An implementation of polygraphs

Maxime Lucas

An implementation of polygraphs

Maxime Lucas *

INRIA

Laboratoire des Sciences du Numérique de Nantes
Université de Nantes, France

November 28, 2019

Abstract

Polygraphs are a higher dimensional structure which serves as a system of generators for higher categories. They are in particular used in higher dimensional rewriting, where they are used to present strict ∞-categories. However, any notion of higher dimensional structure (e.g. weak or strict ∞-categories or groupoids) comes with its own notion of polygraph.

In this paper, we describe an experimental implementation of polygraphs using the proof assistant Coq and prove some of their properties, following the types-as-∞-groupoids interpretation of homotopy type theory. We describe in particular a functor sending any polygraph to the free type it generates and parts of the adjunction between polygraphs and types.

Introduction

Polygraphs [4] (first defined by Street under the name of computads [11] [12]) are systems of generators for higher dimensional structures.

The implementation of polygraphs is an interesting challenge for three reasons. First, their definition involves induction-recursion, and the proof of most of their properties involves mutual induction. Those concepts are often hard to properly state on paper, and a proof assistant could be a valuable tool to check the validity of the proofs concerning polygraphs. For example, multiple papers claim that n-polygraphs form a presheaf category, which is false for $n \geq 3$, as noticed by Makkai and Zawadowski [8].

Second, the definition of polygraphs fundamentally relies on the notion of pushouts, which are implemented in proof assistants through higher inductive types (HITs). HITs are present natively in the various cubical provers (such as Cubical Agda, RedPRL, redtt or cubicaltt), and can be simulated in Coq and Agda. Polygraphs therefore serve as a good stress test to assert the usability of HITs in larger projects.

Finally, since they contain higher dimensional information, polygraphs themselves can be seen as presentation for HITs. In this sense, polygraphs can therefore be seen as first-order HITs, on which it is possible to quantify. This may prove useful in the development of homotopy type theory.

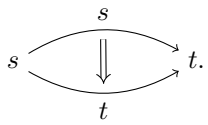
In this paper we investigate the implementation of polygraphs in Martin Löf's type theory with function extensionality extended with higher inductive types (HITs) and inductive-recursive (IR) types. In doing so, we pay close attention to the limitations of proof assistants in dealing with HITs.

To understand the notion of polygraph, let us consider first 1-categories. A system of generators for a 1-category is simply a given by the data of a graph $E_1 \rightrightarrows E_0$. The free category generated by such a

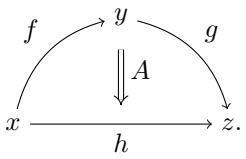
*maxime.lucas@inria.fr

graph is the category whose set of 1-cells is E_1^*, the set of paths in the graph.

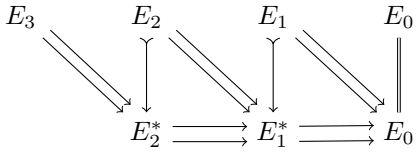
The situation gets more complicated for 2-categories. Any 2-globular set $E_2 \rightrightarrows E_1 \rightrightarrows E_0$ generates a free 2-category. For example taking E_2 to be the singleton set and $E_1 = E_0 = \{s, t\}$, one can form the *walking-2-cell* 2-category:



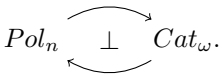
However, not all free 2-categories are generated by a 2-globular set: in a 2-globular set the source and target of a generator is again a generator, which does not hold for general free 2-categories. One example is the *triangle* 2-category, which consists in three generating 0-cells x, y, z, three generating 1-cells f, g, h and one generating 2-cell A:



2-polygraphs are a generalisation of 2-globular sets able to encode such a structure: instead of the source and target of a cell in E_2 living in E_1, we allow them to take their value in E_1^*. Denoting by E_2^* the set of 2-cells generated by this 2-polygraph, we can then continue this process and define a 3-polygraph to be given by an additional set E_3 taking its source and target in E_2^*. Putting all that together, a 3-polygraph can be represented by the following diagram:

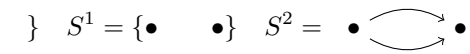


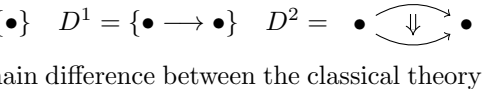
Iterating this construction, one gets an adjunction between the categories of polygraphs and ω-categories:



Batanin [1] noticed that the only property of ω-categories needed for the definition of polygraphs is that they form the Eilenberg-Moore category of a monad T on the category of globular sets. In other words, for any monad T on globular sets, there is a notion of T-polygraphs and an adjunction between T-polygraphs and T-algebras. This construction was later simplified by Garner [6].

A careful analysis of Garner's construction shows that the notion of polygraph can be extended to any category $\mathcal{C}$ equipped with a family $(i_n : A_n \to B_n)_{n\in\mathbb{N}}$ of arrow in $\mathcal{C}$, provided that $\mathcal{C}$ has pullbacks and pushouts. In the case where $\mathcal{C}$ is the category of ω-categories, the family i_n is the family of inclusions from the $(n-1)$-sphere S^n into the n-ball D^n.

$S^0 = \{\quad\}$ $S^1 = \{\bullet \qquad \bullet\}$ $S^2 =$ 

$D^0 = \{\bullet\}$ $D^1 = \{\bullet \longrightarrow \bullet\}$ $D^2 =$ 

The main difference between the classical theory of polygraphs and the one we implement is that types do not form a category in type theory, but rather an ω-category. In particular, function types are not hSets. It is unclear whether our failure to prove the full adjunction is due to this discrepancy or to technical limitations.

Plan of the paper Section 1 is devoted to the definition of polygraphs in the setting of strict ω-categories, using Garner's construction, which our implementation closely follows. Section 2 explains our implementation of pushouts using HITs. Finally in Section 3 we give an overview of our development and explain in particular the need for IR types.

1 Polygraphs

In this section, we closely follow Garner's construction of polygraphs, since it will be the one we will

follow in our implementation. Guiraud and Malbos' survey [7] provides a more elementary approach to polygraphs.

Definition 1.1. A *globular set* is the data, for all $n \in \mathbb{N}$, of a set G_n, together with maps $s, t : G_{n+1} \to G_n$ (respectively called the source and target operations), satisfying the equations $s \circ s = s \circ t$ and $t \circ s = t \circ t$. The set G_n is called the set of n-cells of G. Morphisms of globular sets are level-wise maps that preserve the source and target operations. We denote by *Glob* he category of globular sets.

An ω-category is a globular set G equipped with strictly unital and associative compositions $\circ_i$ defined on G_n for $0 \leq i < n$, satisfying in addition the exchange rule. We denote by Cat_ω the category of ω-categories, which is monadic over globular sets and we still denote by D^n (resp S^n) the free ω-category generated by D^n and S^n.

Definition 1.2. We denote by D^n the globular set of the n-disk. For $k < n$, $D^n_k = \{0, 1\}$, $D^n_n = \{d\}$ and D^n_k is empty for $k > n$. The source and target operations are given by $s(x) = 0$ and $t(x) = 1$ for all x.

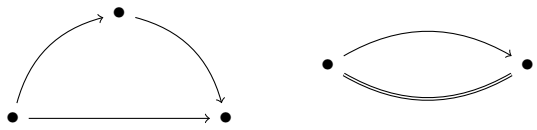
Similarly, S^n is the globular set of the $(n-1)$-sphere and is obtained from D^n by removing the unique n-cell.

Example 1.3. The category of globular sets is the presheaf category on the category so-called globe category. The objects of the globe category are the natural numbers, so that G_n is the value of the presheaf G at n, while D^n is then the representable associated to n. In particular, the Yoneda Lemma gives $Hom(D^n, G) = G_n$.

On the other hand, an element of $Hom(S^n, G)$ corresponds to pairs of $(n-1)$-cells (f, g) in G_{n-1} with same source and same target. We call such a pair an $(n-1)$-sphere in G. The inclusion map $i_n : S_n \to D_n$ induces a map $Hom(D^n, G) \to Hom(S^n, G)$, sending any cell $A \in G_n$ to the pair $(s(A), t(A))$.

Definition 1.4. A 0-polygraph is a set. A 1-polygraph Σ is a graph. We denote by Σ^* the category of paths on Σ.

Example 1.5. Take Σ to be the graph with one vertex and one edge. Then Σ^* is a category with exactly one object $\bullet$. Arrows from $\bullet$ to $\bullet$ are in bijection with $\mathbb{N}$, with composition given by addition. A functor $S^2 \to \Sigma^*$ therefore corresponds to a pair of integers (n, m). For example, $(2, 1)$ and $(1, 0)$ respectively correspond to the following 1-spheres in Σ^* (where each arrow denotes the unique arrow of Σ):



Remark 1.6. Note that there are two ways to describe the category Σ^* on a graph $\Sigma = (E, V)$, that we will respectively call the syntactic and the semantic one. The syntactic one consists in saying that if a and b are objects of Σ, then arrows from a to b are lists of composable arrows from a to b in Σ. This approach can be generalised for polygraphs generating strict ω-categories (see for example the work of Métayer [9][Section 4]): the n-cells of the strict ω-category generated by an n-polygraph will be well-formed terms on the generators of the polygraph, quotiented by suitable relations.

This approach however will not work in our setting, where we do not have a syntactic description of the terms of a generic type. Instead, we follow Garner's semantic approach [6][Definition 5.4], where Σ^* is described as the following pushout in the category of categories, where $(\_ \cdot \_)$ denotes the copower, defined as $E \cdot G = \coprod_{e \in E} G$:

$$\begin{array}{ccc} E \cdot S^1 & \xrightarrow{d} & V \\ {\scriptstyle f_2}\downarrow & \ulcorner & \downarrow \\ E \cdot D^1 & \longrightarrow & \Sigma^* \end{array}$$

Since S^1 is the category with two points, for any element $e \in E$ the source and target operations induce a map $S^1 \to V$. Assembling those maps together yields the map d of the previous diagram.

Definition 1.7. A 2-polygraph is a triple (Σ, E, d), where Σ is a 1-polygraph, E is a set, and d is a map:

$$d : E \cdot S^2 \to \Sigma^*,$$

Definition 1.8. The 2-category Γ^* generated by a 2-polygraph $\Gamma = (\Sigma, E, d)$ is the result of the following pushout in 2-categories:

$$\begin{array}{ccc} E \cdot S^2 & \xrightarrow{d} & \Sigma^* \\ {\scriptstyle f_2}\downarrow & \ulcorner & \downarrow \\ E \cdot D^2 & \longrightarrow & \Gamma^* \end{array}$$

More generally, the notion of n-polygraph is defined mutually recursively with the map which forms the free n-category generated by an n-polygraph. Although the definition can be given in one go, for computational reasons, we first give an intermediate notion of *augmentation*:

Definition 1.9. An *n-augmentation* over an ω-category X is a pair $A = (|A|, d)$, where X is an ω-category, $|A|$ is a set and d is a map:

$$d : |A| \cdot S^n \to X$$

Definition 1.10. The free ω-category A^* generated by an augmentation $A = (|A|, d)$ over an ω-category X is the result of the following pushout in the category of ω-categories:

$$\begin{array}{ccc} |A| \cdot S^n & \xrightarrow{d} & X \\ {\scriptstyle f_n}\downarrow & \ulcorner & \downarrow \\ |A| \cdot D^n & \longrightarrow & A^* \end{array}$$

Definition 1.11. We define by mutual induction the notion of n-polygraph and a map $(\_)^*$ from n-polygraphs to ω-categories as follows:

- A 0-polygraph is a set.
- If E is a 0-polygraph then $E^* := E$, seen as a discrete ω-category.
- An $(n+1)$-polygraph is a pair (Σ, A), where Σ is an n-polygraph and A is an $(n+1)$-augmentation over Σ^*.
- If $\Gamma = (\Sigma, A)$ is an $(n+1)$-polygraph, then we define Γ^* as A^*.

The goal of this paper is to present a formalisation in type theory of the following, due to Garner [6][Definition 5.4] for this particular construction of polygraphs.

Proposition 1.12. *For all $n \in \mathbb{N}$, n-polygraphs form a category Pol_n. The map $(\_)^*$ extends to a left adjoint functor $Pol_n \to Cat_\omega$, so that we get an adjunction:*

$$Pol_n \underset{U}{\overset{(\_)^*}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} Cat_\omega.$$

2 Pushouts and higher inductive types

Our strategy for implementing polygraphs closely follows the one outlined above: first we define a type of augmentations, then the free type they generate, and finally polygraphs together with the Free map.

The definition of augmentations is straightforward.

```
Inductive Aug (X : Type) (n : nat) : Type :=
  mkAug (E : Type)  (d : E × Sphere n → X) : Aug X n.
```

The first difficulty arises when one tries to define the map `FreeA : Aug X n → Type` assigning an augmentation to the free type it generates. Indeed Definition 1.10 involves a pushout, which do not exist in Martin-Löf type theory. Our solution consists in using HITs [13]. HITs are a generalisation of inductive types, where constructors are not only allowed to construct terms of the type, but also equalities between different terms. The HIT corresponding to pushouts would thus have three constructors:

```
Inductive Pushout {A B1 B2}
                  (f1 : A → B1) (f2 : A → B2) : Type :=
  inl : B1 → Pushout f1 f2
| inr : B2 → Pushout f1 f2
| incoh : ∀ a : A, inl (f1 a) = inr (f2 a).
```

Unfortunately, HITs are currently not available in the official distribution of the Coq proof assistant. Following the implementation of HITs in the HoTT library[2], we use the module mechanism of Coq to produce a type mimicking most of their properties. More precisely, inside a module we define a private inductive type and replace the last constructor of the HIT by an axiom.

```
Private Inductive Pushout {A B1 B2}
          (f1 : A → B1) (f2: A → B2) : Type :=
 inl : B1 → Pushout f1 f2
|inr : B2 → Pushout f1 f2.

Axiom incoh {A B1 B2} (f1 : A → B1) (f2: A → B) :
 ∀ a : A, inl (f1 a) = inr (f2 a).
```

The effect of the `Private` keyword is that the recursor of the `Pushout` type is not available outside of the module where it is defined. Instead, we define by hand a new recursor, thereby forcing maps from the pushout to respect the desired equality.

```
Definition Pushout_rect {A B1 B2}
           {f1 : A → B1} {f2 : A → B2}
           (P : Pushout f1 f2 → Type)
           (g1 : ∀ b1 : B1, P (inl b1))
           (g2 : ∀ b2 : B2, P (inr b2))
           (H : ∀ a, (incoh a) # g1 (f1 a) = g2 (f2 a))
           (x : Pushout f1 f2) : P x :=
  match x with
  |inl  b1 ⇒ (fun _ ⇒ g1 b1)
  |inr  b2 ⇒ (fun _ ⇒ g2 b2)
  end H.
```

The limitation of this technique is that this new recursor does not compute as expected on `incoh a`: while the argument `H` is supposed to encode the action of `Pushout_rect P` on `incoh a`, it is actually never used in the body of `Pushout_rect`. Instead, we need an additional axiom.

```
Axiom Pushout_rect_compute {A B1 B2}
           {f1 : A → B1} {f2 : A → B2}
           {P : Pushout f1 F2 → Type}
           {g1 : ∀ b1 : B1, P (inl  b1)}
           {g2 : ∀ b2 : B2, P (inr  b2)}
           {H : ∀ a, (incoh a) # g1 (f1 a) = g2 (f2 a)}
           {a : A} :
      apd (Pushout_rect P g1 g2 H) (incoh a) = H a.
```

We use a similar technique to define the spheres S^n and the balls D^n, although our construction does not really depend on it: all that is needed is a family of maps `inj n`, but the fact that those are the inclusion of the $(n-1)$-sphere into the n-ball does not play a role.

The definition of the free type on an augmentation is now straightforward:

```
Definition FreeA {F n} (aug : Aug F n) : Type :=
 match aug with
   | mkAug E d ⇒
     Pushout d
           λ (x : E × Sphere n) ⇒
(fst x, inj n (snd x))
 end.
```

3 Polygraphs and inductive recursive types

As in Definition 1.11, the type `Pol n` of n-polygraphs needs to be defined inductively, together with the free-type map `Free n : Pol n → Type`. The fact that `Free n P` is defined by recursion on `P` makes `Pol n` an inductive-recursive (IR) type [5]. Coq does not natively provide IR types. We therefore use an experimental branch of Coq containing IR types, developed by Matthieu Sozeau [10]. The definition of n-polygraphs is then straightforward:

```
Inductive Pol : nat → Type :=
|Disc : Type → Pol 0
|Ext {n} (P : Pol n) (aug : Aug (Free P) (S n)) : Pol (S n)
with fix Free {n : nat} (P : Pol n) : Type :=
      match P with
      | Disc A ⇒ A
      | Ext _ _ aug ⇒ FreeA aug
      end.
```

The definition of the notion of morphism of polygraphs follows the same template. First given `aug` and `aug'` being n-augmentations respectively over `F` and `F'` and given `f : F → F'`, we define an inductive type `MAug f aug aug'` of morphisms from `aug` to `aug'` over `f`. Then we define a map `FreeMA` from `Maug f aug aug'` to `FreeA aug → FreeA aug'`. Finally we define the type of morphisms of polygraphs,

```
Inductive MPol {n} : (Pol n) → (Pol n) → Type
with fix FreeM {n P P'} (m : MPol P P') :
                (Free P) → (Free P').
```

We have now defined the objects and the arrows of the category of polygraphs, together with the action on them of the `Free` functor. Similarly, composition of morphisms of polygraphs has to be defined together with the proof that the `Free` functor preserves composition. This is done using a dependent pair:

```
Fixpoint PolyComposeAndFree {n P Q R}
                    (G : MPol P Q) (H : MPol Q R) :
  ∃ Comp : MPol P R, FreeM Comp = (FreeM H) o (FreeM G).
```

The rest of the development follows much the same pattern, with operations on polygraphs needing to be defined together with some of their properties. More precisely, we define successively:

1. The identity morphism on polygraphs

   ```
   idPol {n} (P : Pol n) : MPol P P
   ```

   together with the fact that the free functor preserves the identity:

   ```
   FreeU {n} (P : Pol n) :
         FreeM (idPol P) = (λ x ⇒ x).
   ```

2. The forgetful functor on types

   ```
   Forget (n : nat) (T : Type) : Pol n.
   ```

   together with the Counit of the adjunction between `Pol n` and `Type`:

   ```
   Counit (T : Type) : Free (Forget n T) → T.
   ```

3. The action of the forgetful functor on arrows

   ```
   ForgetM {n T T'} (f : T → T') :
     MPol (Forget n T) (Forget n T')
   ```

   together with the naturality of the counit:

   ```
   CounitM {n T T'} (f : T → T') :
     (Counit T' n) o (FreeM (ForgetM f))
       = f o (Counit T n).
   ```

4. The unit of the adjunction

   ```
   Unit {n} (P : Pol n) : MPol P (Forget n (Free P))
   ```

   together with the first triangular identity

   ```
   Triangle1 {n} (P : Pol n) :
         (Counit (Free P) n) o (FreeM (Unit P))
          = (λ x ⇒ x)
   ```

Conclusion

Polygraphs form an intricate mathematical structure, whose implementation calls forth features of proof checkers rarely used together: IR types and HITs. Apart from prototype implementations, the only proof assistants with native IR types are Agda and Idris. As for HITs, they are natively present in the various cubical theorem provers (such as Cubical Agda, RedPRL, redtt or cubicaltt) and in Lean. They are also implemented through similar tricks as the ones used here in several libraries, including the Coq HoTT library [2] and the HoTT-Agda library [3].

Our takeaway from this implementation is that HITs that do not naively compute are not well-suited for the more intricate uses of HITs. The main shortcomings come from the size of the terms encountered. Indeed, because the eliminator we defined for the pushout types does not compute on equalities, any such computation has to be replaced by a transport along the `Pushout_rect_compute` axiom. While this is tractable on the first definitions of our development, in more complicated proofs the size of the terms considered quickly becomes too much to handle, both for the human prover and for the automated checker, as can be seen from the compilation tymes: around 45s for less than a thousand lines of code.

Interestingly, a tentative implementation of polygraphs in Cubical Agda (where HITs natively compute) did not fare any better: although the terms considered were smaller, the absence of tactics made them much harder to manipulate. As a result, we were not even able to prove that the `Free` functor preserves composition.

References

[1] M. A. Batanin. Computads for finitary monads on globular sets. In *Higher category theory (Evanston, IL, 1997)*, volume 230 of *Contemp. Math.*, pages 37–57. Amer. Math. Soc., Providence, RI, 1998.

[2] A. Bauer, J. Gross, P. L. Lumsdaine, M. Shulman, M. Sozeau, and B. Spitters. The hott library: A formalization of homotopy type theory in coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 164–172, New York, NY, USA, 2017. ACM.

[3] G. Brunerie, K.-B. Hou (Favonia), E. Cavallo, T. Baumann, E. Finster, J. Cockx, C. Sattler, C. Jeris, M. Shulman, et al. Homotopy type theory in Agda.

[4] A. Burroni. Higher-dimensional word problems with applications to equational logic. volume 115, pages 43–62. 1993. 4th Summer Conference on Category Theory and Computer Science (Paris, 1991).

[5] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symbolic Logic*, 65(2):525–549, 2000.

[6] R. Garner. Homomorphisms of higher categories. *Adv. Math.*, 224(6):2269–2311, 2010.

[7] Y. Guiraud and P. Malbos. Polygraphs of finite derivation type. *Math. Structures Comput. Sci.*, 28(2):155–201, 2018.

[8] M. Makkai and M. Zawadowski. 3-computads do not form a presheaf category. *Journal of Pure and Applied Algebra*, 212(11):2543–2546, 2008.

[9] F. Métayer. Cofibrant objects among higher-dimensional categories. *Homology, Homotopy and Applications*, 10(1):181–203, 2008.

[10] M. Sozeau. Inductive recursive types in Coq.

[11] R. Street. Limits indexed by category-valued 2-functors. *J. Pure Appl. Algebra*, 8(2):149–181, 1976.

[12] R. Street. The algebra of oriented simplexes. *J. Pure Appl. Algebra*, 49(3):283–335, 1987.

[13] The Univalent Foundations Program. *Homotopy type theory. Univalent foundations of mathematics.* Princeton, NJ: Institute for Advanced Study; Raleigh, NC: Lulu Press, 2013.