σ (a : α₁) X^(β₁ a) ≃ Σ (a : α₀ ⊕ α₁) X^((β₀ ⊕ β₁) a)‘ -/
@[simps] def addition : cat.{u₁ u₀} → cat.{u₁ u₀} → cat.{u₁ u₀} :=
λ W₀ W₁, ⟨ sum.elim W₀.β W₁.β ⟩

@[simps] instance : has_add cat := { add := addition }
```

With the above we can enjoy easily writing something complicated like

$$\gamma_5 X^5 + \gamma_3 X^2 + X + \gamma_0 + 1$$

for any types γ_5, γ_3 and γ_0 and obtain their corresponding polynomial endofunctors. Now it is easy to state the way in which the polynomial endofunctors we produced in the theory are naturally isomorphic to what we expect.

```
/-- The polynomial endofunctor of the ‘W_type‘ for ‘ℕ‘ is ‘1 + X‘ (a.k.a the maybe monad) -/
def cat_nat_β_eq_one_add_X : cat.mk nat_β ≅ 1 + X := sorry

/-- The polynomial endofunctor for the ‘W_type‘ for ‘list γ‘ is ‘1 + γ X‘ -/
def cat_list_β_eq_one_add_type (γ : Type u₁) : cat.mk (list_β γ) ≅ 1 + monomial γ 1 := sorry
```

To prove the above, we make the obvious lemma: if α_0 and α_1 are equivalent and $\beta_0 : \alpha_0 \to$ Type and $\beta_1 : \alpha_1 \to$ Type are equivalent (along the equivalence between α_0 and α_1), then the W-types generated by each of them are isomorphic (i.e. their polynomial endofunctors are naturally isomorphic). We make use of the extensionality principle `nat_iso.of_components` for natural isomorphisms:

```
/-- Two W_types are isomorphic if their endofunctors are isomorphic -/
def iso_of_iso {W₀ W₁ : cat} (endo_iso : polynomial_endofunctor W₀.β ≅ polynomial_endofunctor W₁.β) :
  W₀ ≅ W₁ :=
{ hom := endo_iso.hom,
  inv := endo_iso.inv,
  hom_inv_id' := endo_iso.hom_inv_id,
  inv_hom_id' := endo_iso.inv_hom_id }.

/-- Two W_types are isomorphic if their defining types are equivalent -/
def iso_of_equiv {W₀ W₁ : cat} (hα : W₀.α ≃ W₁.α) (hβ : ∀ a : W₀.α, W₀.β a ≃ W₁.β (hα.to_fun a)) :
  W₀ ≅ W₁ :=
iso_of_iso (nat_iso.of_components _ _ ) -- a lot of simp lemmas
```

The isomorphisms `cat.mk nat_β ≅ 1 + X` and `cat.mk (list_β γ) ≅ 1 + monomial γ 1` follow, given some equivalences between the obvious types.

7 Further work

I am in the process of pushing the above to mathlib. In my next project I might work on the categorical generalisation of polynomial endofunctors, and try to show that they do in fact coincide with sums of exponentials (as we took the definition to be here) in the case where the category is Type. Another missing aspect of the above is families of inductive types, which could be an interesting direction to explore.